GORAM: Graph-oriented ORAM for Efficient Ego-centric Queries on Federated Graphs

Xiaoyu Fan
Tsinghua University and Ant Group
fxy23@mails.tsinghua.edu.cn

Kun Chen
Ant Group
ck413941@antgroup.com

Jiping Yu
Tsinghua University and Ant Group
yjp19@mails.tsinghua.edu.cn

Xiaowei Zhu
Ant Group
robert.zxw@antgroup.com

Yunyi Chen
Tsinghua University and Ant Group
cyy23@mails.tsinghua.edu.cn

Huanchen Zhang
Tsinghua University and Shanghai Qi Zhi Institute
huanchen@tsinghua.edu.cn

Wei Xu
Tsinghua University and Shanghai Qi Zhi Institute
weixu@tsinghua.edu.cn

ABSTRACT

Ego-centric queries, focusing on a target vertex and its direct neighbors, are essential for various applications. Enabling such queries on graphs owned by mutually distrustful data providers without breaching privacy holds promise for more comprehensive results.

In this paper, we propose GORAM, a graph-oriented data structure that enables efficient ego-centric queries on federated graphs with strong privacy guarantees. GORAM leverages *secure multi-party computation (MPC)* and ensures that no information about the graphs or the querying keys is exposed during the process. For practical performance, GORAM partitions the federated graph and constructs an *Oblivious RAM (ORAM)*-inspired index atop these partitions. This design enables each ego-centric query to process only a single partition, which can be accessed fast and securely.

Utilizing GORAM, we develop a prototype querying engine on a real-world MPC framework. We then conduct a comprehensive evaluation using five commonly used queries similar to the LinkBench workload description [11] on both synthetic and real-world graphs. Our evaluation shows that all five queries can be completed in just 58.1 milliseconds to 35.7 seconds, even on graphs with up to 41.6 million vertices and 1.4 billion edges. To the best of our knowledge, this represents the first instance of processing billion-scale graphs with practical performance on MPC.

PVLDB Reference Format:
Xiaoyu Fan, Kun Chen, Jiping Yu, Xiaowei Zhu, Yunyi Chen, Huanchen Zhang, and Wei Xu. GORAM: Graph-oriented ORAM for Efficient Ego-centric Queries on Federated Graphs. PVLDB, 18(10): 3601 - 3614, 2025.
doi:10.14778/3748191.3748218

Proceedings of the VLDB Endowment, Vol. 18, No. 10 ISSN 2150-8097.
doi:10.14778/3748191.3748218

* Wei Xu, Huanchen Zhang, and Kun Chen are the corresponding authors.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Fannxy/GORAM-ABY3.

1 INTRODUCTION

Privacy-preserving federated query engines allow data providers to collaboratively compute the query results while keeping the inputs and all the intermediate results private. The purpose of this paper is to support federated queries on graphs, specifically the crucial *ego-centric* queries that target a specified vertex and all its direct neighbors. This basic type of query has numerous applications, particularly when multiple parties can cooperate. For example, by analyzing the relations among suspicious accounts, banks can detect money laundering from the transaction graphs [46, 54]. As LinkBench [11] reports, neighbor filtering on the given account, a typical ego-centric query, constitutes 55.6% of all queries at Meta. In fact, all queries in LinkBench are ego-centric.

The above motivating examples demonstrate that it is crucial to keep both the graph and the query keys private. Ego-centric queries are efficient in plaintext but rely on optimizations like popular vertex caching [19, 65], which conflict with privacy requirements. Unlike tabular data, where the schema is public and we only need to protect the data and query keys, the *graph structure* information also needs to be protected. E.g., whether a particular vertex or edge exists and the distribution of vertex degrees. For example, disclosing an edge in the transaction graph could expose sensitive relationships between accounts, thus necessitating protection.

One typical method to implement private queries on federated data is to use *secure multi-party computation (MPC)* [66] throughout the query process [12, 13, 17, 41, 59]. MPC is a cryptographic technique that allows multiple parties to jointly compute a function on their private inputs, guaranteeing that no information is leaked.

The straightforward idea for graph is to encode the entire adjacency matrix in MPC [16], thereby hiding the entire graph. Unfortunately, it requires $O(|V|^2)$ space, where $|V|$ is the number of vertices. As real-world graphs are sparse [26], allocating space for all the possible edges is impractical. Nayak et al. [52] hide graph structure by encoding every vertex and edge as the same encrypted tuple containing a source and destination ID (a vertex stores its ID twice). This structure reduces the space overhead to $O(|V| + |E|)$, where $|E|$ is the number of edges. Given that $|V| + |E| \ll |V|^2$ for real-world graphs, this representation is popular [10, 39, 47, 48, 52]. However, protecting the query key requires scanning the entire list, incurring $O(|V| + |E|)$ time.

We observe that despite the space-inefficiency, using *Oblivious RAM (ORAM)* [32, 67], which allows accessing the ith element of an array in sublinear time without revealing i, we can secretly access any element of an adjacency matrix sublinearly. Our key idea is that if we can split the edge list into multiple partitions and build a matrix on top of the partitions, we can reduce the space size and only scan one partition for each query.

We propose GORAM, a graph-oriented data structure for efficient ego-centric queries on large federated graphs with strong privacy guarantees. GORAM splits the vertices into multiple chunks and segments the graph into a "matrix" of edge lists. Each edge list contains all edges starting from vertices in the row's chunk and destinations in the column's. The graph is then organized as multiple *partitions*, each containing all the information needed for each ego-centric query. Logically, all partitions together contain the entire edge list, and the matrix serves as a secure *index*. Using GORAM, we can locate a particular partition sublinearly, and we only need to scan the partition, greatly reducing running time (Section 4.2). While current ORAM only supports addressing a single element, we extend the idea to access a full partition, maximizing benefits from vectorization and parallelism (Section 4.3). While GORAM is designed agnostic to underlying MPC protocols, we find that some widely used protocols, e.g., ABY3 [50], allow us to design a constant-round shuffling protocol, vastly accelerating ORAM initiation (Section 4.4). Built on GORAM, we implement five ego-centric queries, including *edge existence*, *1-hop neighbors*, *neighbors filtering*, etc., covering all LinkBench queries [11](Section 5).

We evaluate the above queries on three real-world and thirty synthetic graphs with varied distributions and sizes. Results in Section 7 show remarkable efficiency and scalability of GORAM. On the largest graph, Twitter [18], with more than 41.6 million vertices and 1.4 billion edges, all queries complete within 35.7 seconds, with the fastest taking only 58.1 milliseconds, showing 2 to 3 orders of magnitude speedup over the existing secure graph data structures. Initialization only requires less than 3.0 minutes. We provide a detailed performance analysis using the synthetic graphs and show that GORAM outperforms the existing data structures across varied distributions and sizes. To our knowledge, GORAM is the first to support secure computation on graphs with more than one *billion* edges - 2-orders-of-magnitude larger than the prior arts [10, 39, 48].

In summary, our contributions include:

(1) We propose GORAM, a graph-oriented data structure to support efficient sublinear ego-centric queries on federated graphs, guaranteeing strong privacy.

(2) We design comprehensive optimizations for practical performance on large-scale graphs, including local processing, lifecycle parallelisms, and a constant-round shuffling protocol.

(3) We develop a prototype secure querying engine based on GORAM and evaluate it comprehensively using five commonly used queries on 33 synthetic and real-world graphs, demonstrating remarkable efficiency and scalability.

2 CRYPTOGRAPHY BACKGROUND

2.1 Secure Multi-party Computation (MPC)

MPC allows multiple distrusting parties to jointly compute a function while keeping the inputs private.

Secret sharing is popular in MPC [66]. A (t, n)-*secret sharing* schema splits data x to n parties, satisfying that any t parties can reconstruct x while fewer parties learn nothing about x. Similar to [10, 29, 41, 50], GORAM adopts the efficient $(2, 3)$ *boolean* secret sharing. This scheme splits x into (x_1, x_2, x_3) with each x_i ($i \in \{1, 2, 3\}$) being uniformly random and $x \equiv x_1 \oplus x_2 \oplus x_3$ ($\oplus$ denotes bitwise XOR). Each party i owns two shares (x_i, x_{i+1}) cyclically. Thus, one party learns nothing about x, whereas any two can reconstruct it. Denote the boolean secret shares (x_1, x_2, x_3) as $[\![x]\!]$.

Secure operations. For a variety of operations (op) such as XOR, AND, OR, +, ×, and comparisons ($>, \geq, =$), we can use the MPC protocols (OP) to compute $z = \mathsf{op}(x, y)$ collaboratively and securely: $[\![z]\!] = \mathsf{OP}([\![x]\!], [\![y]\!])$, where efficient arithmetic operations rely on transforming $[\![x]\!]$ into $[\![x]\!]^A$ such that $x \equiv x_1^A + x_2^A + x_3^A \pmod{2^k}$ [27, 50]. $[\![x]\!]^A$ denotes the arithmetic shares. Except for XOR and +, other operations require at least one round of communication among the computation parties. It is common to batch the operations to amortize the communication cost [41, 50].

Security guarantees. A well-designed MPC protocol guarantees privacy and correctness against an *adversary model*. Common classifications include *semi-honest* vs. *malicious*, i.e., whether the corrupted parties can deviate from the protocol, and *honest*- vs. *dishonest*- majority, contingent on the corruption proportion. GORAM adopts all its MPC protocols from ABY3 [50] and Araki et al. [10], thus inheriting their semi-honest and honest-majority adversary model, same as [10, 29, 41, 50].

2.2 Oblivious RAM (ORAM)

Oblivious RAM (ORAM) [32] implements oblivious *indexing*, i.e., accessing the ith element in an array while keeping i secret. ORAM offers two desirable properties: (1) it hides the access patterns, i.e., for *any* two indices i, j, servers performing the access cannot distinguish whether the index is i or j, protecting the privacy of the query keys; (2) it enables sub-linear-complexity accesses, thereby providing scalability. The classic ORAM, e.g., Path ORAM [57], is designed for the *client-server* scenario, where a single client stores and retrieves her private data on a single untrusted server [32].

Distributed ORAM (DORAM) extends ORAM to a multi-party setting by secret-sharing the data across several computation servers [45]. Given a secret-shared index $[\![i]\!]$, the servers can jointly access $[\![arr[i]]\!] = [\![arr]\!][[\![i]\!]]$ without leaking the index. We build the index of GORAM by extending the classic Square-root ORAM [67]. Note that GORAM is agnostic to the DORAM implementations, and we choose Square-root ORAM because it has two properties in addition to simplicity: (1) it offers sublinear access time, and (2) it circumvents the reliance on any specific "MPC-friendly" ciphers like LowMC [7] that was later cryptanalyzed by [42]. Section 4.1 provides more details about Square-root ORAM and Section 8 discusses other DORAM structures [20, 28, 29, 58, 67].

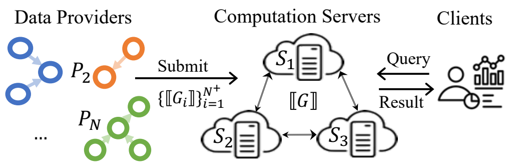

Figure 1: Logical Roles in Private Querying Process

Differences between classic ORAM and DORAM. Compared with the classic *client-server* ORAM, DORAM is an MPC protocol that securely implements a RAM functionality on top of secret-shared data held by multiple computation servers. Therefore, DORAM can be used in our federated setting, where the graph is owned by multiple distrusting data providers. All the ORAMs in the following denote Square-root ORAM.

3 OVERVIEW

GORAM is a query engine designed to support private, ego-centric queries on federated graphs. There are three types of roles: (1) an arbitrary number of *data providers*, each holding a part of the graph, i.e.,*private graph*; (2) three *computation servers* that run the MPC protocol to process queries; and (3) an arbitrary number of *clients* who submit queries to the computation servers and receive the results. Similar to [41], the roles are decoupled. Each party can hold any combination of *different* roles, e.g., both a data provider and a server. GORAM satisfies the following requirements:

(1) Functionality: we want to support *arbitrary* ego-centric queries, allowing us to apply any filter or aggregation to a target vertex or edge along with all its direct neighbors.

(2) Privacy: we want to keep two information private: (a) the query keys, i.e., the target vertices or edges; and (b) the graph, including the graph structure and the attributes of vertices and edges. Also, the client who learns the result cannot infer which data provider contributed to the result.

(3) Scalability: we want to support real-world billion-edge graphs and anticipate responses within a few seconds.

3.1 Formalization

Private ego-centric queries on federated graphs. We assume a *global* directed graph $G = (V, E)$ is distributed among N data providers $P_i, i \in [N], [N] = \{1, 2, ..., N\}$. V and E denote the vertex and edge sets, respectively. Each edge $e \in E$ has a source and a destination vertex, v_s and v_d. We assume the edges (v_s, v_d) are different because they may contain different attributes, e.g., timestamps [11]. We transform undirected graphs into directed ones by representing each edge with two directed edges.

Each *data provider* P_i owns a private graph $G_i = (V_i, E_i)$, satisfying that $V_i \subseteq V$, $E_i \subseteq E$. $E = \cup_{i \in [N]} E_i$ is the set of all edges, and it is always private. V is the set of all possible vertices, which is public. Because the complexity of GORAM is independent of $|V|$, we can safely set it to all the possible vertex names (e.g., a 64-bit ID), without leaking any information about the vertex set. For data structures that cannot support a large V, e.g., adjacency matrix, we set V as the union of all V_i's and make it public. There are three semi-honest *computation servers* S_i, $i \in \{1, 2, 3\}$ holding the secret global graph $[\![G]\!] = (V, [\![E^+]\!])$ and carrying all the query processing. $E^+ \supseteq E$ is a superset of E because it may contain dummy edges for privacy.

Each *client* can submit an ego-centric query with a secret key of either a vertex $[\![v]\!]$ or edge $([\![v_s]\!], [\![v_d]\!])$ $(v, v_s, v_d \in V)$ to the computation servers and receive the results. The ego-centric query can be an arbitrary filter or aggregation on sub-graph $G_{\text{sub}} = (V_{\text{sub}}, E_{\text{sub}})$ containing all the direct neighbors of the target vertex v and the corresponding edges (v, v^*) if $(v, v^*) \in E$, or all edges $(v_s, v_d) \in E$. The relations of the logical roles are shown in Figure 1.

3.2 Security Properties

Threat model. We assume all roles *semi-honest* by inheriting the threat model from the underlying ABY3 protocols, and there exists an adversary who can compromise at most one computing server and see all of its internal states, similar to [10, 12, 29, 41]. Note that GORAM is agnostic to the underlying MPC settings and can be adapted to settings with different numbers of parties or threat models (e.g., malicious) once the required protocols are available. Also, we assume the authentication of data providers is conducted through the *ring signature* [53], a cryptographic technique enabling anonymous authentication.

Security guarantees. GORAM provides two guarantees: (1) *clients' query key privacy*: no other party can learn anything about the client's query key; (2) *data providers' graph privacy*: no other party can learn any information about the data provider's private graph, which includes the graph structure and the attributes of vertices and edges. Additionally, the client who receives the query result learns nothing except the result, including which provider contributed to it. Section 6.1 shows how GORAM ensures these guarantees.

3.3 Strawman Solutions

Two classic data structures are used to present the secure graph, i.e.,*adjacency matrix* (*Mat*) [16] and *edge list* (*List*) [10, 47, 48, 52].

Based on *Mat*. With the public vertex set V, data providers can locally construct the $|V|^2$ adjacency matrix, encrypt it into the secret-shared matrix, and transfer the shares to the computation servers. Each server adds up the N secret matrices to form the secret matrix of the global graph G. Because *Mat* is a $|V| \times |V|$ matrix, we can directly adopt the ORAM for efficient access. Specifically, we build two ORAMs, adj-VORAM, operating over $|V|$ matrix rows (source vertices), and adj-EORAM, operating over an array of $|V|^2$ matrix elements (edges). For a vertex query on v_i, we access the $[\![i]\!]$th row from the adj-VORAM, which contains $|V|$ elements, each representing the edge number between v_i and $v_j, j \in [|V|]$. For an edge query on (v_i, v_j), we access the element $[\![i * |V| + j]\!]$ from the adj-EORAM, which contains the number of edge (v_i, v_j). *Mat* is simple but never practical because: (1) It requires $O(|V|^2)$ space cost, which is impractical for real-world sparse graphs [6, 11, 26, 64]. (2) To limit V, we need to expose the actual vertex set, instead of just the possible namespace, public to all parties, leaking the information of the global set (e.g., the global customer list). (3) We cannot support multi-graph, i.e., edges with different attributes, which is desirable in applications like transactions.

Based on *List*. Each data provider P_i can simply create a secret-shared list of its edges $([\![u]\!], [\![v]\!])$, and other attributes like timestamps. Unlike the prior $O(|V| + |E|)$ vertex-edge tuple lists [52], ego-centric queries require only $O(|E|)$ edge list, further reducing the storage. The compute servers can then concatenate the N secret

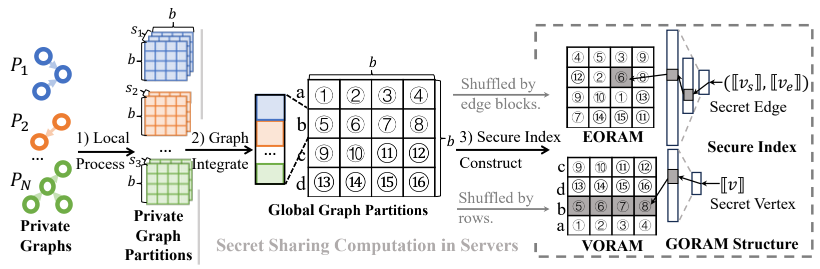

Figure 2: GORAM **Initialization and Structure Overview**

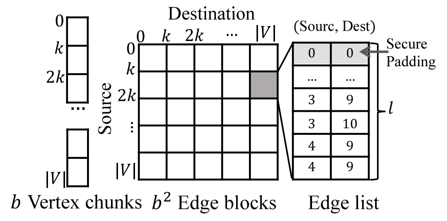

Figure 3: Graph Partition ($k = 2, |V| = 10$)

lists to create the global edge list. The only information leaked to the servers is each provider's edge count, which can be protected by appending extra ϵ_i dummy edges, i.e., ($[\![0]\!], [\![0]\!]$). Although the edge list is much more compact, we must scan the entire *List* for each ego-centric query because all the edges are encrypted, which introduces $O(|E|)$ complexity.

In summary, the two strawman solutions are either too costly on storage or on access time. A viable solution should allow a compact space while supporting sublinear access complexity.

3.4 GORAM **Design**

The idea of GORAM is to split the graph into a "matrix" of edge lists. The matrix structure enables building ORAMs on top of the graph, circumventing the need for a full scan for each query. The use of internal edge lists averts the $O(|V|^2)$ space complexity. This approach seeks to achieve both space- and query-efficiency.

GORAM **Overview.** As Figure 2 shows, GORAM is a secret-shared data structure of the global graph $[\![G]\!]$, held by the computation servers. GORAM randomly shuffles the public vertex set V, splits it into b vertex chunks and splits $[\![G]\!]$ into a "matrix" of b^2 blocks, each block contains *all* the edges starting from and ending in two vertex chunks, and each row of the blocks contains *all* the direct neighbors of a vertex chunk. This gives us two types of *partitions*: the block forms the partition for edge-centric queries, and the row of the blocks constitutes the partition for vertex-centric queries. GORAM then constructs VORAM and EORAM, which can securely access the partition given the secret vertex or edge.

GORAM can be efficiently initialized through three steps: (1) each data provider locally processes their private graph into secret-shared partitions; (2) the computation servers integrate all the partitions of private graphs into the global graph $[\![G]\!]$; and (3) the computation servers construct the secure indices for the partitions. Section 4 provides the details. Through GORAM, we can implement arbitrary ego-centric queries easily. Clients submit a query with a secret query key. For each query, computation servers receive the secret key, find the partition using GORAM, compute the query result on the partition, and return the result to the client. Section 5 shows five examples.

4 GRAPH-ORIENTED ORAM (GORAM)

To satisfy the requirements in Section 3, GORAM splits the global graph G into a "matrix" of edge lists, forming a 2d-partitioned data structure, as Figure 3 shows. This structure groups every successive k vertices into a chunk according to randomly shuffled IDs from the range $[|V|] = \{1, 2, \ldots, |V|\}$, thereby creating $b = \lceil \frac{|V|}{k} \rceil$ chunks. Then, it splits the global graph into b^2 blocks of edge lists for each pair of chunks. Specifically, each block (s, d) contains all the edges $\{(v_s, v_d)\}$, with v_s and v_d belonging to the sth and dth chunk, respectively. To ensure security, each block is equalized in length with dummy edges, making the blocks indistinguishable.

It is worth noting that if an edge (v_s, v_d) exists, it is contained in a single edge block $(\lceil \frac{v_s}{k} \rceil, \lceil \frac{v_d}{k} \rceil)$. For each vertex v, all the direct outing neighbors[1] are included in the $\lceil \frac{v}{k} \rceil$th row of b blocks. We heuristically choose a k based on the density of the graph because many real-world graphs, like social graphs or transaction graphs, have well-known characters D [26]. Given that $D = \frac{|E|}{|V|}$ is the density of the graph, we use a default $k = \frac{|V|}{\max(D,8)}$, making b independent of $|V|$. The $\max(D, 8)$ is to ensure that the number of partitions is not too small to cause frequent ORAM re-initialization demands (see Section 4.1 for details). The block and row of blocks are the graph *partitions* for edge- and vertex-centric queries, respectively. Obviously, *only* one partition needs to be processed for each query. GORAM then builds the partitions as ORAMs to enable secure and fast access.

4.1 Preliminaries of Square-root ORAM

Square-root ORAM [67] enables oblivious access to a secret-shared array $[\![D]\!] = \{[\![D_0]\!], [\![D_1]\!], \ldots, [\![D_{n-1}]\!]\}$ at a secret-shared index $[\![i]\!]$ (i.e., accessing $[\![D_i]\!]$). The key idea is to shuffle $[\![D]\!]$ to $[\![\widetilde{D}]\!] = \pi([\![D]\!])$ using a random permutation π, and then build a secret-shared index map on π that translates a secret logical index $[\![i]\!]$ to the corresponding *plaintext* physical index p ($D_i \equiv \widetilde{D}_p$), thereby enabling direct access. Specifically, after shuffling $[\![\widetilde{D}]\!] = \pi([\![D]\!])$, Square-root ORAM constructs the secret-shared permutation representation $[\![\vec{\pi}]\!]$, where $\vec{\pi}_i$ records the location of D_i in $\widetilde{D}$. We refer to this procedure as ShuffleMem and Section 4.4 provides more details.

For each secret index $[\![i]\!]$, Square-root ORAM computes the index $[\![p]\!] = [\![\vec{\pi}_i]\!]$, reveals $[\![p]\!]$ to plaintext and access $[\![\widetilde{D}_p]\!]$ to obtain the secret-shared element $[\![D_i]\!]$. Note that computing $[\![p]\!] = [\![\vec{\pi}_i]\!]$ given $[\![i]\!]$ is equivalent to an ORAM access problem, with data changes from $[\![D]\!]$ to $[\![\vec{\pi}]\!]$. Therefore, Square-root ORAM recursively applies the same idea to build the index map of $[\![\vec{\pi}]\!]$ as ORAMs: it packs every successive P elements of $[\![\vec{\pi}]\!]$ into a single element, shuffling and creating smaller ORAM on the packed $\frac{|\vec{\pi}|}{P}$ elements recursively until the size is no greater than a threshold T (SectionIII D [67]).

Obliviousness guarantee. Each different logical index $[\![i]\!]$ reveals a different random plaintext physical index p, thereby achieving

[1]If the bidirectional neighbors are interested, the incoming neighbors are included in the $\lceil \frac{v}{k} \rceil$th column of the 2d-partition.

oblivious access pattern. To address the issue when repeating the same logical indices, Square-root ORAM employs a *stash* that stores the accessed elements (SectionIII C [67]). For each logical index, we first scan the stash to check for a match, then access the ORAM using an updated secret index. If the logical index is not in the stash, the update secret index remains the original logical index. Otherwise, it is equal to an unused random index [67]. Every access consists of a linear scan over the stash and a random ORAM access, making any two indices indistinguishable.

Re-initialization. Only when the stash reaches capacity, we need to rebuild the ORAM, which is different from other ORAM schemes that require a re-initialization after each access. By default, the stash size is set to $T = \sqrt{n}$, n is the ORAM element count, which is the same as the recursive threshold according to SectionIII D [67]. For read-only accesses, ORAMs can be built continuously in the background, allowing immediate replacement with an unused ORAM when the stash is full. Including the optimized $O(n)$ rebuilding (Section 4.4), the average access cost over T elements is $O(PT\log(\frac{n}{T}))$.

4.2 GORAM Initialization and Access

The GORAM initialization involves three steps:

Step 1 - Each data provider locally partitions the private graph. Because the global V is public, each data provider P_i can locally split the private graph G_i into the same b^2 blocks according to the public k. Each P_i initializes b^2 blocks using V regardless of the vertices actually owned in the private V_i. P_i then traverses the edge list E_i, pushing each edge (v_s, v_d) to block (s, d), where $\lceil \frac{v_s}{k} \rceil = s$ and $\lceil \frac{v_d}{k} \rceil = d$, and sorting each block using the key $v_s||v_d$ (i.e., concatenation of the source and destination vertices). A sorted order is beneficial for some queries, e.g., `NeighborsGet` in Section 5.

To protect the edge distributions across blocks, P_i pads each block with dummy edges, i.e., $(0, 0)$, to align all blocks to the maximum block size l_i, thereby forming a $(b \times b \times l_i)$ graph partition. Additionally, to protect the variations in l_i across data providers, P_i standardizes the partition size to a uniform parameter $\bar{l}$, which is publicly set to a default value of 8 to avoid excessive padding. Specifically, if $l_i \le \bar{l}$, P_i pads $\bar{l} - l_i$ dummy edges to each block. Otherwise, P_i splits the $(b \times b \times l_i)$ partition into $s_i = \lceil \frac{l_i}{\bar{l}} \rceil$ sub-partitions, each of size $(b \times b \times \bar{l})$. P_i then encrypts each sub-partition into secret shares and sends the shares to the computation servers through a *separate* anonymous channel, authorized via the *ring signature* [53].

Step 2 - MPC servers globally integrate the private partitions. In the view of the computation servers, $N^+ = \sum_{i=1}^{i=N} s_i$ $(b \times b \times \bar{l})$ secret matrices are received, each of which is random and thereby indistinguishable from others. Denote each matrix as $[\![G_u.\mathsf{Block}]\!]$. Because of the anonymous channels, the servers cannot identify which $[\![G_u.\mathsf{Block}]\!]$ comes from which data provider, thus protecting the maximum block size l_i of each P_i. The servers run b^2 secure *odd_even_merge_sort* networks [38] concurrently to merge blocks from each $[\![G_u.\mathsf{Block}]\!]$, thereby constructing the global secret partitioned graph, $[\![G.\mathsf{Block}]\!]$. The size of $[\![G.\mathsf{Block}]\!]$ is $(b \times b \times l)$, $l = N^+\bar{l}$.

Step 3 - MPC servers construct the secure index. To enable secure and efficient access to the target partition for each query, GORAM builds the partitions as ORAMs. Specifically, we model the partitioned graph $[\![G.\mathsf{Block}]\!]$ with two ORAMs for vertex- and edge-centric queries as shown in Figure 2: (1) VORAM models $[\![G.\mathsf{Block}]\!]$ as an array of b partitions, each is one row of b blocks; and (2) EORAM models $[\![G.\mathsf{Block}]\!]$ as an array of b^2 partitions, each is an edge block. The indices of EORAM are the flattened indices of the blocks, i.e., the index of block (i, j) is $ib + j$.

The two sub-ORAMs are initialized in the following way, similar to ORAM: (1) Shuffling the partitions according to a random permutation π, and storing the secret permutation representation $[\![\vec{\pi}]\!]$. We refer to this procedure as ShuffleMem, which is the primary bottleneck. Section 4.4 provides a constant-round ShuffleMem for the $(2, 3)$-secret shares to optimize this step. (2) Constructing the index map that translates the logical index $[\![i]\!]$ into a physical index p pointing to the target partition, which is the same as Section 4.1.

Access the partition. After the initialization, the MPC servers can jointly access GORAM as follows: (1) Given the target vertex $[\![v]\!]$ (from the clients), the servers can compute the partition index $[\![\lceil \frac{v}{k} \rceil]\!]$ and obtain the partition containing bl secret edges by accessing VORAM. This partition contains *all* the direct neighbors of $[\![v]\!]$. (2) Given the target edge $([\![v_s]\!], [\![v_d]\!])$, the servers can compute the partition index $[\![\lceil \frac{v_s}{k} \rceil]\!] * b + [\![\lceil \frac{v_d}{k} \rceil]\!]$ and obtain the partition containing l secret edges by accessing EORAM. This partition contains *all* edges $([\![v_s]\!], [\![v_d]\!])$ of the global graph if the edges exist.

For each given query, the servers only need to access one partition and scan it to obtain the result (see Section 5).

Batched update. To update the global graph, each data provider can locally update the private graph G_i and send the updated edge blocks to the computation servers. The servers then update the corresponding edge blocks in the global graph $[\![G.\mathsf{Block}]\!]$ and reconstruct the entire secure index, including the underlying ORAMs and the index map, using the same procedure as Step 3.

4.3 Parallelization and Vectorization

GORAM is a *parallel-friendly* data structure. All stages in its lifecycle can be accelerated through parallel processing.

In the local process stage, each data provider can independently split the private graph G_i into the 2d-partitioned format, during which the edge blocks can be processed in parallel. During the global integration, the primary bottleneck, i.e., the *odd_even_merge_sort* of b^2 edge blocks from N^+ secret graphs $\{[\![G_u.\mathsf{Block}]\!]\}_{i=1}^{i=N^+}$, can be performed in at most b^2 tasks in parallel. After obtaining the $b \times b \times l$ global graph partitions, we can split it into p *partition slices* for arbitrary $p \le l$ because each edge block is a list of l edges that can be processed in parallel. Specifically, we split the $b \times b$ edge blocks by edges into p $b \times b \times l^{(j)}$ partition slices, $l = \sum_{j=1}^{p} l^{(j)}$. Each slice contains all the edge blocks but fewer edges per block. We can then establish p secure indices for each partition slice concurrently. For each query, the p partition slices can be accessed and processed in parallel, and each slice can be processed as a single vector for better performance (Section 5). The query result can be obtained by merging the results of p partition slices.

4.4 Optimization on ShuffleMem Step

The ShuffleMem procedure dominates GORAM initialization, which shuffles n partitions according to a random permutation π and stores the secret permutation representation $[\![\vec{\pi}]\!]$. The original ShuffleMem (i.e.,*Waksman permutation network*[67]) incurs $O(n\log n)$

1	$S1(A, B)$	$S2(B, C)$	$S3(C, A)$
		0) Construct the shares of $L = [n]$.	
2	$L_A = Z_1 \oplus L$	$L_B = Z_2$	$L_C = Z_3$
3	$\leftarrow L_A$	$\leftarrow L_B$	$\leftarrow L_C$
		1) Prepare the correlated randomness.	
4	$Z_{12}, Z_{12}^L, \tilde{B}$	$Z_{12}, Z_{12}^L, \tilde{B}$	
5	π_{12} and π_{12}^{-1}	π_{12} and π_{12}^{-1}	
6		$Z_{23}, Z_{23}^L, \tilde{L}_C$	$Z_{23}, Z_{23}^L, \tilde{L}_C$
7		π_{23} and π_{23}^{-1}	π_{23} and π_{23}^{-1}
8	$Z_{31}, Z_{31}^L, \tilde{A}, \tilde{L}_A$		$Z_{31}, Z_{31}^L, \tilde{A}, \tilde{L}_A$
9	π_{31} and π_{31}^{-1}		π_{31} and π_{31}^{-1}
		2) Main protocol: computation and communications	
10	$X_1 = \pi_{12}(A \oplus B \oplus Z_{12})$	$Y_1 = \pi_{12}(C \oplus Z_{12})$	
11	$X_2 = \pi_{31}(X_1 \oplus Z_{31})$		
12		$LY_1 = \pi_{23}^{-1}(L_B \oplus Z_{23}^L)$	$LX_1 = \pi_{23}^{-1}(L_C \oplus L_A \oplus Z_{23}^L)$
13			$LX_2 = \pi_{31}^{-1}(LX_1 \oplus Z_{31}^L)$
14	$X_2 \leftrightarrow LY_1$		$Y_1 \leftrightarrow LX_2$
15			$Y_2 = \pi_{31}(Y_1 \oplus Z_{31})$
16		$X_3 = \pi_{23}(X_2 \oplus Z_{23})$	$Y_3 = \pi_{23}(Y_2 \oplus Z_{23})$
17		$\tilde{C}_1 = X_3 \oplus \tilde{B}$	$\tilde{C}_2 = Y_3 \oplus \tilde{A}$
18	$LY_2 = \pi_{31}^{-1}(LY_1 \oplus Z_{31}^L)$		
19	$LY_3 = \pi_{12}^{-1}(LY_2 \oplus Z_{12}^L)$	$LX_3 = \pi_{12}^{-1}(LX_2 \oplus Z_{12}^L)$	
20	$\tilde{L}_{B_1} = LY_3 \oplus \tilde{L}_A$	$\tilde{L}_{B_2} = LX_3 \oplus \tilde{L}_C$	
21	$L_{B_1} \leftrightarrow L_{B_2}$		$\tilde{C}_1 \leftrightarrow \tilde{C}_2$
22		$\tilde{C} = \tilde{C}_1 \oplus \tilde{C}_2$	$\tilde{C} = \tilde{C}_1 \oplus \tilde{C}_2$
23	$\tilde{L}_B = \tilde{L}_{B_1} \oplus \tilde{L}_{B_2}$	$\tilde{L}_B = \tilde{L}_{B_1} \oplus \tilde{L}_{B_2}$	
		3) Output	
24	$\tilde{A}, \tilde{B}, \tilde{L}_A, \tilde{L}_B$	$\tilde{B}, \tilde{C}, \tilde{L}_B, \tilde{L}_C$	$\tilde{C}, \tilde{A}, \tilde{L}_C, \tilde{L}_A$

Protocol 1: ShuffleMem **Build Protocol $\Pi_{\textbf{ShufMem}}$ - Lightgray operations are our extensions of Araki et al. [10] to compute $[\![\vec{\pi}]\!]$.**

communication and computation. GORAM optimizes this with a constant-round $O(n)$ protocol.

The ShuffleMem **procedure.** The computation servers begin with a secret shared array $[\![D]\!] = \{[\![D_0]\!], [\![D_1]\!], \ldots, [\![D_{n-1}]\!]\}$ of n partitions. At the end of the protocol, the computation servers output two secret shared arrays $[\![\widetilde{D}]\!]$ and $[\![\vec{\pi}]\!]$, where $\widetilde{D}$ is a permutation of D under some random *permutation* π and $[\![\vec{\pi}]\!]$ is the secret-shared *permutation representation* of π. The permutation π is a bijection mapping from D to itself that moves the ith partition D_i to place $\pi(i)$. The permutation result $\widetilde{D} = \{\widetilde{D}_0, \widetilde{D}_1, \ldots, \widetilde{D}_{n-1}\} = \pi(D)$ satisfies that $D_i = \widetilde{D}_{\pi(i)}, \forall i \in \{0, 1, \ldots n-1\}$. The permutation representation $\vec{\pi}$ explicitly records the location of each D_i in $\widetilde{D}$ in its ith element $\vec{\pi}_i$.

Key idea of constant-round construction. We design the constant-round ShuffleMem by extending Araki et al. [10], which computes $[\![\widetilde{D}]\!]$ in $O(n)$ complexity and $O(1)$ communication rounds using $(2, 3)$-secret sharing. The key idea is to compute $[\![\vec{\pi}]\!]$ simultaneously by leveraging the properties of permutations:

∗ Permutations are composable, i.e., $\pi_1 \circ \pi_2$ is also a permutation such that $(\pi_1 \circ \pi_2)(x) = \pi_1(\pi_2(x))$ given array x.

∗ Permutations are inversible, for each permutation π, there exists π^{-1} such that $(\pi^{-1} \circ \pi)(x) \equiv x$.

∗ The permutation representation $\vec{\pi} = \pi^{-1}(L)$, where $L = \{0, 1, \ldots, n-1\}$, n is the size of x.

Specifically, Araki et al. [10] implement the random shuffle by letting the computation servers collaboratively apply three random permutations π_{12}, π_{31} and π_{23}. They compute $[\![\widetilde{D}]\!] = \pi_{23} \circ \pi_{31} \circ \pi_{12}([\![D]\!]) = \pi([\![D]\!])$. The randomness of the final permutation π is guaranteed by the fact that each computation server only knows two out of the three random permutations; consequently, the overall permutation π remains random to any server. Using the same collaborative shuffle procedure, we can compute the secret permutation representation $[\![\vec{\pi}]\!]$ simultaneously by shuffling the ranging array $L = \{0, 1, \ldots, n-1\}$ using the inverse permutations i.e., $[\![\vec{\pi}]\!] = \pi^{-1}([\![L]\!]) = \pi_{12}^{-1} \circ \pi_{31}^{-1} \circ \pi_{23}^{-1}([\![L]\!])$.

ShuffleMem **construction.** Protocol 1 shows the ShuffleMem construction. Each pair of computation servers S_i and S_j share a common random seed $s_{i,j}$ beforehand. As inputs to this protocol, each computation server holds two out of the three shares A, B, C, where $D \equiv A \oplus B \oplus C$ (Line 1). Also, the computation servers construct the shares of the ranging array $L \equiv L_A \oplus L_B \oplus L_C$. Specifically, S_1, S_2 and S_3 at first construct an array of secret shares on zeros, i.e., $Z_1 \oplus Z_2 \oplus Z_3 \equiv \vec{0}$, $|\vec{0}| = n$, which requires no interactions using [50]. Each Z_i is uniformly random and is only known to S_i. S_1 locally computes $L_A = Z_1 \oplus L$ and each server sends its share to the previous server to obtain the secret shares of L (step 0, Lines 2-3). The first step is to set up the correlated randomness using the pairwise random seed. Variables with the same notation mean the same value, e.g., Z_{12} is the same random value shared by S_1 and S_2. Also, each pair of S_i and S_j generates a random permutation $\pi_{i,j}$ and the inverse permutation $\pi_{i,j}^{-1}$ (Lines 4-9).

The computation servers then compute the result shares in Lines 10-23, during which there are two invariants held: (1) All the computed $X_i \oplus Y_i$ is a permutation of D and (2) $LX_i \oplus LY_i$ is an inverse permutation of L. For example, $X_1 \oplus Y_1 = \pi_{12}(A \oplus B \oplus Z_{12}) \oplus \pi_{12}(C \oplus Z_{12}) = \pi_{12}(A \oplus B \oplus C) = \pi_{12}(D)$ (Line 10), $LY_1 \oplus LX_1 = \pi_{23}^{-1}(L_B \oplus Z_{23}^L) \oplus \pi_{23}^{-1}(L_C \oplus L_A \oplus Z_{23}^L) = \pi_{23}^{-1}(L)$ (Line 12). That is, during the main protocol, the servers sequentially compute $X_1 \oplus Y_1 = \pi_{12}(D)$, $X_2 \oplus Y_2 = (\pi_{31} \circ \pi_{12})(D)$ (Lines 11 and 15) and $X_3 \oplus Y_3 = (\pi_{23} \circ \pi_{31} \circ \pi_{12})(D)$ (Line 16), which constitutes the final shares of $\pi(D)$, $\pi = \pi_{23} \circ \pi_{31} \circ \pi_{12}$. The permutation representation $\vec{\pi} = \pi^{-1}(L) = (\pi_{12}^{-1} \circ \pi_{31}^{-1} \circ \pi_{23}^{-1})(L)$ is constructed similarly in the reverse order.

Correctness. From the two invariants, it is straightforward to see the correctness of ShuffleMem Protocol 1. Because the final shares satisfy that $\tilde{A} \oplus \tilde{B} \oplus \tilde{C} = \tilde{X_3} \oplus \tilde{Y_3} = \pi(D)$, and $\tilde{L_A} \oplus \tilde{L_B} \oplus \tilde{L_C} = \tilde{LX_3} \oplus \tilde{LY_3} = \pi^{-1}(L)$, the correctness is guaranteed.

Security. Protocol 1 satisfies Theorem 1. We provide a sketch proof here and the complete proof is shown in the technical report [31].

Theorem 1. *Protocol 1 securely implements the* ShuffleMem *procedure against any semi-honest adversary controlling at most one computation server.*

Proof. (sketch) We prove Theorem 1 with the *real-ideal* paradigm [21]. Let $\mathcal{A}$ denotes the real-world adversary and $\mathcal{S}$ denotes the simulator, which simulates the view of $\mathcal{A}$ in the ideal world. Protocol 1 is secure if for all $\mathcal{A}$, there exists a simulator $\mathcal{S}$, such that for all inputs and for all corrupted party $S_i, i \in [3]$,

$$\text{View}(\mathcal{A}) \equiv \text{View}(\mathcal{S}) \tag{1}$$

For each possible corrupted S_i, all the messages it receives are uniformly random in its view. This is achieved by masking all the messages S_i receives with at least one randomness that S_i does not know. Therefore, we can construct the simulator $\mathcal{S}$ by randomly sampling all the messages that $\mathcal{A}$ receives. The View($\mathcal{S}$) is uniformly random and therefore indistinguishable from View($\mathcal{A}$). □

5 QUERYING GRAPHS THROUGH GORAM

We provide five ego-centric query examples, covering all queries listed in LinkBench [11]. The other queries can be similarly achieved.

Algorithm 1: EdgeExist (MPC servers compute)

```
Inputs : Target edge ($[\![v_s]\!], [\![v_d]\!]$).
Output : $[\![\text{flag}]\!]$ indicating whether the target edge exist in global $G$.
1  Compute the secret partition ID $[\![i]\!] = [\![\lceil \frac{v_s}{k} \rceil * b + \lceil \frac{v_d}{k} \rceil]\!]$;
2  Fetch the target edge partition $[\![B]\!] \leftarrow$ EORAM.access($[\![i]\!]$), where
     $[\![B]\!]$ contains $l$ source_nodes and dest_nodes;
   // Vectorized edges comparisons.
3  Construct $[\![\vec{v_s}]\!]$ and $[\![\vec{v_d}]\!]$ by expanding $[\![v_s]\!]$ and $[\![v_d]\!]$ $l$ times;
4  Compute $[\![\text{mask}_s]\!] \leftarrow$ EQ($[\![\vec{v_s}]\!], [\![B]\!]$.source_nodes) ;
5  Compute $[\![\text{mask}_d]\!] \leftarrow$ EQ($[\![\vec{v_d}]\!], [\![B]\!]$.dest_nodes) ;
6  Compute $[\![\text{mask}]\!] \leftarrow$ AND($[\![\text{mask}_s]\!], [\![\text{mask}_d]\!]$) ;
   // Aggregating the result through OR.
7  while len($[\![\text{mask}]\!]$) > 1 do
8      Pad $[\![0]\!]$ to $[\![\text{mask}]\!]$ to be even ;
9      Split $[\![\text{mask}]\!]$ half-by-half to $[\![\text{mask}]\!]_l$ and $[\![\text{mask}]\!]_r$;
10     Aggregate $[\![\text{mask}]\!] \leftarrow$ OR($[\![\text{mask}]\!]_l, [\![\text{mask}]\!]_r$) ;
11 end
12 return $[\![\text{flag}]\!] = [\![\text{mask}]\!]$ to the client.
```

Algorithm 2: NeighborsCount (MPC servers compute)

```
Inputs : Target vertex $[\![v]\!]$.
Output : $[\![\text{num}]\!]^A$, the number of $v$'s outing neighbors.
1  Compute the secret partition ID $[\![i]\!] = [\![\lceil \frac{v}{k} \rceil]\!]$;
2  Fetch the target edge partition $[\![B]\!] \leftarrow$ VORAM.access($[\![i]\!]$), where
     $[\![B]\!]$ contains ($bl$) source_nodes and dest_nodes;
   // Filtering real neighbors using vectorization.
3  Construct $[\![\vec{v}]\!]$ by expanding $[\![v]\!]$ $bl$ times;
4  Compute $[\![\text{mask}]\!] \leftarrow$ EQ($[\![\vec{v}]\!], [\![B]\!]$.source_nodes) ;
5  Obtain the arith shares $[\![\text{mask}]\!]^A \leftarrow$ B2A($[\![\text{mask}]\!]$) ;
6  Compute $[\![\text{num}]\!]^A \leftarrow$ SUM($[\![\text{mask}]\!]^A$);
7  return $[\![\text{num}]\!]^A$ to the client.
```

Algorithm 3: NeighborsGet (MPC servers compute)

```
Inputs : Target vertex $[\![v]\!]$.
Output : $[\![\text{neighbors}]\!]$, the unique outing neighbor's IDs of $[\![v]\!]$.
1  Compute the secret partition ID $[\![i]\!] = [\![\lceil \frac{v}{k} \rceil]\!]$;
2  Fetch the target edge blocks $[\![B]\!] \leftarrow$ VORAM.access($[\![i]\!]$), where $[\![B]\!]$
     contains ($bl$) source_nodes and dest_nodes;
   // 1) Filtering real neighbors.
3  Construct $[\![\vec{v}]\!]$ by expanding $[\![v]\!]$ $bl$ times;
4  Compute $[\![\text{mask}]\!] \leftarrow$ EQ($[\![\vec{v_s}]\!], [\![B]\!]$.source_nodes) ;
5  Compute $[\![\text{candidate}]\!] \leftarrow$ MUL($[\![\text{mask}]\!], [\![B]\!]$.dest_nodes) ;
   // 2) De-duplicating neighbors.
6  $[\![\text{same\_mask}]\!] \leftarrow$ NEQ($[\![\text{candidate}]\!]_{[1:]}, [\![\text{candidate}]\!]_{[:-1]}$);
7  Compute $[\![\text{same\_mask}]\!]$.append($[\![1]\!]$);
8  Compute $[\![\text{neighbors}]\!] \leftarrow$ MUL($[\![\text{same\_mask}]\!], [\![\text{candidate}]\!]$);
9  Compute $[\![\text{neighbors}]\!] \leftarrow$ SHUFFLE($[\![\text{neighbors}]\!]$);
10 return $[\![\text{neighbors}]\!]$ to the client.
```

For each query, the client submits the secret query key to the servers. The servers access one secret-shared partition through GORAM, process it using the following protocols, and send the resulting shares to the client. During the process no information is leaked except the final results, which only the client can reconstruct.

5.1 Basic Queries

EdgeExist Algorithm 1 checks whether an edge (v_s, v_d) exists. We first access EORAM using secret index $[\![\lceil \frac{v_s}{k} \rceil]\!] * b + [\![\lceil \frac{v_d}{k} \rceil]\!]$, then compare all the edges in the partition using vectorization. The result is a secret $[\![\text{mask}]\!]$ indicating which edge is equivalent to the given edge. We obtain the result by aggregating $[\![\text{mask}]\!]$ using OR.

NeighborsCount Algorithm 2 counts the number of vertex v's outing neighbors. We refer to VORAM for the partition containing all its outing neighbors using secret index $[\![\lceil \frac{v}{k} \rceil]\!]$. We obtain the result by comparing all starting vertices to v and summing up the comparison result. To perform the summation with minimum communications, we transform $[\![\text{mask}]\!]$ to $[\![\text{mask}]\!]^A$ first.

NeighborsGet Algorithm 3 extracts all the 1-hop outing neighbors of v while hiding the number of edges between each neighbor and v. The first 3 lines access the target partition and compare all the starting vertices with v to construct $[\![\text{mask}]\!]$, indicating the edges started from v. Then, we multiply the $[\![\text{mask}]\!]$ and the destination vertices to obtain $[\![\text{candidate}]\!]$. Each element is $[\![0]\!]$ or $[\![u]\!]$ if u is an outing neighbor of v. The number of $[\![u]\!]$ implies the number of edges between u and v, we then de-duplicate $[\![\text{candidate}]\!]$ in lines 5-8. Because the construction stage sorts the edges by key $v||u$, all the same outing neighbors in $[\![\text{candidate}]\!]$ are located successively as a group. We apply a single shifted NEQ on $[\![\text{candidate}]\!]$ to compute $[\![\text{same\_mask}]\!]$, where only the last neighbor in each group is $[\![1]\!]$, while the rest are $[\![0]\!]$. By multiplying $[\![\text{same\_mask}]\!]$ and $[\![\text{candidate}]\!]$, the duplicate neighbors are masked as $[\![0]\!]$. Because the gap between two successive neighbors u_i, u_{i+1} still implies how many u_{i+1} exist, we apply SHUFFLE to permute this information.

5.2 Complex queries

We provide two complex queries by extending the above queries.

Cycle-identification. Identifying whether the transactions across multiple suspicious accounts form a cycle is an effective way for money laundering detection [46, 54], which can be achieved by submitting multiple EdgeExist queries. For example, given three vertices v_1, v_2 and v_3, by submitting EdgeExist queries on edges $(v_1, v_2), (v_2, v_3), (v_3, v_1)$ and their reverse edges, the client can detect whether a cycle exists among the three vertices.

Neighbors-filtering queries are one of the most common queries on graphs with attributes, i.e., each edge has attributes like creation timestamp and transaction amounts. We can implement these queries by extending the basic queries with filters. For instance, *association range queries* [11] that count the outing edges created after a provided timestamp can be implemented by extending NeighborsCount with an extra comparison to compute whether the creation timestamp is greater than the given timestamp before counting the result. Specifically, we compute $[\![\text{t\_mask}]\!] \leftarrow$ GT($[\![\text{timestamp field}]\!], [\![\text{given threshold}]\!]$) and update the neighbors mask in the 3rd line of Algorithm 2 to $[\![\text{mask}]\!]$ = AND($[\![\text{mask}]\!], [\![\text{t\_mask}]\!]$).

6 SECURITY AND COMPLEXITY ANALYSIS

6.1 Security Analysis

End-to-end security. We illustrate the security guarantees of GORAM by showing that no involved party (data providers, clients,

Table 1: Complexity Summarization (see Section 7.3 for empirical comparison and technical report [31] for derivation.)

Data Structures		Initialization		Partition Access		Partition Processing for Basic Queries					
						`EdgeExist`		`NeighborsCount`		`NeighborsGet`	
		Comp	Round	Comp	Round	Comp	Round	Comp	Round	Comp	Round
Mat	adj-VORAM	$O(N\|V\|^2)$	$3 + 2\log_P(\frac{\|V\|}{T})$	$O(PT\log_P(\frac{\|V\|}{T}))$	$O(\log_P(\frac{\|V\|}{T}))$	$O(1)$	$O(1)$	$O(\|V\|)$	$O(1)$	$O(\|V\|)$	$O(1)$
	adj-EORAM		$3 + 2\log_P(\frac{\|V\|^2}{T})$	$O(PT\log_P(\frac{\|V\|^2}{T}))$	$O(\log_P(\frac{\|V\|^2}{T}))$						
List		$O(\|E\|\log(\|E\|)\log(N))$	$\log(\|E\|)\log(N)$	$O(1)$	NA	$O(\|E\|)$	$O(\log(\|E\|))$	$O(\|E\|)$	$O(1)$	$O(\|E\|)$	$O(1)$
GORAM	VORAM	$O(b^2 l\log(l)\log(N^+))$	$3 + \log(l)\log(N^+) + 2\log_P(\frac{b}{T})$	$O(PT\log_P(\frac{b}{T}))$	$O(\log_P(\frac{b}{T}))$	$O(l)$	$O(\log(l))$	$O(bl)$	$O(1)$	$O(bl)$	$O(1)$
	EORAM		$3 + \log(l)\log(N^+) + 2\log_P(\frac{b^2}{T})$	$O(PT\log_P(\frac{b^2}{T}))$	$O(\log_P(\frac{b^2}{T}))$						

P and T denote the pack and stash size of ORAM. b, l are the configuration parameters of 2d-partition, where $b = \frac{|V|}{k} = \frac{|E|}{|V|} = D$ is the graph density. The Round complexities with the $O(\cdot)$ notation is in the unit of secure operations like EQ, and the Partition Access complexities are the averaged complexity of successive T queries.

Table 2: Parameters of Real-world Graphs

Graph	$\|V\|$	$\|E\|$	$\|E\| / \|V\|$	k	b	l	bl	$b^2l / \|E\|$	$\|V\|^2/b^2l$
Slashdot	82.2K	948.5K	11.5	10.2K	8	25.4K	203.5K	1.7	4.1K
DBLP	524.2K	706.3K	1.3	65.5K	8	15.7K	126.0K	1.4	272.6K
Twitter	41.7M	1.5G	35.3	650.8K	64	1.1M	71.8M	3.1	377.1K

and computation servers) can learn anything about the private graphs and the query keys.

(1) Each data provider learns nothing because they only submit secret-shared graph partitions to the servers.

(2) Each client learns nothing except the final results because they only submit secret-shared keys to the servers and receive results obtained from the global graph.

(3) During GORAM initialization and query process, the computation servers only execute MPC protocols on N^+ indistinguishable $b \times b \times \bar{l}$ secret-shared 2d-partitions and secret-shared query keys and learn nothing except the above public parameters. These protocols are executed in the standard *modular composition* manner [21] obliviously, i.e., the execution paths are indistinguishable for *any* inputs with the same size, which ensures that the overall computation performed by the servers inherits the same security guarantees of the underlying protocols. These include the ABY3 protocols [50], the underlying Square-root ORAM protocol, and the proposed ShuffleMem protocol whose security is proven in Theorem 1. Consequently, the only potential source of private information for the servers is the sizes of the input matrices, i.e., b, $\bar{l}$ and the total matrices number N^+. Among these, $\bar{l}$ is a public parameter, and $b = \lceil \frac{|V|}{k} \rceil$ is derived from two public parameters, $|V|$ and k. The number of the matrices, N^+, only reveals the partition size of the global graph and cannot be traced back to any specific data provider because of the anonymous authentication.

6.2 Complexities Analysis

Complexity summarization. Table 1 summarizes the complexities of initialization, partition access, and basic query processing using GORAM and the strawman solutions. GORAM's initialization is bottlenecked by the $O(b^2 l\log(l)\log(N^+))$ merge sort on N^+ partitioned graphs (Section 4.2). Partition access complexity is $O(PT\log_P(\frac{b}{T}))$ or $O(PT\log_P(\frac{b^2}{T}))$ for VORAM and EORAM, respectively (Section 4). Partition processing is linear to the partition size (Section 5.1). The discussion of complexities on *Mat* and *List* is left to Section 7.1, and Section 7.3 provides the comparisons across the three data structures. Technical report [31] provides the derivation.

Partition size l analysis. Table 1 shows that the performance of GORAM depends on $l = N^+\bar{l}$ (Section 4.2), which is related to the graph topology and the intrinsic randomness of GORAM. We analyze the distributions of l across five distributed graphs using the *Monte Carlo* method. Our findings in Section 7.3 indicate that, across all distributions, l lies in a small range and is notably smaller than the total edge count, i.e., $l \ll |E|$, indicating that GORAM can effectively process queries by accessing only one partition. The worst-case $l = |E|$ is *unlikely* in practice because it requires: (1) all the edges in the graph only involve at most $2k$ vertices, k for starting vertices and k for the endings, and (2) the graph forms a bipartite graph starting from and ending in two chunks after the *random* permutation in the initialization (Section 4). Also, $l < k^2$ for most graphs, meaning that the size of GORAM is smaller than the full $|V|^2$ adjacency matrix. The unlikely case $l \geq k^2$ happens only when two *randomly* assigned vertex chunks contain more edges than a complete bipartite graph.

7 EVALUATION

7.1 Evaluation Setup

Setup. We implement the GORAM prototype based on ABY3 [50]. All values are 64-bit secret shares. We use three computation servers, each equipped with 16 × 2.0 GHz Intel CPU cores, 512 GB memory and 10 Gbps full duplex Ethernet with an average round-trip-time (RTT) of 0.12 ms. We also use a WAN network with 900 Mbps bandwidth and 20 ms RTT to evaluate the query performance as a sensitivity analysis to network settings. Note that we use the entire memory just to support a larger-scale *Mat* baseline.

Workload. We use graphs of two types over all five queries in Section 5. First, we use three real-world graphs, i.e., Slashdot [6], DBLP [64], and Twitter [18], with the number of edges ranging from less than 1 million to more than 1 *billion.* Table 2 lists the key parameters of these graphs. For micro-benchmarks, we use thirty synthetic graphs of five edge distributions with $|V| \in [1K, 32K]$ from `igraph` [33]. Table 3 summarizes their key parameters.

Strawman solutions for comparison. Since there are no prior systems that provide the same functionality as GORAM, we implement the strawman solutions in Section 3.3, upon which we implement the same queries. We also add vectorization and parallelism, wherever applicable, to minimize the performance difference from implementation quality. In fact, much of the low-level code is shared with GORAM, including the ORAM and all the protocols.

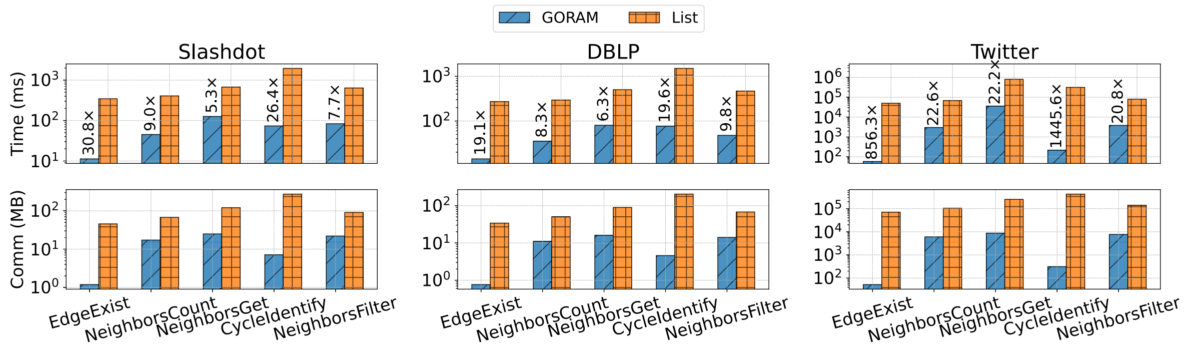

Figure 4: Queries on Real-world Graphs * y-axes are in log-scale. Queries on Twitter are in 16 threads and the others are single-threaded.

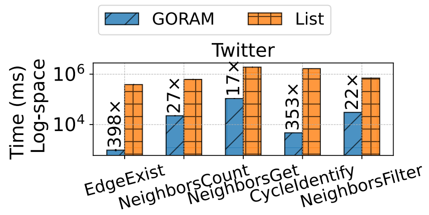

Figure 5: Queries on WAN Network

For *Mat*, we first implement the three basic queries: (1) `EdgeExist` on ($[\![v_i]\!]$, $[\![v_j]\!]$): we extract the element (i, j) from adj-EORAM (Section 3.3) and compare it to 0; (2) `NeighborsCount`: we access adj-VORAM and sum up the number of edges; and (3) `NeighborsGet`: we access adj-VORAM, and compare the elements with 0 through GT to hide the exact number of edges, and then return the result to the client. `CycleIdentify` is then implemented by composing six `EdgeExist`, the same as GORAM. We ignore `NeighborsFilter` because *Mat* can not support multi-graphs, as Section 3.3 analyzed. For *List*, we implement all queries by scanning the whole edge list, reusing GORAM's implementation on each partition.

Execution time measurements. All the reported execution times are the wall-clock time measured on the computation servers from the start to the end of the initialization or query processing. We report the average from 5 runs. For GORAM and *Mat* that require ORAM accesses, the query processing time is the averaged time of successive T queries , where T is the stash size, and we use $T = \sqrt{\#(\text{items in ORAM})}$, the default setting in [67].

7.2 Performance on Real-world Graphs

We first provide an overview of GORAM performance using three real-world graphs with 700K to 1.4 billion edges. The graph parameters are shown in Table 2, including the number of vertices $|V|$, edges $|E|$, partitions b, and the partition size l and bl for EORAM and VORAM. All the parameters are default values following Section 4.

Overall Query Performance. The first row of Figure 4 shows the query execution time using vectorization. We use single-threading for smaller graphs and 16 threads for large Twitter.

For Slashdot [6] and DBLP [64], GORAM completes all the queries within 126.4 ms. To our knowledge, this is the only system that supports sub-second ego-centric queries on these graphs with strong privacy guarantees. In comparison, *Mat* gets out of memory even with the entire 512 GB of memory per server, and *List* is 15.9× and 4.2× slower on average on these two graphs.

For Twitter, GORAM takes 58.1 ms to 35.7 seconds to complete all queries, achieving a remarkable average speedup of 473.5× over *List*. Across all queries, GORAM achieves more significant speedups for `EdgeExist` and `CycleIdentify`, with 856.3× and 1445.6× speedups, respectively. This is because these two queries use EORAM, with which we only need to access *one* of the $b^2 = 4096$ edge blocks with 1.1M edges, which is less than 0.8‰ of the total edges. Instead, the other three use the VORAM that accesses one *row* of the $b = 64$ edge blocks, which contain 71.2M edges, accounting 4.9% for the total edges. Figure 5 shows the performance of GORAM and *List* on WAN. We can see that even on a limited network bandwidth with 20 ms latency, GORAM completes all the queries in an average of 33.4 seconds. On the contrary, *List* costs 1057.2 seconds in average.

Fast query execution comes from processing less data. To verify that the execution time reduction indeed comes from reduced computation through partitioning, we also measure the total bytes transferred in each query on each graph. The second row in Figure 4 shows the results, which are consistent with execution time - we observe an average communication reduction of 78.4% compared to *List*. Also, the maximum reduction of 99.9% is observed in `EdgeExist` and `CycleIdentify`, for the same reason above.

Fast query execution also comes from parallel execution. Query performance on large graphs also benefits from parallelization, as shown in Figure 6. On the large Twitter, we observe that using 16 threads, we can achieve an average speedup of 6.3× across the five queries over a single thread. We observe that except for `NeighborsGet`, we can achieve almost linear scalability using up to 16 threads. The slight derivation from linear scalability is because 8 threads already accelerate the computation to sub-second levels; adding more threads provides minimal speedup while increasing the aggregation overhead. `NeighborsGet` does not parallel well because the SHUFFLE procedure (line 8 in Algo. 3) needs to process the entire partition to permute the result, which is inherently sequential.

Initialization performance. Unlike *List*, both VORAM and EORAM require a non-trivial initialization step to construct the secure indices (Section 4.2). While the two smaller graphs only take a few seconds, it takes dozens of minutes on the large Twitter using a naive sequential algorithm. However, we can parallelize the initialization using multiple threads, as shown in Section 4.3. In this

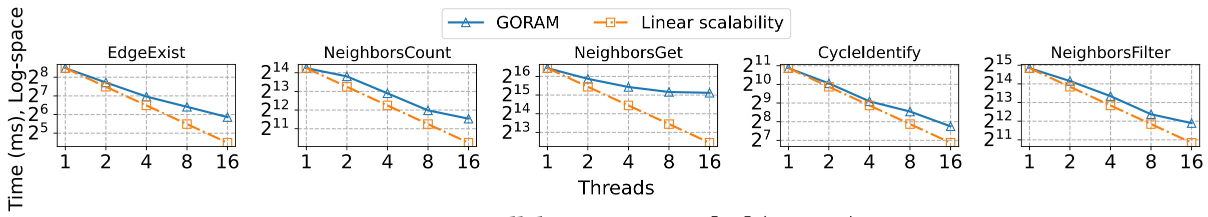

Figure 6: Parallelism on Twitter [18] (Queries)

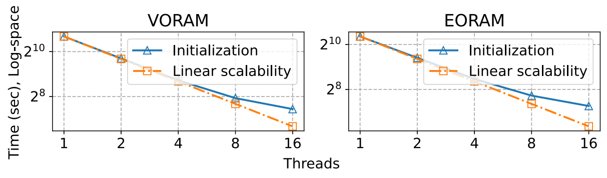

Figure 7: Parallelism on Twitter [18] (Initialization)

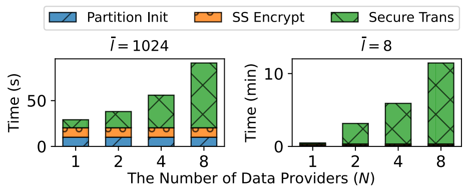

Figure 8: Deployment Cost

Table 3: Synthetic Graphs

Graph Types	Generation Methods	Average Degree
`k_regular`	`K_Regular` [1]	7.5
`bipartite`	`Random_Bipartite` [2]	134.4
`random`	`Erdos_Renyi` [3]	268.8
`powerlaw`	`Barabasi` [4]	523.7
`geometric`	`GRG` [5]	1198.5

Table 4: Parameters of Synthetic Graphs ($|V|$=32K)

Graph Type	$\|E\|$	k	b	l	bl	$b^2l/\|E\|$
`k_regular`	0.2M	4096	8	3960	31.8K	1.03
`bipartite`	13.4M	64	512	88	44.5K	1.72
`random`	26.8M	32	1024	56	57.3K	2.19
`powerlaw`	52.3M	16	2048	48	98.3K	3.85
`geometric`	105.0M	8	4096	32	131.1K	5.11

way, we can construct both VORAM and EORAM for the billion-edge-scale graph within 2.9 minutes using 16 threads. Figure 7 shows the experiments using different numbers of threads. We observe an average speedup with 16 threads is 9.4× over a single thread, and at this setting, we have saturated the 10 Gbps network bandwidth.
Deployment cost. Figure 8 shows the deployment cost of GORAM using Twitter graph, varying the number of data providers from 1 to 8 and using two configuration parameters, $\bar{l} = 8$ and 1024 (Section 4.2). We evenly split the edge list of Twitter into 8 parts, each assigned to a data provider running with 16 CPU cores. Figure 8 breaks down the costs into: (1) Partition initialization, which transforms the local edge lists into 2d-partitions; (2) Secret sharing encryption, which encrypts the 2d-partitions into secret shares; and (3) Secure transmission, which transmits the secret shares through multiple anonymous channels to the computation servers (Step1 in Section 4.2). The first two are measured on the data provider side, and the last is measured on the computation server, from secure channel setup with all data providers to receipt of all secret shares. We can see that the secure transmission time dominates, and this cost decreases as $\bar{l}$ increases. Increasing $\bar{l}$ from 8 to 1024 reduces the overall deployment time with 8 data providers from 11.4 minutes to 91.0 seconds. This is because there are $N^+ = \sum_{i=1}^{N} \lceil \frac{l}{\bar{l}} \rceil$ channels to be established, and smaller $\bar{l}$ leads to more channels. Larger $\bar{l}$ reduces deployment cost but increases the query latency because data providers need to pad more edges to be multiples of $\bar{l}$. Developers can tune $\bar{l}$ to prioritize the deployment cost or query speed.

7.3 Micro-benchmarks

We benchmark GORAM on 30 synthetic graphs using three basic queries against strawman solutions. All tests are single-threaded to highlight the benefits of GORAM structure.

Adaption to various distributed graphs. Each distribution presents a different density, i.e. vertex degree $D = \frac{|E|}{|V|}$. Table 3 shows the average degrees. `k_regular` is the sparsest, and `geometric` is the densest. We present the default configuration parameters k, vertex chunk numbers b, the partition sizes for EORAM (l) and VORAM (bl) in Table 4, all of them are default parameters of GORAM (see Section 4). The ratio $\frac{b^2 l}{|E|}$ further shows the amplification factor of GORAM with padded edges compared to the original edge list. For sparse `k_regular`, GORAM uses k up to 4096, while for `geometric`, k gets as small as 8. Recall that a key property when we partition is that all outgoing edges of a vertex are within a single row of the 2d-partition. Thus, when the graph is sparse, we can partition it into fewer chunks, i.e., smaller $b = \frac{|V|}{k}$, enjoying the benefits of quicker partition access time. However, on a dense graph, we partition it into more chunks to make each partition smaller, thereby reducing the partition scanning time per query.
Partition size l distributions across diverse distributed graphs are shown in Figure 9, derived through the *Monte Carlo* method. We generate 10K graphs for each distribution, each with 32K vertices. For each graph, we randomly permute the vertices using different random seeds and construct the 2d-partitioned structure to obtain the partition size l as Section 4 illustrates without padding. As Figure 9 shows, the range of l is relatively narrow for all distributions. E.g., l varies within a range of 188 in `k_regular` and only varies within a range of 11 in `random` graph. Notably, the partition size l is significantly smaller than $|E|$ for all five graph types (see Table 4), suggesting that GORAM often significantly outperforms *List*.
Query execution time. Each row of Figure 10 presents the execution time of a query across different graph densities from the sparsest to the densest, using both GORAM and the two strawman solutions. Overall, we observe that GORAM delivers highly efficient

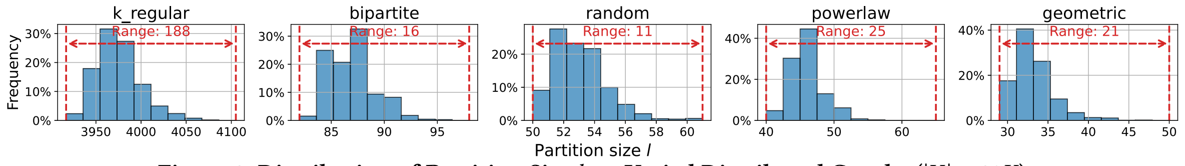

Figure 9: Distribution of Partition Size l on Varied Distributed Graphs ($|V| = 32K$).

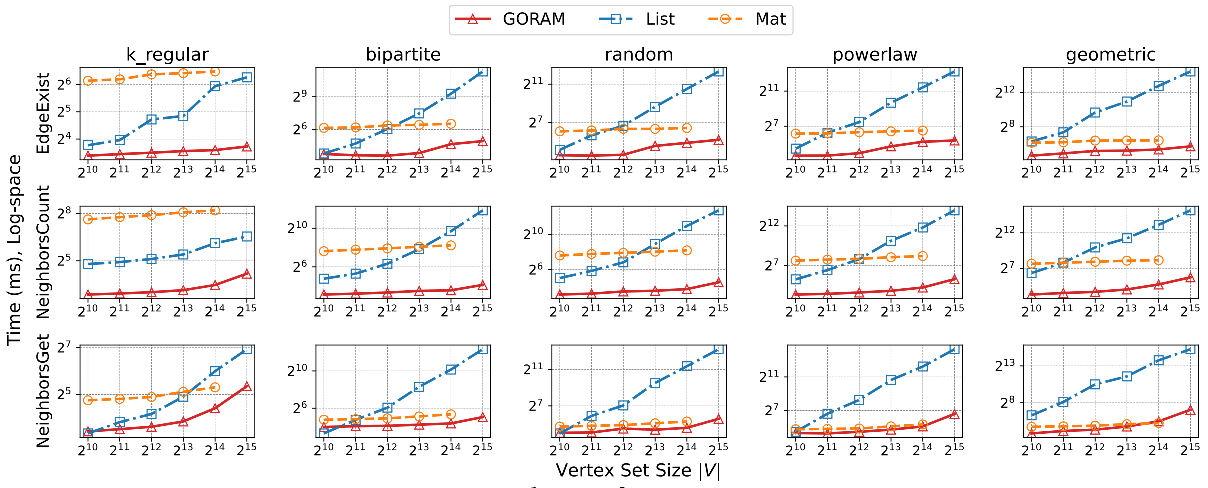

Figure 10: Online Performance Overview

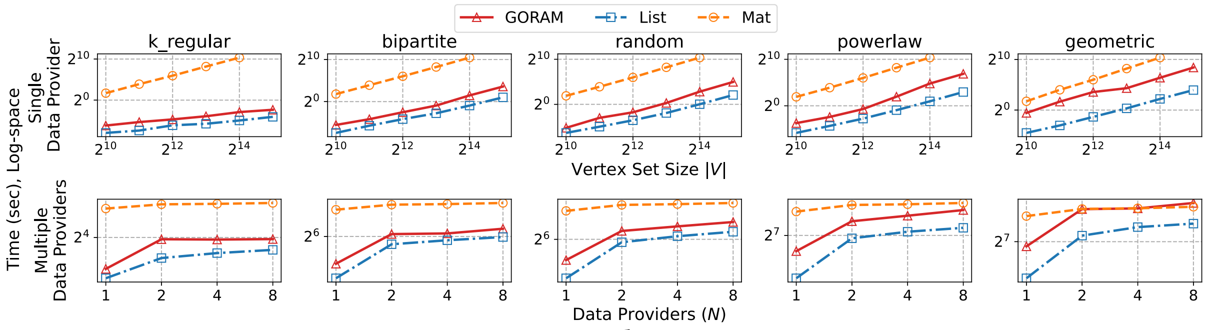

Figure 11: Initialization Cost

query responses across all 90 test cases (6 sizes × 5 graph distributions × 3 queries), offering an average query completion time of 22.0 ms. We provide a detailed performance analysis below.

Performance vs. graph density and size. *Mat* is the least sensitive to graph density (e.g., all around 2^6 ms across the first row), because its performance only depends on $|V|$ (see Table 1). *List*, on the contrary, is very sensitive to density, given its $O(|E|)$ query time. GORAM works well across different densities, exhibiting sublinear execution time on both $|V|$ and the graph density. The trend in execution time matches the theoretical complexities in Table 1.

For sparse graphs (first two columns in Figure 10), *Mat* performs the worst on almost all $|V|$ settings, as expected, because it spends too much resource processing empty cells. *List* performs as well as, or even better than GORAM on very small graphs (i.e., 1024 vertices) for `NeighborsGet`. This is because `NeighborsGet` requires multiple secure comparison and multiplication operations, and the communication round latency becomes the bottleneck for small graphs. *List*, in this case, saves the communication rounds required for partition access, providing advantages compared to GORAM. However, the performance gets worse fast as $|V|$ increases, because the $O(|E|)$ complexity of *List* quickly dominates the performance.

For dense graphs (last two columns in Figure 10), *Mat* gets closer performance with GORAM as far as it supports the scale, but *List* performs poorly except for the smallest cases. In fact, in the largest version of the densest `geometric` graph, *List* can be as much as 703.6× slower because of its $O(|E|)$ complexity.

Performance with queries. *Mat* is always slower than GORAM on `NeighborsCount`, because it takes $O(\log(|V|))$ time to access adj-VORAM while GORAM only takes $O(\log(b))$, $b = \frac{|V|}{k}$. For `EdgeExist`, although GORAM requires less ORAM access time, the advantage is less significant because of the $O(\log(l))$ rounds of OR for aggregation (see Algo 1). However, *Mat* only needs a single round of comparison. Therefore, *Mat* becomes closer to GORAM for denser

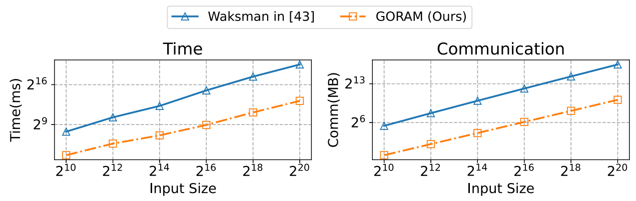

Figure 12: ShuffleMem **Construction** (* the y-axes are in log-scale.)

graphs with larger l. For NeighborsGet, GORAM and *Mat* perform similarly except for the sparsest k_regular. This is because GORAM requires multiple secure operations over vertex partition with bl edges while *Mat* only needs a single comparison on $|V|$ matrix cells. For denser graphs, this advantage offsets the higher overhead of ORAM access, i.e., $O(\log(|V|))$ for *Mat* and $O(\log(b))$ for GORAM.

Initialization performance. The first row in Figure 11 shows the initialization cost when there is only one data provider, which is the wall-clock time from data loading to secure indices construction (*List* does not require establishing indices). The initialization cost is linear to the graph sizes, i.e., $|V|^2$ or $|E|$. Given the relationship $|E| < b^2 l < |V|^2$ among *List*, GORAM, and *Mat*, the initialization costs follow the order: *Mat* > GORAM > *List*. *Mat* has the highest, constant cost on both sparse and dense graphs because of its $O(|V|^2)$ complexity. *Mat* runs out of memory building adj-EORAM on all graphs with 32K vertices, even with 512 GB memory.

Initialization performance with multiple data providers. The second row in Figure 11 presents the cost with multiple data providers (1 to 8). We simulate the distributed graphs by randomly assigning each edge of the synthetic graph with 16K vertices to each data provider, which is the largest scale *Mat* supports. For $N > 1$ data providers, both GORAM and *List* need to perform the merge sort on N^+ or N ordered private graphs (see Section 4.2). Merging adds the overhead, but the effect is limited because extra workload is logarithmic, i.e., $\log(N)$ or $\log(N^+)$, $N^+ \approx \frac{l}{l}$.

ShuffleMem construction comparison. Figure 12 compares the cost of ShuffleMem construction of an array of n secret-shared integers using *Waksman* permutation network, as adopted in [67], and our optimized constant-round ShuffMem protocol introduced in Section 4.4. The ShuffleMem construction is the main bottleneck of building ORAM. We can see that GORAM significantly accelerates both computation and communication, achieving 17.4× to 83.5× speedups and 97.5% to 98.8% communication savings as input sizes increase. This is because our method reduces the original $O(n\log(n))$ computation and communication to $O(n)$, thereby showing better performance as input sizes increase. Furthermore, unlike Waksman network, which necessitates an expensive switch operation, amounting to approximately $\approx 6n$ communications per layer in the $2\log(n)$ depth network, GORAM only requires shares transmission and XOR operations that do not need communications.

8 RELATED WORK

We discuss the prior arts related to GORAM.

Secure federated databases focus on conducting *public* SQL queries protecting the individual tuples. Examples include [13–15, 34, 41, 59]. For each query, these databases analyze the statements and run secure protocols on the required data to obtain the result. Based on the above progress, Aljuaid et al. [8, 9] proposes to process federated graph queries by directly translating the graph queries into SQL through [36]. Compared with these *public* query systems, we focus on ego-centric queries with private query keys.

Secure graph processing with theoretical guarantees. Beyond the *Mat* and *List* introduced in Section 3.3, there are proposals leveraging *Structured Encryption (SE)* [25] to query graphs securely. They focus on encrypting the graph in a way that can be privately queried [30, 40, 49, 62]. However, they require a shared key between the data provider and the client and, consequently, cannot be directly extended to allow multiple data providers and third-party clients. Other studies [43, 56] focus on private analytics over a set of devices organized as a graph, a topic orthogonal to our work.

Graph processing under other security settings. Numerous proposals focus on graph queries guaranteeing *differential privacy (DP)* like [35, 37, 43, 47, 48, 55, 60, 61]. They focus on protecting inputs from the results. In addition to the unavoidably inaccurate results, they pay less attention to the information leakage during the computation. E.g., FEAT [43] leaks noisy vertex degrees, which may be close approximations to control errors. GORAM, however, focuses on protecting *all* the information during the computation except the result, which is orthogonal to DP's goal. TEE-based approaches [22–24, 63] are vulnerable to side-channel attacks [44, 51].

DORAM implementations. FLORAM [29] and DuORAM [58] focus on high-latency and low-bandwidth settings. They trade a linear computation complexity for reduced communications. 3PC-DORAM [20], GigaORAM [29], and Square-root ORAM [67] struggle for sub-linear complexity. GigaORAM and 3PC-DORAM rely on the *Shared-In Shared-Out Pseudo Random Functions (SISO-PRF)* based on LowMC [7], which has known cryptanalysis [42]. GORAM builds its indices on Square-root ORAM, ensuring sublinear complexity and robust security guarantee.

9 CONCLUSION AND FUTURE WORK

We propose GORAM, the first step towards achieving efficient private ego-centric queries on federated graphs. GORAM introduces a methodology for reducing the to-be-processed data sizes in secure computations, relying on query-specific data partitioning and secure indices. We hope this method can be generalized to other applications beyond ego-centric queries. Extensive evaluations validate that GORAM achieves practical performance on real-world graphs, even with 1.4 billion edges. For future work, we aim to expand GORAM's capabilities for more advanced applications, including complex graph queries like path filtering and pattern matching, while also optimizing its performance and scalability.

ACKNOWLEDGMENTS

We thank Professor Wenguang Chen for his help during the design of GORAM. We also thank the anonymous reviewers for their valuable feedback. This work is supported in part by the National Key R&D Program of China 2023YFC3304802, National Natural Science Foundation of China (NSFC) Grant U2268202 and 62176135, and Ant Group Research Intern Program.

REFERENCES

[1] 2006. https://igraph.org/python/api/0.9.11/igraph._igraph.GraphBase.html#K_Regular

[2] 2006. https://igraph.org/python/api/0.9.11/igraph.Graph.html#Random_Bipartite

[3] 2006. https://igraph.org/python/api/0.9.11/igraph._igraph.GraphBase.html#Erdos_Renyi

[4] 2006. https://igraph.org/python/api/0.9.11/igraph._igraph.GraphBase.html#Barabasi

[5] 2006. https://igraph.org/python/api/0.9.11/igraph._igraph.GraphBase.html#_GRG

[6] 2009. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters.. In *Internet Mathematics*.

[7] Martin R Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. 2015. Ciphers for MPC and FHE. In *Advances in Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCYRPT)*.

[8] Nouf Aljuaid, Alexei Lisitsa, and Sven Schewe. 2023. Secure Joint Querying Over Federated Graph Databases Utilizing SMPC Protocols.. In *ICISSP*.

[9] Nouf Aljuaid, Alexei Lisitsa, and Sven Schewe. 2024. Efficient and Secure Multi-party Querying over Federated Graph Databases. In *International Conference on Data Science, Technology and Applications (DATA)*.

[10] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. 2021. Secure Graph Analysis at Scale. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[11] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: a Database Benchmark based on the Facebook Social Graph. In *Proceedings of the International Conference on Management of Data (SIGMOD)*.

[12] Gilad Asharov, Koki Hamada, Ryo Kikuchi, Ariel Nof, Benny Pinkas, and Junichi Tomida. 2023. Secure Statistical Analysis on Multiple Datasets: Join and Group-By. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[13] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel N Kho, and Jennie Rogers. 2017. SMCQL: Secure Query Processing for Private Data Networks.. In *Proceedings of the VLDB Endowment*.

[14] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. 2018. Shrinkwrap: efficient SQL query processing in differentially private data federations. In *Proceedings of the VLDB Endowment*.

[15] Johes Bater, Yongjoo Park, Xi He, Xiao Wang, and Jennie Rogers. 2020. SAQE: Practical Privacy-preserving Approximate Query Processing for Data Federations. In *Proceedings of the VLDB Endowment*.

[16] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. 2013. Data-oblivious Graph Algorithms for Secure Computation and Outsourcing. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIA-CCS)*.

[17] Dan Bogdanov, Sven Laur, and Jan Willemson. 2008. Sharemind: A Framework for Fast Privacy-preserving Computations. In *European Symposium on Research in Computer Security (ESORICS)*. Springer.

[18] Paolo Boldi and Sebastiano Vigna. 2004. The Webgraph Framework I: Compression Techniques. In *Proceedings of the International Conference on World Wide Web (WWW)*.

[19] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. 2013. TAO:Facebook's Distributed Data Store for The Social Graph. In *USENIX Annual Technical Conference (ATC)*.

[20] Paul Bunn, Jonathan Katz, Eyal Kushilevitz, and Rafail Ostrovsky. 2020. Efficient 3-party Distributed ORAM. In *Security and Cryptography for Networks (SCN)*.

[21] Ran Canetti. 2000. Security and Composition of Multiparty Cryptographic Protocols. In *Journal of CRYPTOLOGY*.

[22] Javad Ghareh Chamani, Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. 2024. GraphOS: Towards Oblivious Graph Processing. In *Proceedings of the VLDB Endowment*.

[23] Zhao Chang, Dong Xie, Sheng Wang, and Feifei Li. 2022. Towards Practical Oblivious Join. In *Proceedings of the International Conference on Management of Data (SIGMOD)*.

[24] Zhao Chang, Lei Zou, and Feifei Li. 2016. Privacy Preserving Subgraph Matching on Large Graphs in Cloud. In *Proceedings of the International Conference on Management of Data (SIGMOD)*.

[25] Melissa Chase and Seny Kamara. 2010. Structured Encryption and Controlled Disclosure. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*.

[26] Fan Chung. 2010. Graph Theory in the Information Age. In *Notices of the AMS*.

[27] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY-A framework for Efficient Mixed-protocol Secure Two-party Computation.. In *The Network and Distributed System Security Symposium (NDSS)*.

[28] Jack Doerner and Abhi Shelat. 2017. Scaling ORAM for Secure Computation. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[29] Brett Falk, Rafail Ostrovsky, Matan Shtepel, and Jacob Zhang. 2023. GigaDORAM: breaking the billion address barrier. In *Proceedings of the USENIX Conference on Security Symposium (USENIX Security)*.

[30] Francesca Falzon, Esha Ghosh, Kenneth G Paterson, and Roberto Tamassia. 2024. PathGES: An Efficient and Secure Graph Encryption Scheme for Shortest Path Queries. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[31] Xiaoyu Fan, Kun Chen, Jiping Yu, Xiaowei Zhu, Yunyi Chen, Huanchen Zhang, and Wei Xu. 2024. GORAM: Graph-oriented ORAM for Efficient Ego-centric Queries on Federated Graphs (Technical Report). *arXiv preprint arXiv:2410.02234* (2024). https://arxiv.org/pdf/2410.02234

[32] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. In *Journal of the ACM (JACM)*.

[33] Tamás Nepusz Gábor Csárdi. 2006. The igraph Software Package for Complex Network Research. In *InterJournal Complex Systems*.

[34] Feng Han, Lan Zhang, Hanwen Feng, Weiran Liu, and Xiangyang Li. 2022. Scape: Scalable collaborative analytics system on private database with malicious security. In *IEEE International Conference on Data Engineering (ICDE)*.

[35] Jacob Imola, Takao Murakami, and Kamalika Chaudhuri. 2021. Locally Differentially Private Analysis of Graph Statistics. In *Proceedings of the USENIX Conference on Security Symposium (USENIX Security)*.

[36] Alekh Jindal, Praynaa Rawlani, Eugene Wu, Samuel Madden, Amol Deshpande, and Mike Stonebraker. 2014. Vertexica: Your Relational Friend for Graph Analytics!. In *Proceedings of the VLDB Endowment*.

[37] Vishesh Karwa, Sofya Raskhodnikova, Adam Smith, and Grigory Yaroslavtsev. 2011. Private Analysis of Graph Structure. In *Proceedings of the VLDB Endowment*.

[38] Donald E Knuth. 1973. The Art of Computer Programming, VOL. 3: Searching and Sorting (The Odd Even Mergesort Network Section). In *Reading MA: Addison-Wisley*.

[39] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. 2024. Graphiti: Secure Graph Computation Made More Scalable. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[40] 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 *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIA-CCS)*.

[41] John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia. 2023. SECRECY: Secure Collaborative Analytics in Untrusted Clouds. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[42] Fukang Liu, Takanori Isobe, and Willi Meier. 2021. Cryptanalysis of Full LowMC and LowMC-M with Algebraic Techniques. In *Advances in Annual International Cryptology Conference (CRYPTO)*.

[43] Shang Liu, Yang Cao, Takao Murakami, Weiran Liu, Seng Pei Liew, Tsubasa Takahashi, Jinfei Liu, and Masatoshi Yoshikawa. 2023. Federated Graph Analytics with Differential Privacy. In *International Workshop on Federated Learning for Distributed Data Mining*.

[44] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. 2021. A Survey of Microarchitectural Side-channel Vulnerabilities, Attacks, and Defenses in Cryptography. In *ACM Computing Surveys (CSUR)*.

[45] Steve Lu and Rafail Ostrovsky. 2013. Distributed Oblivious RAM for Secure Two-party Computation. In *Theory of Cryptography Conference (TCC)*.

[46] Nav Mathur. 2021. Graph Technology for Financial Services. Neo4j. https://go.neo4j.com/rs/710-RRC-335/images/Neo4j-in-Financial%20Services-white-paper.pdf (White Paper).

[47] Sahar Mazloom and S Dov Gordon. 2018. Secure Computation with Differentially Private Access Patterns. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[48] Sahar Mazloom, Phi Hung Le, Samuel Ranellucci, and S Dov Gordon. 2020. Secure Parallel Computation on National Scale Volumes of Data. In *Proceedings of the USENIX Conference on Security Symposium (USENIX Security)*.

[49] Xianrui Meng, Seny Kamara, Kobbi Nissim, and George Kollios. 2015. GRECS: Graph Encryption for Approximate Shortest Distance Queries. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[50] Payman Mohassel and Peter Rindal. 2018. ABY3: A Mixed Protocol Framework for Machine Learning. In *ACM SIGSAC conference on computer and communications security (CCS)*.

[51] Antonio Muñoz, Ruben Rios, Rodrigo Román, and Javier López. 2023. A survey on the (in) security of trusted execution environments. In *Computers & Security*. Elsevier.

[52] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. GraphSC: Parallel Secure Computation Made Easy. In *IEEE Symposium on Security and Privacy (S&P)*.

[53] Shen Noether, Adam Mackenzie, et al. 2016. Ring Confidential Transactions. *Ledger* (2016).

[54] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time Constrained Cycle Detection in Large Dynamic Graphs. In *Proceedings of the VLDB Endowment*.

[55] Leyla Roohi, Benjamin IP Rubinstein, and Vanessa Teague. 2019. Differentially-private Two-party Egocentric Betweenness Centrality. In *IEEE INFOCOM Conference on Computer Communications*.

[56] Edo Roth, Karan Newatia, Yiping Ma, Ke Zhong, Sebastian Angel, and Andreas Haeberlen. 2021. Mycelium: Large-scale distributed graph queries with differential privacy. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*.

[57] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2018. Path ORAM: an extremely simple oblivious RAM protocol. In *Journal of the ACM (JACM)*.

[58] Adithya Vadapalli, Ryan Henry, and Ian Goldberg. 2023. DuORAM: A Bandwidth-Efficient Distributed ORAM for 2-and 3-Party Computation. In *Proceedings of the USENIX Conference on Security Symposium (USENIX Security)*.

[59] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. 2019. Conclave: Secure Multi-party Computation on Big Data. In *Proceedings of The European Conference on Computer Systems (EuroSys)*.

[60] Songlei Wang, Yifeng Zheng, and Xiaohua Jia. 2024. GraphGuard: Private Time-Constrained Pattern Detection Over Streaming Graphs in the Cloud. In *USENIX Security Symposium (USENIX Security)*.

[61] Songlei Wang, Yifeng Zheng, Xiaohua Jia, Qian Wang, and Cong Wang. 2023. MAGO: Maliciously Secure Subgraph Counting on Decentralized Social Graphs. In *IEEE Transactions on Information Forensics and Security (TIFS)*.

[62] Songlei Wang, Yifeng Zheng, Xiaohua Jia, and Xun Yi. 2022. PeGraph: A System for Privacy-Preserving and Efficient Search Over Encrypted Social Graphs. In *IEEE Transactions on Information Forensics and Security (TIFS)*.

[63] Lyu Xu, Byron Choi, Yun Peng, Jianliang Xu, and Sourav S Bhowmick. 2023. A framework for privacy preserving localized graph pattern query processing. In *Proceedings of the International Conference on Management of Data (SIGMOD)*.

[64] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities Based on Ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*.

[65] Juncheng Yang, Yao Yue, and KV Rashmi. 2021. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. In *ACM Transactions on Storage (TOS)*.

[66] A. C. Yao. 1986. How to Generate and Exchange Secrets. In *27th Annual Symposium on Foundations of Computer Science (FOCS)*.

[67] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. 2016. Revisiting Square-root ORAM: Efficient Random Access in Multi-party Computation. In *IEEE Symposium on Security and Privacy (S&P)*.