Highly-Scalable Protected Graph Database Search with Oblivious Filter

Jamie Cui
Ant Group
Hangzhou, China
shanzhu.cjm@antgroup.com

Chaochao Chen
Ant Group
Hangzhou, China
chaochao.ccc@antgroup.com

Li Wang
Ant Group
Hangzhou, China
raymond.wangl@antgroup.com

ABSTRACT

With the emerging popularity of cloud computing, the problem of how to query over cryptographically-protected data has been widely studied. However, most existing works focus on querying protected relational databases, few works have shown interests in graph databases. In this paper, we first investigate and summarise two single-instruction queries, namely Graph Pattern Matching (GPM) and Graph Navigation (GN). Then we follow their design intuitions and leverage secure Multi-Party Computation (MPC) to implement their functionalities in a privacy-preserving manner. Moreover, we propose a general framework for processing multi-instruction query on secret-shared graph databases and present a novel cryptographic primitive Oblivious Filter (OF) as a core building block. Nevertheless, we formalise the problem of OF and present its constructions using homomorphic encryption. We show that with OF, our framework has sub-linear complexity and is resilient to access-pattern attacks. Finally, we conduct an empirical study to evaluate the efficiency of our proposed OF protocol.

KEYWORDS

Protected Graph Database Search; Secret Sharing; Secure Multiparty Computation; Oblivious Filter

1 INTRODUCTION

Graph Database-as-a-Service (GDaaS) has gained much popularity in the industry recently. Many cloud service providers (CSPs) offer built-in GDaaS such as Google's *Aura* and Amazon's *Neptune*. How to resolve people's concern over the confidentiality of their data and provide search functionality has become one of the most imperative challenges for cloud databases.

This problem has been comprehensively studied for relational databases. First introduced in 2000 by Song [53], the problem of *Protected Database Search* (PDS) has motivated the research in different aspects such as Private Information Retrieval (PIR) [18, 27], Searchable Symmetric Encryption (SSE) [13], and Order-Preserving Encryption (OPE) [3]. Roughly, in PDS, data owner securely outsources its data onto cloud servers (presumably in an encrypted or secret-shared form), and then client performs searches on the protected data. The security of PDS requires that cloud servers eventually learn nothing about the securely-outsourced data except the search results. To date, existing work on PDS can be divided into three categories: encryption based solutions[42, 44, 50, 55, 58, 59], secret-sharing based solutions [6], and trusted-hardware based solutions [26]. Despite the extensive study of PDS problem, most of the work target on relational databases [6, 26, 42, 44, 50, 55, 58, 59]. There is a lack of study on searching over protected graph databases. Unlike relational databases, graph databases by design maintain the relationship between nodes, which allows fast and efficient connection check between two nodes [8]. To bridge the gap, in this work, we bring the PDS problem to graph databases, and study the *Protected Graph Database Search* (PGDS) problem.

Problem and Chanllenges. Similar to PDS, PGDS aims to provide search functionality over on a cryptographically-protected and outsourced graph database. Many works [6, 45] and industrial systems such as Crypten [1] and TFEncrypted [20] leverage secret-share secure computation model to provide protection for secret data. Therefore, we focus on the PGDS problem where the outsourced graph data is protected by a secret sharing scheme (e.g. [52]). Generally, in our solution of PGDS, we assume the existence of three types of participants:

- An *honest data holder* $\mathcal{H}$ who holds a private graph database but has limited storage and computation resources, and therefore $\mathcal{H}$ outsources its private data to servers;
- Two *non-colluding cloud servers* $\mathcal{S}_0, \mathcal{S}_1$ who have practical storage, computation, and networking capability;
- An *honest client* who C performs secure searching functionality by interacting with $\mathcal{S}_0$ and $\mathcal{S}_1$.

Intuitively, for privacy concern, cloud servers should not learn anything about the underlying graph data except for metadata (e.g., the number of vertices and edges). Also, since we facilitate two-party secret sharing for secure computation, we further restrict our threat model to a single semi-honest adversary.

There are two challenges towards the secret-sharing based PGDS problem. First, *scalability.* This is because graph data can contain millions or even trillions of edges and vertices in the real-world application, e.g. social graphs, interest graphs, and consumption graphs. Therefore it is essential for the solution to be highly-scalable. Second, *security.* Previous works on PDS leaks access patterns in order to boost performance [13], which further broadens the attack plane. Recent work [36] has demonstrated a practical attacks utilising access patterns. In this work, we aim at eliminating all information leakages, especially access patterns.

1.1 Existing Work

Apart from the vast amount of work on relational databases based PDS, recently, Lai et al. [39] proposed GraphSE^2, which addresses the problem of PGDS by leveraging a SSE scheme called Oblivious Cross-Tags (OXTs) [13] protocol. In particular, GraphSE^2 provides an encrypted structural data model to facilitate parallel and encrypted data access (like set operations and graph traversal). However, it also inherits the leakage from the original OXT protocol and inevitably leaks access patterns, and thus is prone to access pattern attacks, which contradicts the initiatives of this work.

1.2 Our Contribution

To resolve the above challenges, in this paper, we propose a novel query processing system for secret-shared PGDS problem. In particular, our system can be seen as a graph traversal machine working on secret-shared graph databases. There are two general features supported by our system: *Graph Pattern Matching* (GPM) and *Graph Naviagtion* (NG). We begin by introducing a novel graph sharing scheme which serves as the infrasturcture of our system, and then present the secure evaluation of GPM based on Ullmann's algorithm [56], and secure evaluation of GN based on Gremlin [49], a popular industrial query processing machine. Afterwards, we propose a secure evaluation framework for multi-instruction query, which allows the evaluation of composed single instructions. The proposed secure evaluation framework for multi-instruction query relies on a novel cryptographic primitive: *Oblivious Filter.* Similar to the famous oblivious transfer [17, 38], oblivious filter is a two-party functionality between a server and a client, where the client obliviously filters an input list from server, and obtains the result in a secret-shared form. During the protocol, both server and client know nothing about the result. We also present a variant of OF, i.e., *Secret-Shared Oblivious Filter*, which can be used as a secret-input and secret-output blackbox filter, improving the scalability of our framework. Afterwards, we provide a HE-based construction of OF and prove its security under the simulation-based security.

Contribution. We summarize our main contributions in this paper as follows.

- We introduce a new graph sharing scheme which allows efficient connectivity check between two nodes, and has practical storage complexity. The graph sharing scheme only leaks the metadata.
- We summarize exiting queries on graph database into two single instruction queries, i.e., GPM and GN, and propose their secure evaluations.
- We propose a secure graph traversal machine based on oblivious filter. Our traversal machine works on secret-shared graph databases, and allows evaluation on the sequential composition of multiple instructions.
- We introduce a new cryptographic primitive, i.e., oblivious filter, allowing obliviously filtering the database with conditions. We also provide its HE-based construction, which is secure under a semi-honest non-adaptive adversary.
- We conduct an empirical study to evaluate the performance of our proposed building block — oblivious filter.

2 BACKGROUNDS AND TOOLS

In this section, we present some background knowledge and definitions about graphs, databases, and some cryptographic primitives. In summary, Section 2.1 first introduces two secure computation techniques: secret sharing and additive homomorphic encryption. Then Section 2.2 shows how to model and represent graph data. Finally Section 2.3 takes a in-depth look at two fundamental query functionalities, i.e., graph pattern matching and graph navigation.

Syntax. Our solution involves many interactions among *a data holder* $\mathcal{H}$, *two cloud servers* $\mathcal{S}_0, \mathcal{S}_1$, and *a client* C. In the following of this paper, we use a pair with angle brackets, i.e., $\langle$ and $\rangle$, to denote the data that is only visible to the corresponding party. The ordering that we use is $\langle \mathcal{H}, \mathcal{S}_0, \mathcal{S}_1, C\rangle$. For example $\langle x, y, z, k\rangle$ means that x is only visible to $\mathcal{H}$, y is only visible to $\mathcal{S}_0$, z is only visible to $\mathcal{S}_1$, and k is only visible to C. Also we use $\perp$ to represent empty. We further simplify the notation and omit the empty symbol $\perp$. That is, we use $[\![x]\!]$ or $\langle [\![x]\!]_0, [\![x]\!]_1\rangle$ as an abbreviation of $\langle \perp, [\![x]\!]_0, [\![x]\!]_1, \perp\rangle$, $\langle x, \circ\rangle$ as an abbreviation of $\langle x, \perp, \perp, \perp\rangle$, and $\langle \circ, x\rangle$ as an abbreviation of $\langle \perp, \perp, \perp, x\rangle$.

2.1 Secure Computation Techniques

Secure computation addresses the main challenge of the secret-share based PGDS problem. We introduce the basic concepts and notations of two secure computation techniques, i.e., secret-sharing and additive homomorphic encryption.

Secret sharing. Secret sharing is a cryptography primitive aiming to distribute a secret among a group of parties (participants), such that each party holds a share of the secret [52]. And secret sharing ensures that only with a sufficient amount of participants, the secret can be revealed. More formally, a secret sharing scheme is a pair of algorithms between t parties: $(\mathsf{Shr}, \mathsf{Rec})$. Since in Secret Shared Graph Database (SSGD), we only consider the situation with a distributor and two receivers, the functionalities of secret sharing from H to S_0 and S_1 could be denoted as:

- $\langle [\![x]\!]_0, [\![x]\!]_1\rangle \leftarrow \mathsf{Shr}(\langle x, \circ\rangle)$,
- $\langle x, \circ\rangle \leftarrow \mathsf{Rec}(\langle [\![x]\!]_0, [\![x]\!]_1\rangle)$.

Additionally, secret-sharing scheme has been the building block for many multiparty computation (MPC) protocols [24, 29]. We use additive secret sharing with the field of power of two to facilitate efficient arithmetic and boolean operations.

Additive homomorphic encryption. Additive homomorphic encryption is an asymmetric encryption scheme which allows addition operation on the encrypted data [22, 47, 48]. With a formal description, additive homomorphic encryption is a tuple of algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$, with "+" as its homomorphic operation. Briefly,

- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)$: Taking a security parameter λ, Gen generates the public key pk and secret key sk,
- $c \leftarrow \mathsf{Enc}_{\mathsf{pk}}(m)$: Enc encrypts a message m with public key, and returns a cipher c,
- $m \leftarrow \mathsf{Dec}_{\mathsf{sk}}(c)$: Dec decrypts a cipher c with secret key, and reveals the message m.

As for additive homomorphism, those schemes allow the secure evaluation of $\mathsf{Enc}(m_1 + m_2)$ using two ciphers c_0, c_1 and the public key, that is, $\mathsf{Enc}_{\mathsf{pk}}(m_1 + m_2) = \mathsf{Eval}(\mathsf{Enc}_{\mathsf{pk}}(m_1), \mathsf{Enc}_{\mathsf{pk}}(m_2))$. The security of additive HE schemes is defined by the generally-adopted semantic security [30] with a computationally-bounded adversary. Appendix A describes the detailed construction of an additive homomorphic encryption scheme — paillier encryption [48].

2.2 Graph Model and Representation

Graph databases use graph model as the basic data structure for graphs. To begin with, there are two most popular graph models in the literature: Resource Description Framework (RDF) [9] and Labeled Property Graph (LPG) [5]. For simplicity, in this work, we choose a limited, yet simple and widely-adopted variant of LPG model — *Edge-labeled Graph Model* [4], where edges are assigned with labels to indicate different relationships between nodes. In the following, we give a formal definition of edge-labeled graph model.

Definition 2.1 (Edge-labeled graph). An edge-labeled graph G is a pair (V, E), where $V = \{v_1, v_2, ..., v_n\}$ is a finite set of vertices, and $E = \{e_1, e_2, ..., e_m\}$ is a finite set of edges, with n and m denoting the number of vertices and edges, respectively. In addition, $E \subseteq V \times L \times V$, where L is a set of labels.

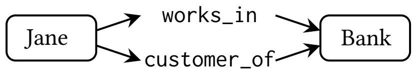

Figure 1: A simple example of edge-labeled graph, where `works_in` and `customer_of` are two labels between vertices 'Jane' and 'Bank'.

Most existing graph databases use either adjacency matrix or adjacency list to represent edge-labeled graph, e.g. OrientDB[1], Neo4j[2], and MS Graph Engine[3]. Theoretically, *adjacency matrix* has storage complexity of $O(n^2)$, and can check connectivity in $O(1)$. In contrast, *adjacency list* has storage complexity of $O(n + m)$, and can check connectivity in $O(|A_u|) \subseteq O(\hat{d})$, where A_u stands for the adjacency list for vertex u, and $\hat{d}$ denotes the maximum degree in G. We notice that the storage requirement of adjacency matrix grows squarely with the size of vertices, and thus is not an ideal solution for large-scale graph data. Thus, we adopt the adjacency list to represent the underlying graph data.

More concretely, to represent a string-format graph, we first use two string lists V and L to store the mappings between a valid string and its ID. That is, $V : \Sigma^* \rightarrow \mathsf{ind}(V)$ and $L : \Sigma^* \rightarrow \mathsf{ind}(L)$, where we use Σ^* to denote a valid string, and $\mathsf{ind}(*)$ to denote the indexes of a certain type. Afterwards, we use indexed adjacency list to denote the relationships between vertex-IDs. For the example in Figure 1, let $V = \{$"Jane" $: v_1$, "Bank" $: v_2\}$, $L = \{$"`works_in`" $: l_1$, "`customer_of`" $: l_2\}$, the above example can be represented in the following indexed adjacency list:

$$v_1 : (l_1, v_2), (l_2, v_2),$$
$$v_2 : \backslash.$$

2.3 Graph Query Functionalities

Graph query languages express the searching functionality and serve as the core component of graph database systems. Currently, there are many practical graph query languages in the industry, e.g. Cypher[25], SPARQL[33] and Gremlin[49]. Even though these languages differ enormously in purpose, expressivity, implementation, and so on. On the high level, graph query languages share two most fundamental functionalities [4, 8], namely *Graph Pattern Matching* (GPM) and *Graph Navigation* (GN).

2.3.1 Graph Pattern Matching (GPM). In the theory of graph query languages, GPM refers to the problem of finding exact graph pattern matches for a given graph. Formally, we define an edge-labeled graph pattern as a graph $\tilde{G}$ comprises of constants and variables,

[1]OrientDB: https://www.orientdb.org/
[2]Neo4j: https://neo4j.com/
[3]MS Graph Engine: https://www.graphengine.io/

Inputs. Graph G, graph pattern $\tilde{G}$, and a possible match set H.
Outputs. False if there is no match; Return the matches otherwise.
Algorithm. Starts from an empty root. For variable $\delta_i \in \mathsf{Var}(\tilde{G})$, every node in the search-tree at level i represents a mapping from δ_i to a possible vertex $v \in G$. Then every path that connecting the root and the leaf is a possible match $h \in H$. To find a match:

(1) Prune subtrees by eliminating repeated variables values, that is, the matching h is an injective (one-to-one) mapping.
(2) Forward check if all the edges connecting two nodes in the tree preserve the relationships between their corresponding variables, if not, delete the edges.
(3) Return the path which remained in the tree from root to a leaf.

Example.

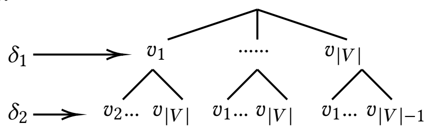

Figure 2: Ullmann's match algorithm [56] for edge-labeled graph.

where constants are denoted as $\mathsf{Const}(G) \subseteq V \cup E \cup L$ and variables are denoted as $\mathsf{Var}(G)$. For instance, a graph pattern query "Search for the people that `Marko` `knows`" towards graph $G = (V, E)$ could be turned into a graph pattern $\tilde{G} = (\tilde{V}, \tilde{E})$, where $\tilde{V} = \{$"`Marko`"$, \delta\}$, and $\tilde{E} = \{($"`Marko`", "`knows`"$, \delta)\}$, where δ is the variable. To find a match, the graph pattern $\tilde{G}$ is first matched to the graph G, and then the graph database searches for the occurrences of the pattern. The GPM problem is known to be NP-hard theoretically by reduction from the graph homomorphism problem [31]. However, in reality, the size of the graph pattern query is usually much smaller than the size of graph database, and many existing works [2, 16, 43] have showed that the GPM problem can be solved in polynomial time and even subpolynomial time with a fixed-size query. More formally, a match for a graph pattern is defined as follows:

Definition 2.2 (Match). Given an edge-labeled graph $G = (V, E)$ and a graph pattern $\tilde{G} = (\tilde{V}, \tilde{E})$, a match h of $\tilde{G}$ in G is a mapping from $\mathsf{Const} \cup \mathsf{Var}$ to Const such that the mapping h maps constants to themselves and variables to constants; if the image of $\tilde{G}$ under h is contained within G, then h is a match.

Technically, we follow Ullmann's basic approach [56] to find the matches. It requires to list all possible mappings of vertices in the graph pattern and invoke a depth-first tree search algorithm. Since Ullmann's algorithm has exponential search space, it is common to involve a pre-processing phase to prune unpromising subtrees. We denote the match search tree of graph G as $\mathsf{Tree}(G)$, and show the details of Ullmann's match algorithm in Figure 2.

An example. Now, we give an example of processing the GPM query in Figure 3, where Figure 3 (a) describes a graph data $G = (V, E)$ and Figure 3 (b) shows a graph pattern query $\tilde{G} = (\tilde{V}, \tilde{E})$, and the query is "Find the person who Jim knows, and the place this person works in". Following Ullmann's algorithm, first we build the possible search tree for graph pattern $\tilde{G}$ and prune the repeated values. As the result, we obtain Figure 4 with a total of six possible matches.

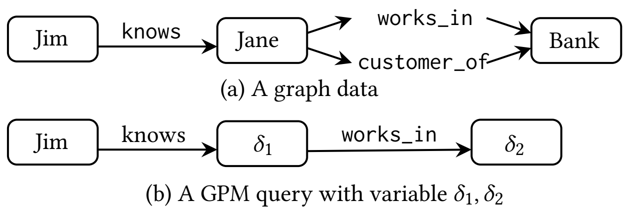

Figure 3: An example of the GPM problem.

To find the match of $\tilde{G}$ in the depth-first manner, we begin by looking at the first value of δ_1 on level 1 (that is "Jim"). In the graph pattern $\tilde{G} = (\tilde{V}, \tilde{E})$, we extract the edges containing δ_1 only, that is, ("Jim", "knows", δ_1). By substituting δ_1 with its value "Jim", we obtain ("Jim", "knows", "Jim"), which is not an edge in the original graph, therefore, we delete this node and its subtrees. After many iterations, the algorithm reaches the end, and the remaining paths are the matches. In this example, we have δ_1 = "Jane" and δ_2 = "Bank" (the dash line in Figure 4).

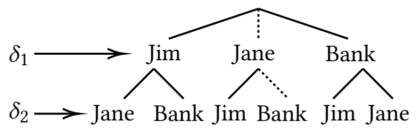

Figure 4: Possible matching tree.

2.3.2 Graph Navigation (GN). In general, GN allows navigation towards the graph topology. GN queries have long been established as the core of navigational querying in graphs by the research community [7, 11, 57] and are widely adopted in graph query languages (e.g. SPARQL, Cypher and Gremlin). One typical example of such a query is "finding all friends-of-a-friend of Marko". Here we are not only interested in the immediate acquaintances of a person, but also the people she might know through other people; namely, her friends-of-a-friend, their friends, and so on. Traditionally, the functionality of graph navigation uses a basic component called *Path Query*, where path query navigates through arbitrary number of edges in the graph. Formally, we define path query as follows.

Definition 2.3 (Path Query). Given an edge-labeled graph $G = (V, E)$, a path query has the general form $P = x \to^{\alpha} y$, where x, y specify the beginning and the ending nodes (vertices) of the path, and α denotes the condition (traversal instruction) on the paths.

Generally, the evaluation of a path query $P = x \to^{\alpha} y$ over G, denoted as $P(G)$, consists of all the paths in G who conform the condition α. Due to the fact that the evaluation of a path query differs vastly from different systems, we focus on Gremlin's processing system, and introduce the details in Section 2.3.3.

2.3.3 Graph Query Language Processing. Industrial graph query processing systems use different mechanisms and semantics to process GPM and GN queries. For instance, SPARQL [33] processes graph queries in a non-instructive and expressive fashion, while Gremlin [49] processes graph queries through graph traversal.

In this work, we focus on the more general framework — Gremlin's graph traversal machine. A traversal machine in Gremlin is defined as $t \in T$, which traverses over a graph G according to an instruction set Ψ. Here, we take a simplified formalization of a graph traversal machine for edge-labeled graph, that is,

$$T = (U \times \Psi),$$

where $U = V \cup E \cup L$ is the traverser's location in the graph G. Evaluating a single step traversal instruction $\psi \in \Psi$ in Gremlin can be taken as a specification of one of the following maps:

(1) flatMap
- $\mathcal{P}(V) \to \mathcal{P}(E)$: Move the traverser from vertices to edges;
- $\mathcal{P}(E) \to \mathcal{P}(V)$: Move the traverser from edges to vertices;
- $\mathcal{P}(U) \times \tilde{G} \to \mathcal{P}(U)$: Move the traverser according to graph pattern $\tilde{G}$;

(2) filter
- $\mathcal{P}(E) \times I_e \to \mathcal{P}(E)$: Filter edges with the index set I_e;
- $\mathcal{P}(V) \times I_v \to \mathcal{P}(V)$: Filter vertices with the index set I_v;
- $\mathcal{P}(E) \times L \to \mathcal{P}(E)$: Filter edges with the label $l \in L$;

where we use $\mathcal{P}(*)$ to denote the power set of $*$. Since we use a simplified representation for the graph traverser, it only supports the most basic graph traversal — *Simple Traversal*, where traversal instructions are processed in a sequential order ($\psi_1 \rightsquigarrow \psi_2 \rightsquigarrow \dots \rightsquigarrow \psi_{|\Psi|}$). For the rest of this paper, we use the term "*instruction*" or "*traversal instruction*" to short for a single step traversal instruction.

Following this intuition, we further detail the categorization of our graph query languages, namely *single-instruction query* and *multi-instruction query*, where a single-instruction query contains only one traversal instruction and multi-instruction query consists of multiple instructions. In Section 4 and Section 5, we will present how to securely evaluate single-instruction query and multi-instruction query on secret shared graph database, respectively.

3 GRAPH SHARING SCHEME

Though we specify the plaintext graph model and representation, it still remains a challenge of how to securely and efficiently share the graph between two parties. One the one side, the graph sharing scheme should be secure against a semi-honest adversary, and on the other side, it should also support: (1) *secure validity check* for a constant string (whether the string exists in vertices set or label set); (2) *secure connectivity check* for two string-format vertices, including the case of shared string vs. public string, shared string vs. shared string, and public string vs. public string. Here, shared means that the string is secret shared between $\mathcal{S}_0, \mathcal{S}_1$, and public means that the string is public to both $\mathcal{S}_0$ and $\mathcal{S}_1$.

Intuitively, as is shown in the previous section, an edge-labeled graph G could be sufficiently represented by a vertex list V, a label list L, and an adjacency list A. This graph representation method uses V and L for validity check of a string and retrieves the string's index if it is valid. Afterwards, given two vertex-IDs, the adjacency list A checks the connectivity of the two vertices.

In our solution, we adopt such design intuition and utilize two different data structures for the graph sharing scheme, namely *shared lookup table* and *shared adjacency list*. To begin with, we use $\mathsf{hash}(v)$ to represent hashed strings, where $\mathsf{hash}(*)$ is a cryptographic hash function. First, we hash all the strings in the original graph, and share the hashed values $S = \mathsf{hash}(V) \cup \mathsf{hash}(L)$, i.e., $\langle [\![S]\!]_0, [\![S]\!]_1 \rangle \leftarrow \mathsf{Shr}(\langle S, \circ \rangle)$. The shared string list is later used for string's validity checking. Then, to allow secure connectivity check of two vertices, we build a shared index lookup table $[\![T]\!]$, which allows index retrieving for a vertex string or a label string. Finally, we share the indexed adjacency list $[\![A]\!]$ to allow the connectivity check between two vertices. We will describe the details of shared lookup table $[\![T]\!]$ and shared adjacency list $[\![A]\!]$ later. In summary, a shared graph $[\![G]\!]$ is represented as:

$$[\![G]\!] = ([\![S]\!], [\![T]\!], [\![A]\!]).$$

Shared lookup tables $[\![T]\!]$. Formally, a shared lookup table associates with algorithms BuildT and $\mathsf{LookupT}$. Let $X = \{x_0, ..., x_{d-1}\}$ and $Y = \{y_0, ..., y_{d-1}\}$ be two sets with the same length d, and we want to share the mapping $X \mapsto Y$. Additionally, we define the final result $[\![T]\!]$ as a set of polynomial coefficients $[\![a_0]\!], ..., [\![a_{d-1}]\!]$, where the shared $[\![a_i]\!]$ are the coefficients used to retrieve the mapping results. For sharing the graph, we build a shared index lookup table by $[\![T]\!] \leftarrow \mathsf{BuildT}(\mathsf{hash}(V), \mathsf{ind}(V))$.

Notation. $X = \{x_0, ..., x_{d-1}\}$ and $Y = \{y_0, ..., y_{d-1}\}$ have d elements, and we also define $[\![T]\!] = [\![a_0]\!], ..., [\![a_{d-1}]\!]$.
Algorithm 1. $[\![T]\!] \leftarrow \mathsf{BuildT}(X, Y)$.

(1) Data holder gets the polynomial coefficients from the input mapping: $a_0, ..., a_{d-1} \leftarrow \mathsf{LagrangeInterpolation}(X \mapsto Y)$;
(2) For every $a \in \{a_0, .., a_{d-1}\}$, data holder shares the coefficients to servers: $[\![a]\!] \leftarrow \mathsf{Shr}(\langle a, \circ \rangle)$.

Algorithm 2. $[\![y]\!] \leftarrow \mathsf{LookupT}(x, [\![T]\!])$.

(1) $\mathsf{S_i}$ calculates $[\![y]\!]_i = [\![a_0]\!]_i + [\![a_1]\!]_i x + ... + [\![a_{d-1}]\!]_i x^{d-1}$.

Figure 5: Algorithms for shared lookup table.

In Figure 5, we use the term "$\mathsf{LagrangeInterpolation}$" to denote the function that outputs a polynomial (degree $d-1$) that interpolates the input mapping. The resulting coefficients $a_0, .., a_{d-1}$ and the polynomial $f(x) = \sum_{i=0}^{d-1} a_i x^i$ satisfies that $f(x_i) = y_i$ for all $x_i \in X$ and $y_i \in Y$. The computation complexity of computing all coefficients of $\mathsf{LagrangeInterpolation}$ is $O(d \log d)$ according to [54], and sharing the coefficients requires $d|[\![*]\!]|$-bit communication, where $|[\![*]\!]|$ indicates the bit size of the shares. As for the lookup algorithm, evaluating the polynomial only requires d additions and d multiplications using Horner's Method [34], and lookup algorithm requires no extra communication cost.

Shared adjacency list $[\![A]\!]$. The objective of a shared adjacency list is to allow the connectivity check for two vertices with a given label. A simple way is to reconstruct all the indexes and apply a plaintext lookup, but unfortunately this leaks access patterns. To share the adjacency list, first we use an indexed representation for the head node (the u of the A_u) of the adjacency list, and then we include the tail node's index in the edge representation, and pad every line in adjacency list to the maximum degree $\hat{d}$ with "0"s. Finally, we share the mapped element in the adjacency list (notice that we do not need to share the head vertex indexes as long as they are stored in order). For example, the shared adjacency list of Figure 1's case is:

$$\begin{aligned} &\mathsf{ind}(v_0): \\ &\qquad ([\![\mathsf{ind}(l_0)]\!]_i, [\![\mathsf{ind}(v_1)]\!]_i), ([\![\mathsf{ind}(l_1)]\!]_i, [\![\mathsf{ind}(v_1)]\!]_i) \\ &\mathsf{ind}(v_1): \\ &\qquad ([\![\mathsf{ind}(0)]\!]_i, [\![\mathsf{ind}(0)]\!]_i), ([\![\mathsf{ind}(0)]\!]_i, [\![\mathsf{ind}(0)]\!]_i) \end{aligned}$$

where $\mathsf{S_0}$ holds the list with $i = 0$ and $\mathsf{S_1}$ holds the list with $i = 1$. This secret sharing scheme has storage complexity of $O(4n\hat{d}+2n+m)$ on both server's size and checks connectivity of two vertices with $O(\hat{d})$ secure equality test operations. We denote the adjacency list as A, and shared adjacency list as $[\![A]\!]$.

Security. This sharing scheme for graphs inevitably inherits the leakage profile from shared lookup table and shared adjacency list. All those leakages are leaked to servers $\mathsf{S_0}$ and $\mathsf{S_1}$. Specifically, $[\![S]\!]$ leaks the total number of vertices and labels ($|V| + |L|$), $[\![T]\!]$ leaks the number of vertices ($|V|$), and the shared adjacency list $[\![A]\!]$ additionally leaks the maximum degree of the graph ($\hat{d}$). Here, we only discuss the leakage of those data structures, and we leave the security of connectivity check in Section 4. In summary, the leakage profile of this sharing scheme is defined as:

$$\mathcal{L}_{\mathsf{share}} = (|V|, |L|, \hat{d}).$$

4 SECURE EVALUATION OF A SINGLE-INSTRUCTION QUERY

In the previous section, we introduce how to convert GPM (graph pattern) queries and GN (path) queries into traversal instructions. In this section, we show the secure processing of single-instruction queries (e.g. a GPM query or a GN query which could be translated into only a single instruction). The security of our protocols follows the standard semi-honest definition using simulation-based technique [12, 28], where security says the behaviour of the adversary can be simulated given only the view of an honest participant.

Methodology. Aiming at secure and efficient single-instruction query evaluation, we first formalise the ideal functionalities of answering GPM and GN queries. We then present and analyse the secure constructions using real world vs. ideal world simulation paradigm. To allow an easy analysis, we describe functionalities with a leakage profile $\mathcal{L}$, which models the information leakage. Also, some of our proposed protocols leverage a known secure equality test protocol in MPC. To give an abstraction of the whole protocol, we prove the security in the *MPC-hybrid model*, where we assume the presence of a secure equality test functionality EQ. The detailed constructions are shown in [14].

4.1 Securely Evaluating A GPM Query

Recall that a GPM query is essentially a graph pattern that could be represented as a graph $\tilde{G} = (\tilde{V}, \tilde{E})$, containing constants and variables. Formally, we adopt the graph pattern matching algorithm from Ullmann[56], and describe the functionality of GPM as $\mathcal{F}_{\mathsf{GPM}}$

in Figure 6, where we use $\mathsf{isIn}(x \in \mathcal{X}, \mathbf{X} \in \mathcal{X}^*)$ to denote an algorithm which checks if the element x belongs to the set $\mathbf{X}$. This functionality works between C, $\mathcal{S}_0$, and $\mathcal{S}_1$ and allows the secure evaluation of a graph pattern query (containing a single instruction) over the secret shared graph database. The ideal functionality $\mathcal{F}_{\mathsf{GPM}}$ could be split into three phases:

1. *Check phase*, where the client checks the validity of a given graph pattern by sending all strings in graph pattern to the ideal functionality;
2. *GetID phase*, where the client retrieves IDs for every query string;
3. *Query phase* where the client first builds $\mathsf{Tree}(\tilde{G})$, and then checks the existence of an edge in $\mathsf{Tree}(\tilde{G})$ in a depth-first-search manner.

```
Global Parameters. Security parameter $\lambda$.
Check. Given a hash set Hset from $C$, and $[\![G]\!]_i$ from $\mathcal{S}_i$,
  (1) Invokes $(S, T, A) \leftarrow \mathsf{Rec}([\![G]\!])$;
  (2) For all $\sigma_j \in \mathsf{Hset}$, invokes $b_j \leftarrow \mathsf{isIn}(\sigma_j, S)$. Leaks $|\mathsf{Hset}|$ to $\mathcal{S}_i$;
  (3) Returns $b_1, ..., b_{|\mathsf{Hset}|}$ to $C$.
GetID. Given a hash set Hset from $C$, and $[\![G]\!]_i$ from $\mathcal{S}_i$,
  (1) Invokes $(S, T, A) \leftarrow \mathsf{Rec}([\![G]\!])$;
  (2) For all $\sigma_j \in \mathsf{Hset}$, invokes $\mathsf{ind}_j \leftarrow \mathcal{F}_{\mathsf{LookupT}}(\sigma_j, T)$;
  (3) Returns $\mathsf{ind}_1, ..., \mathsf{ind}_{|\mathsf{Hset}|}$ to $C$.
Query. On receiving Eset from $C$, and $[\![G]\!]_i$ from $\mathcal{S}_i$,
  (1) Invokes $(S, T, A) \leftarrow \mathsf{Rec}([\![G]\!])$;
  (2) For every $(h_j, l_j, t_j) \in \mathsf{Eset}$, finds $A_{h_j}$, and invokes $b_j \leftarrow \mathsf{isIn}((l, t), A_{h_j})$;
  (3) Returns $b_1, ..., b_k$ to $C$.
```

Figure 6: Ideal Functionality $\mathcal{F}_{\mathsf{GPM}}$.

Protocols in Figure 7, 8, and 9 describe the detailed constructions of *Check phase*, *GetID phase*, and *Query phase* for functionality $\mathcal{F}_{\mathsf{GPM}}$, respectively.

```
C.Check. Given a graph pattern $\tilde{G}$, client $C$:
  (1) Initiates $\mathsf{Hset} = \emptyset$;
  (2) For all $\phi_j \in \mathsf{Const}(\tilde{G})$, calculates and pushes $\mathsf{hash}(\phi_j)$ into Hset;
  (3) Sends Hset to servers $\mathcal{S}_i$;
  (4) On receiving messages from $\mathcal{S}_i$, invokes $b_1, ..., b_{|\mathsf{Hset}|} \leftarrow \mathsf{Rec}(\langle [\![b_1]\!]_0, ..., [\![b_{|\mathsf{Hset}|}]\!]_0, [\![b_1]\!]_1, ..., [\![b_{|\mathsf{Hset}|}]\!]_1 \rangle)$, aborts if $\exists j \leq |\mathsf{Hset}|, b_j = 0$.
S_i.Check. On receiving a hashed list Hset from $C$, server $\mathcal{S}_i$:
  (1) Initiates $[\![b_1]\!]_i, ..., [\![b_{|\mathsf{Hset}|}]\!]_i = 0$;
  (2) For every $\sigma_j \in \mathsf{Hset}$:
    (a) For every shared hashed string $[\![s]\!]_i \in [\![S]\!]_i$, calculates $[\![b_j]\!]_i = [\![b_j]\!]_i + \mathsf{EQ}(\sigma_j, [\![s]\!]_i)$;
  (3) Sends $[\![b_1]\!]_i, ..., [\![b_{|\mathsf{Hset}|}]\!]_i$ to $C$.
```

Figure 7: Protocol Π_{GPM}.Check for functionality $\mathcal{F}_{\mathsf{GPM}}$.

```
C.GetID. Given a graph pattern $\tilde{G}$, client $C$:
  (1) Initiates $\mathsf{Hset} = \emptyset$;
  (2) For all $\phi_j \in \mathsf{Const}(\tilde{G})$, calculates and pushes $\mathsf{hash}(\phi_j)$ into Hset;
  (3) Sends Hset to servers $\mathcal{S}_i$;
  (4) On receiving $\mathcal{S}_i$'s messages, invokes $\mathsf{ind}_1, ..., \mathsf{ind}_{|\mathsf{Hset}|} \leftarrow \mathsf{Rec}(\langle [\![\mathsf{ind}_1]\!]_0, ..., [\![\mathsf{ind}_{|\mathsf{Hset}|}]\!]_0, [\![\mathsf{ind}_1]\!]_1, ..., [\![\mathsf{ind}_{|\mathsf{Hset}|}]\!]_1 \rangle)$.
S_i.GetID. On receiving a hash set Hset from $C$,
  (1) For all $\sigma_j \in \mathsf{Hset}$, invokes $[\![\mathsf{ind}_j]\!]_i \leftarrow \mathsf{LookupT}([\![T]\!], \sigma_j)$;
  (2) Sends $[\![\mathsf{ind}_1]\!]_i, ..., [\![\mathsf{ind}_{|\mathsf{Hset}|}]\!]_i$ to $C$.
```

Figure 8: Protocol Π_{GPM}.GetID for functionality $\mathcal{F}_{\mathsf{GPM}}$.

```
C.Query. Once client has $\mathsf{ind}_1, ..., \mathsf{ind}_{|\mathsf{Hset}|}$,
  (1) Replaces all $\phi_j \in \mathsf{Const}(\tilde{G})$ with $\mathsf{ind}_j$;
  (2) Builds the match search tree $\mathsf{Tree}(\tilde{G})$ by replacing variables with possible vertex-IDs, then prunes repeated vertex-IDs over a possible match path (from root to a leaf).
  (3) Performs forward check in a depth-first manner, for each possible match path, at level $j$ of $\mathsf{Tree}(\tilde{G})$,
    (a) Let Eset represents all edges in $\tilde{G}$ containing $\delta_j$ only or $\delta_j$ with lower-level variables;
    (b) Sends Eset to $\mathcal{S}_i$ as Query2;
    (c) Invokes $b \leftarrow \mathsf{Rec}(\langle [\![b]\!]_0, [\![b]\!]_1 \rangle)$;
    (d) Delete the nodes if $b = 0$.
S_i.Query. On receiving Eset from $C$, for every $(h, l, t) \in \mathsf{Eset}$:
  (1) Finds the shared adjacency list $[\![A_h]\!]_i$;
  (2) Initiates $[\![b]\!]_i = 1$;
  (3) For each edge $(h, [\![l']\!]_i, [\![t']\!]_i) \in [\![A_h]\!]_i$, $[\![b]\!]_i = ([\![b]\!]_i + 2 - (\mathsf{EQ}([\![l']\!]_i, l) - \mathsf{EQ}([\![t']\!]_i, t)))$;
  (4) Sends $[\![b]\!]_i$ back to $C$.
```

Figure 9: Protocol Π_{GPM}.Query for functionality $\mathcal{F}_{\mathsf{GPM}}$.

LEMMA 4.1. *Protocol Π_{GPM} is secure against non-adaptive adversary in the MPC-hybrid model.*

Proof sketch. For the security proof of Π_{GPM}, we use a simulator for each server in an invocation of Π_{GPM}.Check, Π_{GPM}.GetID and Π_{GPM}.Query, and prove its security in the MPC-hybrid model.

Simulator for $\mathcal{S}_i$: During the Check phase, $\mathcal{S}_i$ eventually receives nothing, therefore the simulation for Check phase is trivial. To simulate the getID phase and query phase, the simulator uniformly samples random IDs for every edge header ϕ_*, and then obtains the final result. This simulation is indistinguishable from the real execution under the condition that the simulator cannot distinguish between a randomly sampled vertex-ID and the real vertex-ID for a specific string. Since we only assume a non-adaptive semi-honest adversary and the real vertex-IDs are no-repeatable and independently distributed, the simulation completes.

Communication efficiency. The check phase only runs in one round and has communication complexity of $O(|\mathsf{Const}(Q)|)$ for C, $\mathcal{S}_0$, and $\mathcal{S}_1$. And the getID phase has communication complexity of

$O(|\mathsf{Const}((Q)|)$ for $\mathcal{S}_0$ and $\mathcal{S}_1$. As for query phase, the communication complexity depends on the actual query, while at the worst case, it invokes $O(kt(\hat{d}+n+m))$ equality test operations, where k is the number of variables in the query, t is the number of all possible combinations of variables, i.e., the length of the list L, and $\hat{d}$ is the maximum degree of vertices. Existing frameworks such as SPDZ [19, 23, 37] can perform equality test with online communication complexity of $O(\beta)$ bits for β-bit integers [21]. This brings the total communication complexity of secure GPM to $O(kt\beta(\hat{d}+n+m))$ bits.

4.2 Securely Evaluating A Single-Instruction GN Query

Recall that in Section 2.3.2, GN queries allow the navigation towards the topology of the graph. Similar to $\mathcal{F}_{\mathsf{GPM}}$, the ideal functionality $\mathcal{F}_{\mathsf{GN}}$ could be divided into three phases:

(1) *Check phase*, where the client checks if a rich-text query is valid. This phase is identical to $\mathcal{F}_{\mathsf{GPM}}$.Check;
(2) *GetID phase*, where the client retrieves IDs for every query string. This phase is identical to $\mathcal{F}_{\mathsf{GPM}}$.GetID;
(3) *Query phase*, which takes a shared traverser as input and outputs a new shared traverser based on the latest traversal instruction;

Query.flatMap. On receiving a flatMap instruction ψ, and shared candidate set $[\![U]\!]$,

(1) Invokes $(S, T, A) \leftarrow \mathsf{Rec}([\![G]\!])$, $U \leftarrow \mathsf{Rec}([\![U]\!])$;
(2) Gets the adjacency list $A_{\mathsf{ind}(u)}$ for every $\mathsf{ind}(u) \in \mathsf{ind}(U)$, finds its neighbor vertex index u' with label defined in instruction ψ, pushes u' into U';
(3) Invokes $[\![U']\!] \leftarrow \mathsf{Shr}(U')$.

Query.filter. On receiving a filter instruction ψ, and shared candidate set $[\![U]\!]$,

(1) Invokes $(S, T, A) \leftarrow \mathsf{Rec}([\![G]\!])$, $U \leftarrow \mathsf{Rec}([\![U]\!])$;
(2) Filters $\mathsf{ind}(U)$ with condition defined by instruction ϕ. Pushes the filtered elements into U';
(3) Invokes $[\![U']\!] \leftarrow \mathsf{Shr}(U')$.

Figure 10: Ideal Functionality $\mathcal{F}_{\mathsf{GN}}$.Query.

Formally, a traverser is defined as $t = ([\![U]\!], \psi)$, where U is an abstract term referring to the location of the traverser, and ψ is a traversal instruction extracted from the rich-text query. Note that we have listed all the possible instructions in Section 2.3.3. Also, since we take different approaches for different instructions, we further divide the functionality of query phase into: *Query.flatMap* and *Query.filter*. We show the ideal functionality of processing a single-instruction GN query in Figure 10.

More precisely, in the secure instantiation of $\mathcal{F}_{\mathsf{GN}}$, the representation of the location indicator U depends on current traversal location. That is to say, if the input traverser $t = (U, \phi)$ locates within a vertex set, U is defined as $U \subseteq \mathsf{ind}(V) = \{v_1, ..., v_*\}$, and we additionally use $A_U = \{A_{v_1}, ..., A_{v_*}\}$ to denote the adjacency list corresponding to U. By such construction of A_U, $\mathcal{F}_{\mathsf{GN}}$ naturally supports mapping from vertex set to edge set. On the other hand, if the traverser locates within an edge set, U is defined as $U = \mathsf{ind}(E) = \{e_1, ..., e_*\}$, and we use $E_U = \{A_{v_1}(1), ..., A_{v_1}(\hat{d}), A_{v_2}(1), ..., A_{v_*}(\hat{d})\}$ to denote the corresponding edge set extracted from A. Note that at the beginning of each traversal, we define the initial location as V_g, where $V_g = \mathsf{ind}(V)$. We present two secure instantiations for $\mathcal{F}_{\mathsf{GN}}$.Query, i.e., $\mathcal{F}_{\mathsf{GN}}$.Query.FlatMap and $\mathcal{F}_{\mathsf{GN}}$.Query.Filter, in Figures 11 and 12, respectively. Since we use standard MPC arithmetic for the protocol, consequently, the protocol is secure under the MPC-hybrid model and the security proof is trivial.

Communication efficiency. The communication cost for those protocols depending on the instruction and the database itself, and it requires $O(|U|)$ equality test operation and additionally $O(|\mathsf{Const}(Q)|)$ communications.

C.Query.flatMap Given a single-instruction path query with condition α, C:

(1) Sends $\mathsf{ind}(U)$, $\psi_{v\to e}$ or $\psi_{e\to v}$ to $\mathcal{S}_i$;
(2) On receiving server's messages, invokes $b_1, ..., b_k \leftarrow \mathsf{Rec}(\langle [\![b_1]\!]_0, ..., [\![b_k]\!]_0, [\![b_1]\!]_1, ..., [\![b_k]\!]_1\rangle)$;

$\mathcal{S}_i$.Query.flatMap

On receiving $\mathsf{ind}(U)$ and an instruction $\psi_{v\to e}$ from C,

(1) Extracts current candidate set $[\![u_j]\!]_i \in [\![U]\!]_i$, and its corresponding shared adjacency list $[\![A_U]\!]_i$;
(2) For all $[\![A_{u_j}]\!]_i \in [\![A_U]\!]_i$, filters and gets the indication vector for every edge in $[\![A_U]\!]_i$, that is $[\![b_1]\!]_i, ..., [\![b_{|U|\hat{d}}]\!]_i \leftarrow \mathsf{EQ}([\![A_{u_j}(*)]\!]_i, \phi_{v\to e})$;
(3) Sends $[\![b_1]\!]_i, ..., [\![b_{|U|\hat{d}}]\!]_i$ to C;

On receiving $\mathsf{ind}(U)$ and an instruction $\psi_{e\to v}$ from C,

(1) Extracts current candidate set $[\![E_{u_j}]\!]_i \in [\![E_U]\!]_i$ which is in the form of (label, tail vertex): $([\![l]\!], [\![t]\!])$;
(2) For all $[\![E_{u_j}]\!]_i \in [\![E_U]\!]_i$, filters and gets the indication vector for every edge in $[\![E_U]\!]_i$, that is $[\![b_1]\!]_i, ..., [\![b_{|U|}]\!]_i \leftarrow \mathsf{EQ}([\![E_{u_j}(*)]\!]_i, \phi_{e\to v})$;
(3) Sends $[\![b_1]\!]_i, ..., [\![b_{|U|}]\!]_i$ to C;

Figure 11: Protocol Π_{GN}.Query.flatMap for functionality $\mathcal{F}_{\mathsf{GN}}$.

C.Query.filter Given a single-instruction path query with condition α,

(1) Sends $\mathsf{ind}(U)$, ψ to $\mathcal{S}_i$;
(2) On receiving server's messages, invokes $b_1, ..., b_{|U|\hat{d}} \leftarrow \mathsf{Rec}(\langle [\![b_1]\!]_0, ..., [\![b_{|U|\hat{d}}]\!]_0, [\![b_1]\!]_1, ..., [\![b_{|U|\hat{d}}]\!]_1\rangle)$;

$\mathcal{S}_i$.Query.filter

On receiving an instruction ψ from C,

(1) Extracts current candidate set $[\![u_j]\!]_i \in [\![U]\!]_i$, and its corresponding shared adjacency list $[\![A_U]\!]_i$;
(2) For all $[\![A_{u_j}]\!]_i \in [\![A_U]\!]_i$, filters and gets the indication vector for every edge in $[\![A_U]\!]_i$, that is $[\![b_1]\!]_i, ..., [\![b_{|U|\hat{d}}]\!]_i \leftarrow \mathsf{EQ}([\![A_{u_j}(*)]\!]_i, \phi_{v\to e})$;
(3) Sends $[\![b_1]\!]_i, ..., [\![b_{|U|\hat{d}}]\!]_i$ to C;

Figure 12: Protocol Π_{GN}.Query.filter for functionality $\mathcal{F}_{\mathsf{GN}}$.

5 SECURE EVALUATION OF A MULTI-INSTRUCTION QUERY

Now that we have introduced how to securely evaluate a single-instruction query in Section 4. It still remains a question: can protocols Π_{GPM} and Π_{GN} preserve efficiency and security under the compositions of single instruction queries?

To answer this question, we first observe that secure Single Instruction Evaluation (SIE) protocols (Π_{GPM}.Query and Π_{GN}.Query) share a common structure. We show the general structure of SIE in Figure 13, where Π.Query takes input of a single instruction (Π_{GPM} or Π_{GN}) and a traversal location set (not required for Π_{GPM}), and outputs a shared indicator vector $[\![\mathbf{b}]\!]$ (when client and server do not invoke reconstruction function). For the case of SIE, $[\![\mathbf{b}]\!]$ is first reconstructed to client, and client then filters the $\mathsf{ind}(G)$ (which equals to $\mathsf{ind}(V) \cup \mathsf{ind}(E) \cup \mathsf{ind}(L)$), and finally gets the query result. However, this interactive approach is not desired when facing a multi-instruction query, since the whole evaluation process will involve $O(|\Psi|)$ interactions between servers and client. Also such naive composition of SIE will have linear communication complexity, which also contradicts the goal (sub-linear) of this work. Moreover, reconstructing $[\![\mathbf{b}]\!]$ to servers inevitably leaks internal traversal locations. Therefore, how to perform filter in an oblivious manner, without reconstructing $[\![\mathbf{b}]\!]$, is the key to achieve sublinear complexity.

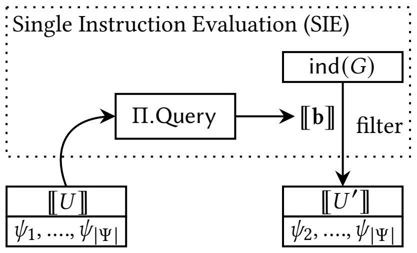

Figure 13: Single Instruction Evaluation (SIE).

In this work, we propose a novel cryptographic primitive called *Oblivious Filter*, which we will present in details in Section 6. We denote the Single Instruction Evaluation with Oblivious Filter as OF-SIE, and show the framework of secure multi-instruction evaluation in Figure 14, where OF-SIE is sequentially executed. Informally, for a traverser $t = ([\![U]\!], \Psi)$ with instruction set $\Psi = \{\psi_1, .., \psi_{|\Psi|}\}$, the evaluation routine works as follows:

$$\text{result} \leftarrow \text{OF-SIE}(\ldots\text{OF-SIE}(\text{OF-SIE}(t = ([\![U]\!], \Psi)))\ldots),$$

where we denote the initial location as $[\![U]\!]$, and result location as $[\![U']\!]$. Since each shared location set for OF-SIE is a strict subset of $\mathsf{ind}(G)$, sub-linear communication complexity is guaranteed.

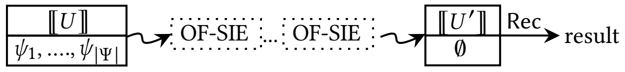

Figure 14: Multi-Instruction Evaluation Framework.

6 OBLIVIOUS FILTER

In this section, we present the building block for our proposed general query processing system, i.e., oblivious filter.

Definition. We define the oblivious filter primitive as a 2-party functionality between a server and a client. A server holds a list of pairs $(\mathbf{t}, \mathbf{v})$, and every pair consists of an indication bit $t_i \in \{0, 1\}$ and a fixed finite value $v_i \in \mathcal{V}$. The client holds a choice bit $c \in \{0, 1\}$. Intuitively, oblivious filter performs a secure filtering on the list of pairs $(\mathbf{t}, \mathbf{v})$, and outputs v_i in secret sharing format if v_i's indication bit equals to the choice bit, namely $t_i = c$. The size of output is denoted as n (the notations in this section are independent from other sections). We describe this functionality in Figure 15.

Inputs. A list of m pairs $(\mathbf{t}, \mathbf{v})$ from server, where $\mathbf{t} \in \{0, 1\}^m$, $\mathbf{v} \in \mathcal{V}^m$; A choice bit vector $\mathbf{c} \in \{0, 1\}^m$ from client.
Outputs (shared). $[\![\mathbf{v}']\!]$, s.t. $\mathbf{v}' \subseteq \mathbf{v}$, and for every element $v' \in \mathbf{v}'$, its corresponding t' equals to c.

Figure 15: Functionality of Oblivious Filter.

Relations with other primitives. Oblivious filter can be seen as a instantiation of Private Function Evaluation (PFE) [32, 35] with shared outputs. Moreover, one building block for PFE — Oblivious Extended Permutation (OEP) [40] could be seen as a relaxation of oblivious filter, where the filter function is held by client instead of both parties. Roughly, OEP assumes that server holds an extended permutation $\pi : \{1, .., m\} \to \{1, ..., n\}$, a mask $\mathbf{r} \in \mathcal{V}^n$, while client holds a private input $\mathbf{v} \in \mathcal{V}^m$. At the end of OEP protocol, client learns $\{v_{\pi^{-1}(1)} + r_1, ..., v_{\pi^{-1}(n)} + r_n\}$, and server holds $\{r_1, ..., r_n\}$. Though the design intuition is similar, oblivious filter takes a simpler construction from homomorphic encryption (other than universal circuits), which makes it more communication-efficient than the OEP protocols.

Additionally, oblivious filter can also be seen as a simpler version of Private Secret-Shared Set Intersection (PS^{3}I) [10] and therefore has a more efficient construction comparing with PSI, which assumes two parties hold a list $(\mathbf{t}_c, \mathbf{v}_c)$ and $(\mathbf{t}_s, \mathbf{v}_s)$ respectively, and the protocol finally outputs the shared intersection of $\mathbf{v}'_\mathbf{c}$ and $\mathbf{v}'_\mathbf{s}$. Notice that for all $v'_c \in \mathbf{v}'_c \subseteq \mathbf{v}_c$ and $v'_s \in \mathbf{v}'_c \subseteq \mathbf{v}_s$, $t_{c,i} = t_{s,i}$ holds. PS^{3}I ensures that at the end of the protocol, two parties only learn the shares the intersection and nothing else.

Our work is also related to other primitives, such as Oblivious Data Structures [51] and Oblivious Shuffling [15, 35, 41, 46]. In the literature, there are three approaches for PFE protocol construction [35]: homomorphic encryption, universal circuit, and oblivious switching network. We follow such design intuition and try to build oblivious filter based on HE.

Security of oblivious filter. Roughly, we define that an instantiation of oblivious filter should follow simulation-based security with the presence of a semi-honest adversary. That is, at the end of the protocol, server should know nothing about c and client should know nothing about $(\mathbf{t}, \mathbf{v})$. Formally, security holds that any party's view during the attack can be simulated given only its own input and output.

Definition 6.1 (semi-honest security). Let Π_{OF} be an oblivious filter protocol. We say that Π_{OF} is secure against semi-honest adversary in the two-party settings if for all adversaries $\mathcal{A}$, there exists a polynomial-time simulator $\mathcal{S}$, such that

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

In the following of this section, we will present a construction of oblivious filter using HE, and then introduce a variant of oblivious filter, namely secret-shared oblivious filter.

6.1 Construction of OF from HE

We first present a simple construction of oblivious filter based on partially homomorphic encryption scheme. We present the description of this protocol (OF-HE) in Figure 16. The main idea is that server first performs a secure evaluation to shuffled indicator vector, without knowing additional information about $\mathbf{t}$.

During the setup phase, both parties generate encryption key pairs by $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)$, and exchange public key with each other. At the beginning of the protocol, client first uses its public key to encrypt its choice vector $\mathbf{c}$ and sends it to server. Then server performs secure evaluation between the encrypted $\mathbf{c}$ and $\mathbf{t}$, and gets the encrypted $\mathbf{c} - \mathbf{t}$. Afterwards, server generates a random permutation $\pi_s \in S^m$, and encrypts and shuffles the value $\mathbf{v}$. After it, server sends it to client $(\pi_s(\mathsf{Enc}_{\mathsf{pk_c}}(\mathbf{c} - \mathbf{t})), \pi_s(\mathsf{Enc}_{\mathsf{pk_s}}(\mathbf{v})))$.

Then client filters $\mathsf{Enc}_{\mathsf{pk}}(\pi_s(\mathbf{v})))$ by decrypting the first part of server's message $\pi_s(\mathsf{Enc}_{\mathsf{pk_c}}(\mathbf{c} - \mathbf{t}))$. As a result, client gets a new vector $\mathsf{Enc}_{\mathsf{pk_s}}(\pi_s(\mathbf{v}')))$. Then client samples a new vector r and gets $\mathsf{Enc}_{\mathsf{pk_s}}(\mathbf{v}' - \mathbf{r}))$. Next, client applies a new permutation $\pi_c \in S^n$ to the encrypted result and sends $\mathsf{Enc}_{\mathsf{pk_s}}(\pi_c(\mathbf{v}' - \mathbf{r})))$ back to server. After decryption, server sets $[\![\mathbf{v}']\!]_0 = \mathbf{v}' - \mathbf{r}$ and client sets $[\![\mathbf{v}']\!]_1 = \mathbf{r}$.

Lemma 6.2. *The protocol described in Figure 16 is a secure instantiation for oblivious filter functionality.*

The correctness of the simple construction could be easily verified. As for the security of this construction, we prove the security of OF-HE in Appendix B.

Now we analyse the complexity of this protocol.

- Round complexity: Constant-round and communicating $m+n$ ciphers in total.
- Computation complexity: Server performs $m + n$ public-key operations and Client performs n times HE evaluations.

6.2 Secret-Shared Oblivious Filter

Since we use a shared input and shared output format in our querying framework, in this subsection, we present a variant of OF, i.e., Secret Shared Oblivious Filter (SS-OF) input. In this variant of OF, we assume that server and client both hold the shares of a list $(\mathbf{t}, \mathbf{v})$, and they both know a public choice bit $c \in \{0, 1\}$. We formally describe this functionality in Figure 17. In the following, we will present two different construction of SS-OF.

6.2.1 Construction of SS-OF from Oblivious Filter. SS-OF functionality can be achieved from a two-fold oblivious filter (see Figure 18). For the first round, P_0 inputs $[\![\mathbf{t}]\!]_0$ and $[\![\mathbf{v}]\!]_0$ to Π_{OF}, and P_1 inputs $[\![\mathbf{t}]\!]_1$. Then Π_{OF} outputs $[\![[\![\mathbf{v}']\!]_0]\!]_0$ to P_0, and $[\![[\![\mathbf{v}']\!]_0]\!]_1$ to P_1. For the second round, P_0 inputs $[\![\mathbf{v}]\!]_0$ to Π_{OF}, and P_1 inputs $[\![\mathbf{t}]\!]_1$ and $[\![\mathbf{v}]\!]_1$, then Π_{OF} outputs $[\![[\![\mathbf{v}']\!]_1]\!]_0$ to P_0, and $[\![[\![\mathbf{v}']\!]_1]\!]_1$ to P_1. Finally, $[\![[\![\mathbf{v}']\!]_0]\!]_0 + [\![[\![\mathbf{v}']\!]_1]\!]_0$ and $[\![[\![\mathbf{v}']\!]_0]\!]_1 + [\![[\![\mathbf{v}']\!]_1]\!]_1$ are the reshares of the result $\mathbf{v}'$.

Inputs. A list of m pairs $(\mathbf{t}, \mathbf{v})$ from server, where $\mathbf{t} \in \{0, 1\}^m, \mathbf{v} \in \mathcal{V}^m$; A choice bit vector $\mathbf{c} \in \{0, 1\}^m$ from client.
Outputs (shared). $[\![\mathbf{v}']\!]$, s.t. $\mathbf{v}' \subseteq \mathbf{v}$, and for every element $v' \in \mathbf{v}'$, its corresponding t' equals to c.
Setup. Server and client both run $\mathsf{Gen}(1^\lambda)$, and sends their public key to the other party.
Protocol.

1. Client uses its own public key $\mathsf{pk_c}$ to encrypt the choice vector $\mathbf{c}$, then sends $\mathsf{Enc}_{\mathsf{pk_c}}(\mathbf{c})$ to server.
2. Server generates a random permutation $\pi_s \in S^m$, and gets $\mathsf{Enc}_{\mathsf{pk_c}}(\mathbf{c} - \mathbf{t}) \leftarrow \mathsf{Eval}(\mathsf{Enc}_{\mathsf{pk_c}}(\mathbf{c}), \mathsf{Enc}_{\mathsf{pk_c}}(\mathbf{t}))$, shuffles the result $(\pi_s(\mathsf{Enc}_{\mathsf{pk_c}}(\mathbf{c} - \mathbf{t})), \mathsf{Enc}_{\mathsf{pk_s}}(\pi_s(\mathbf{v})))$ and sends to client.
3. Client performs decryption on server's first message, and gets $\pi_s(\mathbf{c} - \mathbf{t})$. Then client filters second message based on $\pi_s(\mathbf{c} - \mathbf{t})$:
   (a) Initiates an empty set: $\mathbf{v}'_s = \emptyset$.
   (b) For every item $\mathsf{Enc}_{\mathsf{pk}_s}(v_{\pi_s(i)})$ in $\pi_s(\mathsf{Enc}_{,\mathsf{pk}_s}(\mathbf{v}))$, if $(c_{\pi_s} - t_{\pi_s}) = 0$, push $\mathsf{Enc}_{\mathsf{pk}}(v_{\pi_s(i)})$ into $\mathbf{v}'_s$.
   (c) The final size of $\mathbf{v}'_s$ is denoted as n.
4. Client randomly samples a size-n vector $\mathbf{r} \xleftarrow{\$} \mathcal{V}^n$ and a random permutation $\pi_c \in S^n$. Then client evaluates $\mathsf{Enc}_{\mathsf{pk_s}}(v'_i - r_i) \leftarrow \mathsf{Eval}(\mathsf{Enc}_{\mathsf{pk_s}}(v'_i), \mathsf{Enc}_{\mathsf{pk_s}}(r_i))$ and updates $\mathbf{v}'_s$ by $\mathsf{Enc}_{\mathsf{pk_s}}(\mathbf{v}' - \mathbf{r})$. Finally, client applies the permutation and sends back $\pi_c(\mathsf{Enc}_{\mathsf{pk_s}}(\mathbf{v}' - \mathbf{r}))$ to server .
5. Server decrypts the message and gets $[\![\mathbf{v}']\!]_0 = \pi_c(\mathbf{v}' - \mathbf{r})$. Client gets $[\![\mathbf{v}']\!]_1 \leftarrow \pi_c(\mathbf{r})$.

Figure 16: Instantiation of OF from Additive HE (OF-HE).

Inputs (shared). A list of m pairs of shares $([\![\mathbf{t}]\!], [\![\mathbf{v}]\!])$, where $\mathbf{t} \in \{0, 1\}^m, \mathbf{v} \in \mathcal{V}^m$. A public choice bit $c \in \{0, 1\}$
Outputs (shared). $[\![\mathbf{v}']\!]$, s.t. for all $v' \in \mathbf{v}'$, its corresponding t' equals to c.

Figure 17: Functionality of SS-OF.

6.2.2 Construction of SS-OF from Oblivious Shuffle. We borrow the idea from [41], and propose a generic construction of SS-OF from Oblivious Shuffle in Figure 19. The communication and computation complexities of this construction mainly depend on the oblivious shuffle protocol, and to the best of our knowledge, the most efficient secret-shared shuffle protocol [15] achieves $O(N \log N \cdot \lambda)$ communication, where N is the set size and λ is the security parameter. With an efficient secret-shared shuffle protocol such as [15], constructing SS-OF from oblivious shuffle is also efficient.

7 EXPERIMENT

We implement the additive homomorphic encryption based oblivious filter using Paillier encryption scheme, which is implemented using C++ with GMP library. Our test environment is Quad-Core

Table 1: Running time of OF-HE by filtering 50% of client's input data.

Key size		**1024 bit**				**2048 bit**			
Input data size		100	1,000	10,000	100,000	100	1,000	10,000	100,000
Offline time	Server	0.366872	3.73717	40.4863	400.302	1.83838	18.9394	187.717	1953.53
	Client	0.373998	3.74880	40.3731	399.779	1.84555	18.9853	187.553	1950.47
Online time	Server	0.062351	0.647129	7.31345	67.1577	0.363488	4.07819	44.9030	399.079
	Client	0.131378	1.28905	14.3267	127.836	0.831858	8.41821	96.0366	821.004

Inputs (shared). A list of m pairs of shares $(\llbracket \mathbf{t} \rrbracket, \llbracket \mathbf{v} \rrbracket)$, where $\mathbf{t} \in \{0,1\}^m$, $\mathbf{v} \in \mathcal{V}^m$; A public choice bit $c \in \{0,1\}$.
Outputs (shared). $\llbracket \mathbf{v}' \rrbracket$, s.t. for all $v' \in \mathbf{v}'$, its corresponding t' equals to c.
Setup. P_0 runs $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)$, and sends pk to P_1.
Protocol.

(1) P_0 and P_1 performs a OF protocol, where P_0 inputs $\llbracket \mathbf{t} \rrbracket_0$ and $\llbracket \mathbf{v} \rrbracket_0$, P_1 inputs $\llbracket \mathbf{t} \rrbracket_1$.
(2) At the end, P_0 receives share $\llbracket \llbracket \mathbf{v}' \rrbracket_0 \rrbracket_0$, and P_1 receives share $\llbracket \llbracket \mathbf{v}' \rrbracket_0 \rrbracket_1$.
(3) Switch roles of P_0 and P_1, where P_0 inputs $\llbracket \mathbf{t} \rrbracket_1$ and $\llbracket \mathbf{v} \rrbracket_1$.
(4) At the end, P_0 receives share $\llbracket \llbracket \mathbf{v}' \rrbracket_1 \rrbracket_0$, and P_1 receives share $\llbracket \llbracket \mathbf{v}' \rrbracket_1 \rrbracket_1$.
(5) P_0 calculates the reshared result as $\llbracket \hat{\mathbf{v}}' \rrbracket_0 = \llbracket \llbracket \mathbf{v}' \rrbracket_0 \rrbracket_0 + \llbracket \llbracket \mathbf{v}' \rrbracket_1 \rrbracket_0$, and P_1 calculates the reshared result as $\llbracket \hat{\mathbf{v}}' \rrbracket_1 = \llbracket \llbracket \mathbf{v}' \rrbracket_0 \rrbracket_1 + \llbracket \llbracket \mathbf{v}' \rrbracket_1 \rrbracket_1$.

Figure 18: Constructing SS-OF from OF.

(1) Both parties apply a *secret-shared shuffling protocol* to permute the original data $(\llbracket \pi(\mathbf{t}) \rrbracket, \llbracket \pi(\mathbf{v}) \rrbracket) \leftarrow \mathsf{Permute}(\llbracket \pi \rrbracket, (\llbracket \mathbf{t} \rrbracket, \llbracket \mathbf{v} \rrbracket))$, where the i-th permuted element is denoted as $(\llbracket t_{\pi(i)} \rrbracket, \llbracket v_{\pi(i)} \rrbracket)$.
(2) Both parties reconstruct the permuted vector $\llbracket \pi(\mathbf{t}) \rrbracket$ of the resulting database $(\llbracket \pi(\mathbf{t}) \rrbracket, \llbracket \pi(\mathbf{v}) \rrbracket)$,
(3) Both parties keep the shares of $\llbracket \mathbf{v}_{\pi(i)} \rrbracket$ for which the corresponding indicator bit $\mathbf{t}_{\pi(i)} = c$.

Figure 19: Constructing SS-OF from oblivious shuffle.

Intel Core i5 2.40GHz CPU with 16G RAM, and we have run the test on different size of databases in LAN. Specifically in our experiment, our protocol can finish oblivious filter on 10^5 data within about 30 minutes. We report the experimental result of OF using HE (OF-HE) in Table 1, where we set the modulus to 1,024 bit and 2,048 bit, respectively. From them, we can find that the evaluation time of OF are linear with data size on both server and client's sides, which indicates its scalability.

8 CONCLUSION

In this paper, we focus on the problem of how to perform scalable and secure query on secret shared graph databases. To do this, we first summarized the queries on secret sharing graph database into two single instruction queries, i.e., Graph Pattern Matching (GPM) and Graph Navigation (GN), and a multi-instruction that is composed by single instruction queries. We then leveraged secure multiparty computation technique to securely evaluate GPM and GN. Next, we proposed a general framework for processing multi-instruction query and introduced a novel cryptographic primitive Oblivious Filter (OF) as a core building block. We constructed OF with homomorphic encryption and proved that our proposed framework has sub-linear complexity and is resilient to access-pattern attacks. Finally, empirical study demonstrated the efficiency of our proposed OF protocol.

REFERENCES

[1] [n.d.]. *Crypten: A research tool for secure machine learning in PyTorch.* https://crypten.ai/
[2] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases.* Vol. 8. Addison-Wesley Reading.
[3] Saleh Ahmed, Annisa, Asif Zaman, Zhan Zhang, Kazi Md. Rokibul Alam, and Yasuhiko Morimoto. 2019. Semi-Order Preserving Encryption Technique for Numeric Database. *Int. J. Netw. Comput.* 9 (2019), 111–129.
[4] R. Angles, M. Arenas, P. Barceló, A. Hogan, Juan L. Reutter, and D. Vrgoc. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Computing Surveys (CSUR)* 50 (2017), 1 – 40.
[5] R. Angles, H. Thakkar, and Dominik Tomaszuk. 2019. RDF and Property Graphs Interoperability: Status and Issues. In *AMW*.
[6] D. W. Archer, D. Bogdanov, Yehuda Lindell, L. Kamm, Kurt Nielsen, J. Pagter, N. Smart, and R. Wright. 2018. From Keys to Databases - Real-World Applications of Secure Multi-Party Computation. *IACR Cryptol. ePrint Arch.* 2018 (2018), 450.
[7] P. Barceló. 2013. Querying graph databases. In *PODS '13*.
[8] Maciej Besta, Emanuel Peter, Robert Gerstenberger, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2019. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. arXiv:1910.09017 [cs.DB]
[9] D. Brickley and R. V. Guha. 2002. Resource Description Framework (RDF) Model and Syntax Specification.
[10] Prasad Buddhavarapu, A. Knox, Payman Mohassel, S. Sengupta, Erik Taubeneck, and Vlad Vlaskin. 2020. Private Matching for Compute. *IACR Cryptol. ePrint Arch.* 2020 (2020), 599.
[11] Diego Calvanese, Giuseppe De Giacomo, M. Lenzerini, and M. Vardi. 2003. Reasoning on regular path queries. *SIGMOD Rec.* 32 (2003), 83–92.
[12] R. Canetti. 2000. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology* 13 (2000), 143–202.
[13] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. 2013. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. *IACR Cryptol. ePrint Arch.* 2013 (2013), 169.
[14] Octavian Catrina and S. D. Hoogh. 2010. Improved Primitives for Secure Multiparty Integer Computation. In *SCN*.
[15] M. Chase, E. Ghosh, and Oxana Poburinnaya. 2019. Secret Shared Shuffle. *IACR Cryptol. ePrint Arch.* 2019 (2019), 1340.
[16] Jiefeng Cheng, J. Yu, B. Ding, P. Yu, and H. Wang. 2008. Fast Graph Pattern Matching. *2008 IEEE 24th International Conference on Data Engineering* (2008), 913–922.
[17] Tung Chou and C. Orlandi. 2015. The Simplest Protocol for Oblivious Transfer. In *IACR Cryptol. ePrint Arch.*
[18] Henry Corrigan-Gibbs and D. Kogan. 2019. Private Information Retrieval with Sublinear Online Time. *IACR Cryptol. ePrint Arch.* 2019 (2019), 1075.
[19] R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing. 2018. SPDZ2k: Efficient MPC mod 2k for Dishonest Majority. In *IACR Cryptol. ePrint Arch.*
[20] M. Dahl, J. Mancuso, Yann Dupis, Ben Decoste, M. Giraud, I. Livingstone, Justin Patriquin, and Gavin Uhma. 2018. Private Machine Learning in TensorFlow using Secure Computation. *ArXiv* abs/1810.08130 (2018).
[21] I. Damgård, D. Escudero, T. Frederiksen, Marcel Keller, P. Scholl, and Nikolaj Volgushev. 2019. New Primitives for Actively-Secure MPC over Rings with Applications to Private Machine Learning. *2019 IEEE Symposium on Security and*

Privacy (SP) (2019), 1102–1120.
[22] I. Damgård and M. Jurik. 2001. A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System. In *Public Key Cryptography.*
[23] I. Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, P. Scholl, and N. Smart. 2013. Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits. In *ESORICS.*
[24] I. Damgård, Valerio Pastro, N. Smart, and Sarah Zakarias. 2011. Multiparty Computation from Somewhat Homomorphic Encryption. *IACR Cryptol. ePrint Arch.* 2011 (2011), 535.
[25] Nadime Francis, Alastair Green, P. Guagliardo, L. Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, M. Rydberg, P. Selmer, and A. Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. *Proceedings of the 2018 International Conference on Management of Data* (2018).
[26] Benny Fuhry, Jayanth Jain H A, and Florian Kerschbaum. 2020. EncDBDB: Searchable Encrypted, Fast, Compressed, In-Memory Database using Enclaves. arXiv:2002.05097 [cs.CR]
[27] C. Gentry and Zulfikar Ramzan. 2005. Single-Database Private Information Retrieval with Constant Communication Rate. In *ICALP.*
[28] Oded Goldreich. 2000. Foundations of Cryptography: Basic Tools.
[29] Oded Goldreich, S. Micali, and A. Wigderson. 2019. How to play any mental game, or a completeness theorem for protocols with honest majority. In *Providing Sound Foundations for Cryptography.*
[30] S. Goldwasser and S. Micali. 2019. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Providing Sound Foundations for Cryptography.*
[31] P. Hell and J. Nesetril. 2004. Graphs and homomorphisms. In *Oxford lecture series in mathematics and its applications.*
[32] Marco Holz, Ágnes Kiss, Deevashwer Rathee, and Thomas Schneider. 2020. Linear-Complexity Private Function Evaluation is Practical. In *IACR Cryptol. ePrint Arch.*
[33] E. P. hommeaux. 2011. SPARQL query language for RDF.
[34] W. Horner. [n.d.]. A new method of solving numerical equations of all orders, by continuous approximation. *Philosophical Transactions of the Royal Society* 2 ([n. d.]), 117–117.
[35] Y. Huang, David Evans, and Jonathan Katz. 2012. Private Set Intersection: Are Garbled Circuits Better than Custom Protocols?. In *NDSS.*
[36] M. S. Islam, M. Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *NDSS.*
[37] Marcel Keller, Valerio Pastro, and Dragos Rotaru. 2017. Overdrive: Making SPDZ Great Again. *IACR Cryptol. ePrint Arch.* 2017 (2017), 1230.
[38] V. Kolesnikov, R. Kumaresan, M. Rosulek, and Ni Trieu. 2016. Efficient Batched Oblivious PRF with Applications to Private Set Intersection. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016).
[39] Shangqi Lai, Xingliang Yuan, Shifeng Sun, J. K. Liu, Yuhong Liu, and D. Liu. 2019. GraphSE²: An Encrypted Graph Database for Privacy-Preserving Social Search. *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security* (2019).
[40] Peeter Laud and J. Willemson. 2014. Composable Oblivious Extended Permutations. *IACR Cryptol. ePrint Arch.* 2014 (2014), 400.
[41] S. Laur, J. Willemson, and B. Zhang. 2011. Round-Efficient Oblivious Database Manipulation. *IACR Cryptol. ePrint Arch.* 2011 (2011), 429.
[42] Ricardo Macedo, J. Paulo, Rogério Pontes, Bernardo Portela, Tiago Oliveira, M. Matos, and R. Oliveira. 2017. A Practical Framework for Privacy-Preserving NoSQL Databases. *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)* (2017), 11–20.
[43] C Martínez and G Valiente. 1997. An algorithm for graph pattern-matching. In *Proc. Fourth South American Workshop on String Processing*, Vol. 8. 180–197.
[44] S. Mehrotra, S. Sharma, J. Ullman, D. Ghosh, and Peeyush Gupta. 2020. Panda: Partitioned Data Security on Outsourced Sensitive and Non-sensitive Data. *ACM Transactions on Management Information Systems (TMIS)* (2020).
[45] Payman Mohassel, Peter Rindal, and M. Rosulek. 2019. Fast Database Joins for Secret Shared Data. *IACR Cryptol. ePrint Arch.* 2019 (2019), 518.
[46] Payman Mohassel and Seyed Saeed Sadeghian. 2013. How to Hide Circuits in MPC: An Efficient Framework for Private Function Evaluation. *IACR Cryptol. ePrint Arch.* 2013 (2013), 137.
[47] T. Okamoto and S. Uchiyama. 1998. A New Public-Key Cryptosystem as Secure as Factoring. In *EUROCRYPT.*
[48] Pascal Paillier. 1999. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *EUROCRYPT.*
[49] Michael A. Rodriguez. 2015. The Gremlin graph traversal machine and language (invited talk). In *DBPL 2015.*
[50] Bharath K. Samanthula, Wei Jiang, and E. Bertino. 2014. Privacy-Preserving Complex Query Evaluation over Semantically Secure Encrypted Data. In *ESORICS.*
[51] Phillipp Schoppmann, A. Gascón, Mariana Raykova, and Benny Pinkas. 2019. Make Some ROOM for the Zeros: Data Sparsity in Secure Distributed Machine Learning. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (2019).
[52] A. Shamir. 1979. How to share a secret. *Commun. ACM* 22 (1979), 612–613.
[53] D. Song, D. Wagner, and A. Perrig. 2000. Practical techniques for searches on encrypted data. *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000* (2000), 44–55.
[54] H. Stoß. 1985. The Complexity of Evaluating Interpolation Polynomials. *Theor. Comput. Sci.* 41 (1985), 319–323.
[55] Stephen Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. 2013. Processing Analytical Queries over Encrypted Data. *Proc. VLDB Endow.* 6 (2013), 289–300.
[56] J. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23 (1976), 31–42.
[57] P. T. Wood. 2012. Query languages for graph databases. *SIGMOD Rec.* 41 (2012), 50–60.
[58] Z. Wu and K. Li. 2018. VBTree: forward secure conjunctive queries over encrypted data for cloud computing. *The VLDB Journal* 28 (2018), 25–46.
[59] Zhiqiang Yang, S. Zhong, and R. Wright. 2006. Privacy-Preserving Queries on Encrypted Data. In *ESORICS.*

A PAILLIER ENCRYPTION

We introduce the paillier encryption scheme [48] which is used to construct oblivious filter. Paillier is an instantiation of additive homomorphic encryption with algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$. We introduce the details of these algorithms below.

Key Generation. $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)$: Taking a security parameter λ as input, generates the public key pk for encryption, and secret key sk for decryption.

- First, generate two large prime numbers with bit size equal to the security parameter $q, p \leftarrow$ large prime, and ensure that $\gcd(pq, (p-1)(q-1)) = 1$.
- Let $n = pq$ and calculate $\lambda = \mathrm{lcm}(p-1, q-1)$.
- Randomly select a group generator for n^2, $g \xleftarrow{\$} \mathbb{Z}^*_{n^2}$, and ensure that n divides the order of g, if not, repeat this step.
- Calculate $\mu = (L(g^\lambda \mod n^2))^{-1} \mod n$, where $L(x) = (x-1)/n$.
- Let $\mathsf{pk} = (n, g)$ and $\mathsf{sk} = (\lambda, \mu)$.

Encryption. $c \leftarrow \mathsf{Enc}_{\mathsf{pk}}(m)$: Taking a message m and public key pk as input, return the encrypted cipher c.

- Randomly select a number $r \xleftarrow{\$} \mathbb{Z}^*_n$.
- Generate the ciphertext $c = g^m r^n \mod n^2$.

Decryption. $m \leftarrow \mathsf{Dec}_{\mathsf{sk}}(c)$: Taking a ciphertext c and secret key sk as input, return the decrypted message m.

- $m = c^\lambda \mu \mod n.$

Evaluation. $c = \mathsf{Eval}_{\mathsf{pk}}(c_0, c_1)$:

- Calculate $c = c_0 \cdot c_1 \mod n^2$.

Here we do not illustrate the details of the correctness and the security properties of paillier, for those who are interested we refer to the original work [48]. Other additive homomorphic encryption schemes can be found in [22, 47].

B SECURITY PROOF OF LEMMA 6.2

In this part, we prove the security of Lemma 6.2 using real world vs. ideal world simulation-based technique [12, 28]. First, recall that

Lemma 6.2. The protocol described in Figure 16 is a secure instantiation for oblivious filter functionality against adaptive semi-honest adversary assuming the existence of a IND-CCA2 secure additive homomorphic encryption scheme.

To start the proof, we first introduce the real/ideal models for the construction of oblivious filter functionality from HE, which we denote as Π_{HE}. Since OF is a two-party functionality between a server and a client, where the server holds the private list $(\mathbf{t}, \mathbf{v})$, and the client holds the private choice vector $\mathbf{c}$, where $\mathbf{t}, \mathbf{c} \in \{0,1\}^n$, and $\mathbf{v} \in \mathcal{V}^n$. We denote $\mathcal{V}$ as the value space.

Definition B.1. Let $f = (f_s, f_c)$ be two functionality, we say a protocol Π securely computes f in the presence of static semi-honest adversaries if there exists a probabilistic polynomial-time algorithm S_s and S_c such that

$$\{(S_s(1^\lambda, \mathbf{t}, \mathbf{v}, [\![\mathbf{v}']\!]_s), \mathbf{v})\} \overset{c}{\equiv} \{(\mathsf{view}_s^\Pi(\lambda, \mathbf{t}, \mathbf{c}), \mathsf{output}^\Pi(\lambda, \mathbf{t}, ,\mathbf{c}))\},$$

$$\{(S_c(1^\lambda, \mathbf{c}, [\![\mathbf{v}']\!]_c), \mathbf{v})\} \overset{c}{\equiv} \{(\mathsf{view}_c^\Pi(\lambda, \mathbf{v}), \mathsf{output}^\Pi(\lambda, \mathbf{t}, ,\mathbf{c}))\}.$$

IND-CCA2 Security. Indistinguishability under adaptive Chosen Ciphertext Attack (IND-CCA2) is defined as an adversary game, where the adversary is given access to both encryption and decryption oracle. The game is defined as follows:

Simulator S_s. To simulate the server in OF, extract the output size n from the ideal output $[\![\mathbf{v}']\!]_s$. Then uniformly sample a boolean list $\mathbf{c} \xleftarrow{\$} \{0,1\}^m$, a random permutation π_s, securely evaluate and send $\pi_s(\mathsf{Enc}_{\mathsf{pk_c}}(\mathbf{c} - \mathbf{t}))$ and $\pi_s(\mathsf{Enc}_{\mathsf{pk_s}}(\mathbf{v}))$ to the client. This simulation is computationally indistinguishable from the real execution assuming the encryption scheme is IND-CCA2 secure. To this end, the simulation is finished since it only receives the filter result afterwards.

Simulator S_c. To simulate the client in OF, first send $\mathsf{Enc}_{\mathsf{pk_c}}(\mathbf{c})$ to the server. Afterwards, extract the output size n from the ideal output $[\![\mathbf{v}']\!]_c$ and uniformly sample a list $\mathbf{r} \xleftarrow{\$} \mathcal{V}^n$, and send to the client. This simulation is computationally indistinguishable from the real execution assuming the encryption scheme is IND-CCA2 secure.