Asymmetric Structure-Preserving Subgraph Queries for Large Graphs

Zhe Fan1  Byron Choi1  Jianliang Xu1  Sourav S Bhowmick2
1Department of Computer Science, Hong Kong Baptist University, Hong Kong, China
{zfan, bchoi, xujl}@comp.hkbu.edu.hk
2School of Computer Engineering, Nanyang Technological University, Singapore
assourav@ntu.edu.sg

Abstract—One fundamental type of query for graph databases is subgraph isomorphism queries (a.k.a subgraph queries). Due to the computational hardness of subgraph queries coupled with the cost of managing massive graph data, outsourcing the query computation to a third-party service provider has been an economical and scalable approach. However, confidentiality is known to be an important attribute of Quality of Service (QoS) in Query as a Service (QaaS). In this paper, we propose the first practical private approach for subgraph query services, asymmetric structure-preserving subgraph query processing, where the data graph is publicly known and the query structure/topology is kept secret. Unlike other previous methods for subgraph queries, this paper proposes a series of novel optimizations that only exploit graph structures, not the queries. Further, we propose a robust query encoding and adopt the novel cyclic group based encryption so that query processing is transformed into a series of private matrix operations. Our experiments confirm that our techniques are efficient and the optimizations are effective.

I. INTRODUCTION

Subgraph queries (via subgraph isomorphism) are a fundamental and powerful query in various real graph applications. In particular, it is fundamental to various modern graph queries, such as graph pattern queries [1] and ontology-based matching [28]. While it is well known that subgraph queries are NP-hard, there has been significant research progress on enhancing their performance, e.g., [7], [14], [25], [29]. A recent attempt has been to outsource costly computation to a query service provider (SP), who is often equipped with powerful machines, to provide query as a service (QaaS). Thereby, users not only obtain high performance, scalability, and elasticity [13] but also are free from the burdens of managing IT infrastructure.

Because SPs may not always be trusted, users’ privacy may be threatened. In fact, (data or query) confidentiality has been recognized as one of the public’s most crucial concerns (e.g., [23]). A stream of research on private query processing has bloomed in the past decade, e.g., in the context of relational databases [16], spatial databases [17] and graphs [3]. However, to date, subgraph queries that preserve the query structure (a.k.a topology) over large networks has not yet been studied. We motivate the problem with an application scenario, which does not require domain knowledge. Others can be found in network medicine [1] and patterns in communication networks.

Example 1.1: Law enforcement agencies are increasingly using social media to solve crimes. According to a recent survey of 1,221 federal, state and local law enforcement who use social media, four out of five officials used social networking websites mainly popular in Iran) held in a third party SP. In order to monitor the online activities of these individuals with one another, the agency wishes to glean information related to interactions between them on the network by issuing a subgraph query representing the relationships between the individuals. Unfortunately, it is possible that the SP may have been infiltrated by friends or sympathisers of these individuals. Hence, in order to protect the privacy of the intent of the agency from the SP, the agency cannot expose the subgraph query directly, especially the query structure (i.e., specific relationship pattern between the individuals). How can the agency glean relevant information using a subgraph query while preserving its topological privacy?

Unfortunately, previous work on privacy-preserving graph queries [2], [3], [12], [15], [19], [21], [24] (except [8]) cannot support subgraph queries while preserving their structure. Fan et al. [8] keep both query and data graphs private. In contrast, as query clients may often have data access privileges, the privacy requirement of this work is on queries only. As a result, querying significantly larger graphs becomes possible. Other work has studied privacy-preserving graph publication [4], [5], [22], [32], [33]. Since the published data are sanitized (i.e., modified), it is not clear how subgraph queries can be supported. Recent studies [10], [11] have addressed the authenticity of the query answers, but not their confidentiality.

It is worth highlighting that the intrinsic technical challenge of this research direction is that although the data graph is available to the SP, the SP cannot optimize the queries by directly exploiting the structure of the query graphs. In comparison, recent subgraph isomorphism algorithms (e.g., VF2 [7], Turboiso [14] and QuickSI [25]) intensively utilize the query graphs for optimization, which by definition, leaks their structural information. More recently, the work reported in [20] supports “structureless” graph queries. However, the

structure is automatically generated by a ranking model and the SP is aware of the queries.

One may also attempt to solve the problem with a naive solution in which the SP exhaustively traverses all of the data graph to enumerate all candidate mappings (i.e., possible mappings) between the query and the graph and return them to the client for verification. The intuition is that since the query structure is not exploited, its privacy is preserved. However, this is infeasible because the number of candidate mappings is exponential to the graph size in the worst case.

The first challenge of this research is then “how to reduce a large data graph and subsequently the number of candidate mappings for verification, without exposing the query structure?” Our first idea is to determine the minimized candidate subgraphs that contain at least a candidate mapping. Then candidate mappings are enumerated from those subgraphs instead of the original graph. In particular, we propose optimizations that use novel neighborhood containment of data vertices to minimize the subgraphs. Second, we determine subgraphs (called candidate matchings) from a candidate subgraph, where candidate mappings are enumerated. In comparison, in previous work [7], [14], [25] where privacy is not a concern, the intuition is that since the query graph is known. We propose a subgraph cache and use neighborhood equivalent classes to further minimize the number of matchings and mappings.

The second challenge is “how to verify if a candidate mapping is a subgraph isomorphism mapping without leaking the query structure?”. We propose a query encoding scheme and adopt an encryption scheme for query graphs. With these, we derive a basic structure-preserving verification method that consists of a series of private matrix operations. Moreover, to minimize communication overheads, we propose to use the complement of the encoding for an enhanced verification method for queries of bounded sizes.

In summary, the contributions of this paper are as follows:

- At query time, we first propose a new candidate subgraph exploration in the absence of query structure, to reduce a large data graph for query processing. We propose further reducing the size of candidate subgraphs by using neighborhood containment.
- Since candidate matchings are determined from candidate subgraphs, we propose a subgraph cache to prune the candidate matchings that are enumerated.
- We propose a robust encoding scheme and its verification method. We propose a model for the client to determine a proper encoding for his/her query.
- We conduct extensive experiments with real datasets to investigate the effectiveness and efficiency of our proposed methods.

Organization. Sec. III introduces the problem. We provide the preliminaries in Sec. III Sec. IV presents query preprocessing at clients. Sec. V details the structure-preserving optimizations that minimize the candidate mappings. Sec. VI presents the verification of subgraph isomorphism mapping in an encrypted domain. We analyze privacy in Sec. VII Sec. VIII shows the experimental results and Sec. IX compares the related work in the literature. We conclude in Sec. X.

II. PROBLEM FORMULATION

This section formulates the technical problem. More specifically, it presents the system model, attack model, privacy target, and problem statement.

System model. The system model resembles the classical server-client model, which contains two parties (illustrated in Fig. 1): (1) A Service Provider (SP) and (2) the query client (or simply client). The SP is equipped with powerful computing utilities such as a cloud and hosts a subgraph query service for publicly known graph data G. The client encrypts his/her query Q using a secret key (generated by himself/herself) as Qk and submits Qk to the SP. The SP then processes the client’s encrypted query Qk over the data G, and returns an encrypted result to the client. The client decrypts the result to obtain the query answer.

Attack model. We assume the semi-honest (adversary) model which is widely used in the database literature [2], [3], [17], [20], where the SP is honest-but-curious. That is, the attacker may be the SP or another adversary hacking the SP. The SP performs computations according to the system model but it may be interested in inferring secrets. For presentation simplicity, we often call the attacker the SP. We assume that the attacker can be both eavesdropping and adopting the chosen plaintext attack (CPA) [20].

Privacy target. To facilitate technical discussions, we assume that the privacy target is to protect the structures of the query graph Q from the SP under the attack model defined above. The structural information of Q that we consider is the adjacency matrices (i.e., the edge information of Q). It is obvious that the complete structure of a query can be derived from the edge information.

To sum up, the problem statement of this paper can be stated as follows: Given the above system and attack model, we seek an efficient approach to complete the subgraph query service while preserving the privacy target.

III. PRELIMINARIES AND OVERVIEW

In this section, we first provide preliminary concepts related to subgraph queries. Then, we present an overview of our proposed solution.

A. Subgraph Queries

The graph G = (V, E, Σ, L) considered in this paper is an undirected labeled connected graph, where V(G), E(G), Σ(G) and L are the set of vertices, edges, vertex labels and the function that maps a vertex to its label, respectively. We use nb(v, G) to denote the set of neighbors of v in G. We use occ(ℓ, G) to represent the number of occurrences of the label
We use $M_G$ to represent the adjacency matrix of $G$. $M_G(v_i, v_j)$ is a binary value, where $M_G(v_i, v_j) = 1$ if $(v_i, v_j) \in E(G)$, otherwise 0. The adjacency matrix $M_G$ represents the edge information. For the clarity of technical details, we present our technique with graphs having vertex isomorphism. We say a graph $G$ is a subgraph of another graph $G'$ iff there exists a subgraph isomorphism mapping (or mapping for short) from $G$ to $G'$, denoted as $G \subseteq G'$ or $\text{subiso}(G, G') = \text{true}$. In this paper, we study subgraph queries stated as: given a query graph $Q$ and a data graph $G$, the subgraph query is to determine if $\text{subiso}(Q, G) = \text{true}$. It is well known that deciding whether $Q$ is the subgraph of $G$ is NP-hard.

**Definition 3.1:** Given two graphs $G = (V, E, \Sigma, L)$ and $G' = (V', E', \Sigma', L')$, a subgraph isomorphism mapping from $G$ to $G'$ is an injective function $f : V(G) \rightarrow V(G')$ such that

- $\forall u \in V(G), f(u) \in V(G'), L(u) = L'(f(u))$; and
- $\forall (u, v) \in E(G), (f(u), f(v)) \in E(G')$.

**B. Overview of Our Approach**

An overview of our solution is sketched in Fig. 2. Our solution essentially consists of the algorithms at the client side and those at the $SP$ side.

**Client-side algorithms.** For the algorithms at the client side, we propose performing lightweight optimization and encryption on the query graph $Q$. (1) We first analyze the query to determine the starting label $\ell_s$ and the minimum height $h$ of $Q$, which are useful for minimizing the number and the sizes of candidate subgraphs of $G$. A candidate subgraph is a subgraph in $G$ that may contain a candidate matching, whereas a candidate matching is a subgraph of the candidate subgraph that may generate a candidate mapping between $Q$ and $G$. (2) We then propose a robust encoding scheme for $Q$ (of any size). (3) We adopt the private-key encryption scheme CGBE to encrypt the encoded $Q$ to encrypted query $Q_k$, which is issued to the $SP$ for query processing. (4) The client decrypts the encrypted answer returned by the $SP$.

**Server-side algorithms.** The main ideas of the algorithms at the $SP$ side are to localize and minimize the enumeration of candidate mappings between $Q$ and $G$ in candidate subgraphs. (1) The $SP$ first efficiently determines the candidate subgraphs $CS_s$ (subgraphs) starting from each starting vertex $s$ of the label $\ell_s$ with the traversal depth $h$. We propose neighborhood containment (NC) to minimize each $CS_s$ in the absence of the structure of $Q$. Subsequently, it minimizes the number of candidate mappings to be enumerated by the $SP$. (2) In each $CS_s$, the $SP$ enumerates all candidate matchings ($CM_s$) and candidate mappings. We propose a canonical labeling-based subgraph cache and apply neighborhood equivalent class (NEC) to further avoid redundant $CM_s$ and candidate mappings. (3) We derive structure-preserving verification $SPVerify$ from [8], where multiple encrypted messages $R_k$ (with negligible false positives) are returned to the client for decryption of the result.

$SPVerify$ is derived from the seminal subgraph isomorphism algorithm: the Ullmann’s algorithm [27]. The major benefit is that its computation flow is simple; hence, we can cast the algorithm into a series of matrix operations (additions and multiplications). Since the encryption of $SPVerify$ supports such matrix operations, privacy is preserved.

We also note that $SPVerify$ may send multiple messages to the client for decryption, which may result in high decryption and network communication costs. Thus we propose $SPVerify^*$. The major difference between $SPVerify^*$ and $SPVerify$ is that $SPVerify^*$ uses different query encodings according to different query sizes and significantly fewer encrypted messages are returned for decryption, and the query size is smaller than a system-related constant.

**IV. QUERY PREPROCESSING AT THE CLIENT**

In this section, we introduce a preprocessing method of the query graph. It comprises three steps: (1) retrieving optimization parameters; (2) encoding the query; and (3) encrypting the encoded query. The encrypted query is sent to the $SP$.

**A. Retrieving Parameters for Optimization**

In order to minimize (1) the size of each candidate subgraph $CS$, and (2) the total number of $CS_s$s, the $SP$ requires the minimum height $h$ of $Q$ and, in the meantime, the starting label $\ell_s$ of $CS_s$s that is infrequent in $G$. These parameters ($h$ and $\ell_s$) are efficiently retrieved by the client.

Given a starting label $\ell_s$, the $SP$ generates $CS_s$s by a breadth first search bounded by the depth $h$ starting at each vertex of $G$ having the label $\ell_s$ (to be detailed in Sec. V.A). On the one hand, to minimize the size of each $CS_s$, we simply find the spanning tree of $Q$ with a minimum height $h$ rooted from a vertex $u$, where $u \in V(Q)$ and $\ell_s = L(u)$. Intuitively, the smaller the value $h$, the smaller the size of each $CS_s$. Note that we cannot choose the vertex $u$ with $h = 1$ since it trivially leaks the structure of $Q$ (to be analyzed in Sec. V1). When there is a tie (i.e., when vertices $u$ and $v$ of $Q$ have the same
the client selects the vertex of the label that is less frequent in G, simply because the number of CSₖ's is bounded by the occurrence of the label in G.

Example 4.1: Fig. 3(a) shows an example of the selection of the starting label of query Q. The heights of the spanning trees rooted from u₁, u₃, and u₄ are 2. u₁ is finally chosen as the starting label as \( \text{occ}(0, G) < \text{occ}(2, G) \), where \( L(u₁) = 0 \), and \( L(u₃) = L(u₄) = 2 \). u₂ is not considered because the height of its spanning tree is 1.

B. Query Encoding

For presentation brevity, we present an encoding scheme for the query Q (in Definition 4.1) to facilitate the discussion of the subsequent encryption scheme. This encoding is extended for further optimization (to be proposed in SubSec. VI-B).

Definition 4.1: The encoding of the entries of \( M_Q \) are:

\[
\begin{align*}
M_Q(u_i, u_j) &= q \quad \text{if } M_Q(u_i, u_j) = 0; \text{and} \\
M_Q(u_i, u_j) &= 1 \quad \text{otherwise},
\end{align*}
\]

where \( q \) is a large prime number.

Example 4.2: Fig. 3(a) also shows an example of the encoding of Q by Def. 4.1. The entries in \( M_Q \) with values 0 are replaced by the large prime \( q \).

C. Query Encryption

Based on the encoding of Q, we adopt our recent private-key encryption scheme [8] (cyclic graph based encryption scheme CGBE) to encrypt the encoding of Q (\( M_Q \)). CGBE not only allows for efficient encryption and decryption but also supports both partial additions and multiplications, which is the core of efficient structure-preserving computation.

Background on cyclic group. Prior to the presentation of the definition of CGBE, we first recall the preliminaries of cyclic group [20]. Let \( G \) be a group. \( p = |G| \) is denoted as the order of \( G \). In particular, \( \forall g \in \mathbb{G} \), the order of \( G \) is the smallest positive integer \( p \) s.t. \( g^p = 1 \). Let \( \langle g \rangle = \{g^i : i \in \mathbb{Z}_p, g^i \in \mathbb{Z}_n\} = \{g^0, g^1, \ldots, g^{p-1}\} \) denote the set of elements generated by \( g \). The group \( G \) is called cyclic if there exists an element \( g \in G \) such that \( \langle g \rangle = G \). \( g \) is called a generator of \( G \).

CGBE scheme. The cyclic group based encryption scheme is defined as follows.

Definition 4.2: [8] The cyclic group based encryption scheme is a private-key encryption scheme, denoted as \( \text{CGBE} = (\text{Gen}, \text{Enc}, \text{Dec}) \), where

- Gen is a key generation function, which generates a secret key \( x \) uniformly at random, a cyclic group \( \langle g \rangle = \{g^i : i \in \mathbb{Z}_p, g^i \in \mathbb{Z}_n\} \). It outputs the private keys as \( (x, g) \) and the value \( p \) which is known to the public.

- Enc is an encryption function, which takes as input a message \( m \) and the secret key \( (x, g) \). It chooses a random value \( r \), and outputs the ciphertext \( e = mrq^x \pmod{p} \).

- Dec is a decryption function, which takes as input a ciphertext \( e \), and the secret key \( (x, g) \). It outputs \( m = \frac{e - qr^x}{g^x} \pmod{p} \).

Note that the decryption function Dec in CGBE only decrypts the ciphertext \( e \) as a product of the message \( m \) and the random value \( r \).

Query encryption. With CGBE, we define the encryption of \( M_Q \) as follows.

Definition 4.3: The encryption of Q is denoted as \( Q_k, Q_k = \{V, M_Q, \Sigma, L\} \), where \( \forall u_i, u_j \in V(Q) \),

\[
M_{Q_k}(u_i, u_j) = \text{Enc}(M_Q(u_i, u_j), x, g)
\]

Example 4.3: For example, \( \forall u_i, u_j, \) if \( M_Q(u_i, u_j) = 1 \), then \( M_{Q_k}(u_i, u_j) = \text{Enc}(1) = rg^x \pmod{p} \); and if \( M_Q(u_i, u_j) = q \), then \( M_{Q_k}(u_i, u_j) = \text{Enc}(q) = r^qg^x \pmod{p} \).

Discussion. We remark that the client holds the secret keys \( (x, g) \) for decryption and moreover, determines the constant \( c \) and an encrypted value \( I \) for encrypting verification results (to be discussed in Sec. VI). At last, \( \ell_s, h, Q_k, c, I \) and \( p \) are sent to the SP for structure-preserving query processing.

V. MINIMIZED SP MAPPING GENERATION

The query preprocessing at the client side (in Sec. [V]) generates \( (\ell_s, h, Q_k, c, I, p) \) for the SP. Upon receiving these, the SP performs structure-preserving subalgo (termed SPSubalgo), presented in Algo. [3]

As outlined in Sec. [I], the SP first minimizes the number of candidate mappings to-be-verified. For brevity, we focus on the most crucial procedures: candidate subgraph generation (Sec. [V-A]), candidate matching generation (Sec. [V-B]), and candidate mapping enumeration (Sec. [V-C]).

A. Candidate Subgraph Generation

To avoid enumerating mappings on a possibly large graph, the SP first generates candidate subgraphs (Fig. [4](b)), where possible mappings can only be embedded in those subgraphs. A candidate subgraph is formally described in Def. [5.1]

Definition 5.1: A candidate subgraph \( s \in V(G) \), denoted as \( \text{CS}_s \), is an induced subgraph of \( G, \) s.t.

1) \( L(s) = \ell_s \);
2) \( \forall v \in V(\text{CS}_s), v \) is reachable from \( s \) within \( h \) hops;
3) \( \forall \ell, \ell' \in V(\text{CS}_s) \implies \ell \in \text{Enc}(Q); \) and
4) \( \forall \ell \in V(\Sigma), \text{occ}(\ell, \text{CS}_s) \geq \text{occ}(\ell, Q). \)

Example 5.1: Suppose \( L(s) = \ell_s = 0 \) and \( h = 2 \). Fig. [2](b) sketches an example of a candidate subgraph \( \text{CS}_s \) (the grey-colored shadow) rooted from \( s \) of \( G \). For each vertex \( v \) in...
forms an independent set of that the answer of $L$ candidate subgraph after the minimization.

Example 5.2: the answer of vertices of $V$ (Line 1). For each vertex $v \in V$, where $L(s) = L(s)$, it invokes GenCandSubGraph (Line 1). GenCandSubGraph simply generates $C_S$ by a breadth first search method started from $s$ on $G$ within $h$ hops (Lines 10-15). $V_{CS}$, is to record the vertices of $CS_1$ determined so far. For each vertex $v \in V_{CS}$, $v$ may be reachable from $s$ within $h$ hops (Lines 13-15), and $L(v) \in \Sigma(Q)$ (Line 13). If $\forall v \in V_{CS}$, $\text{occ}(\ell, CS_1) = \text{occ}(\ell, Q)$ (Line 16), $CS_1$ is set to the induced subgraph of $V_{CS}$ in $G$ (Line 17).

Minimization by neighborhood information. Since the sizes of candidate subgraphs have a significant impact on performance, we propose MinCandSubGraph (Procedure I2) to minimize the size of each $CS_1$. MinCandSubGraph is derived based on our notion of neighborhood containment class (NC) of $CS_1$, defined as follows.

Definition 5.2: Given $N = \{v_1, v_2, \ldots, v_n\}$ of $V(C_S)$, $N$ is a neighborhood containment class (NC), denoted as $v_1 \subseteq v_2 \subseteq \cdots \subseteq v_n$, if $v_i \in V$, $v_j \in N$, $i < j$.

1) $L(v_i) = L(v_j)$;
2) a) $\text{nb}(v_i, CS_1) \subseteq \text{nb}(v_j, CS_1)$, if $N$ is an independent set in $CS_1$; or
b) $\text{nb}(v_i, CS_1) \cup \{v_j\} \subseteq \text{nb}(v_j, CS_1) \cup \{v_j\}$, if $N$ is a clique of $CS_1$.

Based on Def. 5.2 the vertices of a candidate subgraph $CS_1$ exhibit a total ordering with respect to the $\subseteq$ relationships. We have the following lemma for minimizing the size of a candidate subgraph by keeping the “top” vertices in the subgraph. The intuition is that the reduced $CS_1$ preserves all the structures of the original $CS_1$. The proof is established via a simple contradiction. Due to space limits, we present it in our technical report [9].

Lemma 5.1: Denote an NC $N = \{v_1, v_2, \ldots, v_n\}$, where $N \subseteq V(C_S)$ of a graph $G$. Denote the reduced $V(C_S)$ (denoted as $C_S^r$) is the induced subgraph of $V(C_S) \setminus \{N\}$ of $C_S$, i.e. $N_k = \{v_{n-k+1}, v_{n-k+2}, \ldots, v_n\}$ contains top-k vertices of $N$ that are kept, where $k = \text{occ}(L(v_1), Q)$. Then, the answer of $Q$ on $C_S$ is the same as that on $C_S^r$.

Example 5.2: Reconsider Example 5.1 $\{v_1, v_2\}$ is an NC as $L(v_1) = L(v_2), \text{nb}(v_1, CS_2) \subseteq \text{nb}(v_2, CS_1)$ and $\{v_1, v_2\}$ forms an independent set of $CS_2$ in Fig. 3(c). Since $\text{occ}(1, Q) = 1$, by Lemma 5.1 we keep the top-1 vertex. It can be seen that the answer of $Q$ remains the same after removing either $v_1$ or $v_2$ from $CS_2$. For another example, let’s consider the NC $\{v_4, v_5, v_6\}$ in Fig. 3(c), as the neighborhood of $v_4$ is contained by that of $v_5$. Hence, $v_4 \subseteq v_5$. Similarly, $v_5 \subseteq v_6$, $v_4, v_5, v_6$ forms an independent set. Again, by Lemma 5.1 we keep only the top-1 vertex, i.e., $v_6$. The answer of $Q$ remains the same after removing $v_4$ and $v_5$. All in all, Fig. 4(a) shows $CS_1$, the candidate subgraph after the minimization.

Algorithm 1 $\text{SP} \backslash \text{sublo}(Q_{k}, G, \ell_s, h)$

**Input:** The encrypted query graph $Q_{k}$, data graph $G$, starting label $\ell_s$ and hop $h$.

**Output:** The encrypted result $R_k$.

1: Initialize $CS_1 = CM_1 = \emptyset$, Cache = $\emptyset$, and $R_k = 1$
2: for each vertex $s \in V(G)$ with the starting label $\ell_s$
3: $\text{GenCandSubGraph}(Q_k, G, s, h, CS_1)$ /* By Def. 5.1 */
4: $\text{MinCandSubGraph}(Q_k, CS_1)$ /* Minimize $CS_1$ */
5: Initialize set $V_{CS_1} = \{s\}$
6: $\text{GenCandMatch}(V_{CM}, Q_k, CS_1, R_k, Cache)$ /* By Def. 5.1 */
7: return $R_k$

**Procedure 1** $\text{GenCandSubGraph}(Q_k, G, s, h, CS_1)$

1: Initialize $L_1 = \emptyset$
2: $\text{GenInducedSub}(G, s, h)$
3: $\text{MinCandSubGraph}(Q_k, CS_1)$
4: $\text{MinCandMatch}(V_{CM}, Q_k, CS_1, R_k, Cache)$ /* By Def. 5.1 */
5: return $R_k$

**Minimization procedure $\text{MinCandSubGraph}$**

Procedure I2 shows the minimization of $CS_1$ by NC. For each $\ell \in \Sigma(C_S)$, a set $N$ of NC is first initialized as $\{\}$ (Line 18). For each vertex $v \in CS_1$ with the label $\ell$, sorted in ascending order of $|\text{nb}(v, CS_1)|$ (Line 19) for efficiency, $\text{MinCandSubGraph}$ checks if there is an $N$ in $N$, such that $N \cup \{v\}$ forms an NC by Def. 5.2 (Line 20). If so, $v$ is then inserted into $N$ (Line 21). Otherwise, the algorithm creates a new $N = \{v\}$ and unions $N$ to $N$ (Line 22). After the generation of NC of $CS_1$ for the label $\ell$, $CS_1$ can be minimized by Lemma 5.1 via keeping the top-k vertices in each $N$. $\text{NC}$ is set to the induced subgraph of $\text{occ}(\ell, Q_k)$ (Lines 23-24).

**Complexity.** The complexity of the generation of NC in Procedure I2 is $O(d_{max}|V(C_S)|^2)$, where $d_{max}$ is the maximum degree of the vertices in $C_S$. In practice, $|V(C_S)|$ is often in the order of hundreds, which is small.

**B. Candidate Matching Generation**

A unique challenge in structure-preserving query processing is that, in the absence of query structure, the SP matches $Q_k$ to multiple possible subgraph structures in $C_S$. We call such subgraph structures candidate matchings. In contrast, if the query structures were not kept secret, the candidate matching was known to be $Q$. Fig. 4(a) shows four candidate matchings, $CM_{12}$, $CM_{13}$, $CM_{23}$, and $CM_{24}$. For each matching, candidate mappings are enumerated. It is evident that a naive enumeration of all candidate matchings can be inefficient. In this subsection, we propose GenCandMatch to efficiently generate candidate matchings. The main idea is to avoid generating redundant matchings from $CS_1$. 
Definition 5.3: A candidate matching, denoted as \( CM_s \), is a connected induced subgraph of \( CS_s \), s.t.

1) \( |V(CM_s)| = |V(Q)| \); and  
2) \( \forall \ell \in \Sigma(CS_s), \text{occ}(\ell, CM_s) = \text{occ}(\ell, Q) \).

Example 5.3: Fig. 4 (a) lists all the \( CM_s \)s enumerated from \( CS_s \). \( i \in \{1, ..., 4\} \), \( |V(CM_s)| = |V(Q)| \), and \( \forall \ell \in \Sigma(CS_s), \text{occ}(\ell, CM_s) = \text{occ}(\ell, Q) \).

Elimination of redundant \( CM_s \). We make two observations from Example 5.3 and Fig. 4: (1) \( CM_s \) is graph-isomorphic to \( CM_s' \). If candidate mappings are generated from \( CM_s \), it is obvious that generating mappings from \( CM_s' \) is redundant. (2) \( CM_s \) is a supergraph of \( CM_s' \). One can simply generate mappings from \( CM_s \), and skip \( CM_s' \) and \( CM_s'' \).

To remove the redundancies mentioned above, it is exactly to solve the following problem: “given a graph \( G \) and a graph database \( \mathcal{G} : \{G_1, ..., G_n\} \), how to efficiently determine if \( G \) is a subgraph of \( G' \), \( G' \in \mathcal{G} \)?” Such a problem has been extensively studied before (e.g., [25], [29]). Existing solutions involve an index computed offline. In our context, candidate mappings are enumerated online. Hence, the existing solutions cannot be directly applied.

Canonical labeling-based subgraph cache. Let’s recall a crucial property of canonical labeling. In the context of graph query processing, the canonical labeling of a graph \( G \) is denoted as \( \text{can}(G) \), and \( \text{can}(G) = \text{can}(G') \) if and only if \( G \) is isomorphic to \( G' \). While the cost for computing the canonical labeling of a graph is not yet known (P or NP), the cost for comparing whether two graphs are isomorphic using the labeling is \( O(1) \), once computed. This work adopts the minimum \( dfs \) code [29] from the literature.

For each query, we propose \( Cache \) to store \( \text{can}(CM_s) \), where each \( CM_s \) is the checked candidate matching. Once a new \( CM_s' \) is generated, we first check if \( \text{can}(CM_s') \) is already in \( Cache \). If so, \( CM_s' \) is discarded. Otherwise, we insert \( \text{can}(CM_s') \) into \( Cache \). Further, we continue to enumerate subgraphs \( CM_s \)s from \( CM_s' \), where for each \( CM_s', |V(CM_s')| = |V(CM_s)|, CM_s' \subseteq CM_s', \) and \( \text{can}(CM_s') \) is stored in \( Cache \). Putting subgraphs of \( CM_s' \) increases the chance of pruning by \( Cache \). However, the trade-off is that as the query size increases, the computational cost for enumerating all subgraphs of \( CM_s' \) increases exponentially. Thereby, for practical purposes, we enumerate all of the subgraphs \( CM_s \)s of \( CM_s' \) only if \( |V(Q)| \leq \delta \), where \( \delta \) is a user-defined threshold.

Example 5.4: The top of Fig. 4 (a) shows the idea of the canonical labeling-based cache. We assume that \( \delta = 3 \), and the sequence of the generation of \( CM_s \) is from \( CM_s' \) to \( CM_s' \). \( CM_s' \) is eliminated as \( \text{can}(CM_s') \) is in \( Cache \). If we set \( \delta = 5 \), then \( CM_s' \) and \( CM_s'' \) are both eliminated, because \( CM_s' \) is a subgraph of \( CM_s'' \), and when \( CM_s'' \) is processed, \( \text{can}(CM_s') \) is inserted into \( Cache \).

The ordering in \( CM_s \) generation. From Example 5.4, it can be observed that the ordering in \( CM_s \) generation affects the performance of the cache, when \( |V(Q)| \leq \delta \). Suppose \( \delta = 5 \). Assume \( CM_s' \) is generated before \( CM_s'' \). Then, \( CM_s' \) is not eliminated. In general, the earlier the larger \( CM_s \)s are generated, the better the performance is. Therefore, we find a simple ordering for \( CM_s \) generation, by greedily adding vertices to \( CM_s \) by the degree of each vertex.

Due to space limitations, we skip the pseudo-code for \( CM_s \) generation (GenCanMatch) [9], which is essentially an enumeration incorporated with the subgraph cache.

C. Candidate Mapping Generation

For a new candidate matching \( CM_s \) is generated, GenCanMatch (though its pseudo-code is omitted for brevity) invokes GenAllMap (Algo. 2) to enumerate all possible mappings between \( Q_k \) and \( CM_s \).
Suppose that \(u_3\) and \(u_4\) (in Fig. 4(a)) have been mapped to \(v_5\) and \(v_6\), respectively. It is not necessary to map \(u_3\) and \(u_4\) onto \(v_6\) and \(v_5\), respectively. This can be formalized as the following lemma. Foremost, we often use \((u_i \mapsto v_j)\) to denote map\([u_i]\) \(= v_j\) for ease of exposition.

**Lemma 5.2:** Suppose the following are true:

1. \(u_i, u_j \in V(Q), v_{i'}, v_{j'} \in V(CM_s), L(u_i) = L(u_j) = L(v_{i'}) = L(v_{j'});\)
2. \(v_i \neq v_{i'};\)
3. \((u_i \mapsto v_{i'})\) and \((u_j \mapsto v_{j'}).\)

Let map’ be the mapping map except that \((u_i \mapsto v_{i'})\) and \((u_j \mapsto v_{j'}).\) Then, map’ is a candidate mapping between \(Q\) and \(CM_s\) if and only if map’ is also a candidate mapping.

The proof is omitted since it can be established by a simple proof by contradiction. Next, we present the data structures and the mapping generation, that exploit the lemma.

**Data structures.** (i) A vertex label mapping \(M\) is a \(m \times n\) binary matrix, \(m = |V(Q)|\) and \(n = |V(CM_s)|.\) Specifically, \(\forall u, v, M(u, v) = 1\) if \(L(u) = L(v),\) where \(u \in V(Q)\) and \(v \in V(CM_s);\) and otherwise 0. (ii) A vector map of the size \(|V(Q)\)| is to record a mapping from \(Q_k\) to \(CM_s,\) map\([u] = v (i.e., u \mapsto v)\) represents that vertex \(u\) in \(Q_k\) is mapped to vertex \(v\) in \(CM_s.\) map\([u]\) = 0 if \(u\) is not yet mapped. (iii) A vector used of the size \(|V(CM_s)|\) is to denote whether the vertex \(v\) in \(CM_s\) has been mapped to a vertex of \(Q_k\) and recorded in map. map\([u]\) = 0 if \(u\) is not yet mapped. In other words, map\([u]\) = 1 if and only if map\([u]\) = \(v\) for some \(u \in Q_k.\)

**Algorithm for mapping generation.** The detailed algorithm GenAllMap is shown in Algo. 2. It first initializes the data structures, including \(M,\) and map and in Lines 1-3. Line 4 constructs NEC of \(CM_s,\) which is similar to that of NC in Procedure 1.2. EnumMap (Lines 6-14) is then invoked to enumerate all possible mappings. A mapping map is constructed vertex by vertex iteratively. Line 9 checks if \(v_j\) is a possible map of \(u_i\) by \(M\) and used. We then exploit the equivalence class to further check if \(v_j\) can be possibly mapped to \(u_i\) (Lines 10-12). The vertices in a NEC are checked in a predefined order (e.g., lexicographical order). If \(\exists v_{i'} s.t. v_{i'} \neq v_{j'} \neq j\) and \(v_{j'}\) is not used before, then \(v_j\) is skipped (Line 10). If \(v_j\) passes the check, EnumMap is called recursively (Line 13) until a full mapping is constructed (Line 6).

**Example 5.6:** Fig. 4(b) illustrates the possible candidate mapping generation for those \(CM_s\) of Example 5.2. Since \(\bar{v}_5 \neq v_6\) in \(CM_s,\) by Lemma 5.2 we only enumerate map\(_5\), where \(u_3 \mapsto v_5\) and \(u_4 \mapsto v_6,\) but the one with \(u_3 \mapsto v_6\) and \(u_4 \mapsto v_5\) is eliminated.

**VI. SP Mapping Verification**

Section 5 presented a series of optimizations that reduce the number of mappings to be generated. Then, for each mapping map, the SP verifies (in the encrypted domain) if there is no violation in map. The encrypted verification results are aggregated before they are transmitted to the client. In this section, we derive a basic verification (SPVerify) from [8] for our problem setting. Next, we propose an enhanced one (SPVerify*) that aggregates many more messages but requires the query size to be smaller than a user-determined constant.

**A. SPVerify**

Given a mapping map between \(Q_k\) and \(CM_s,\) we determine if \(CM_s\) is a valid mapping or not. Specifically, we define the violation w.r.t. the encoding of \(Q\) as follows: \(\exists u_i, u_j \in V(Q),\) \(M_Q(u_i, u_j) = 1 \land (v_{i'}, v_{j'}) \notin E(CM_s) (1)\) where \(v_{i'}, v_{j'} \in V(CM_s), u_i \mapsto v_{i'}\) and \(u_j \mapsto v_{j'}\). It states that there exists an edge between vertices \(u_i\) and \(u_j\) in \(Q,\) but there is no corresponding edge between the mapped vertices \(v_{i'}\) and \(v_{j'}\) in \(CM_s.\) We term the case in Formula 1 as a violation of subgraph isomorphism (or simply violation). A mapping without violation(s) is called a valid mapping.

**Example 6.1:** Let’s take the two mappings map1 and map2 of \(CM_s\) in Fig. 4(b) as an example. First, no violation is found in map1. Second, for map2, we find that \(M_Q(u_i, u_j) = 1\) and \((s, v_4) \notin E(CM_s),\) where map2\([u_i]\) = \(s\) and map2\([v_4]\) = \(4.\) Therefore, map2 is invalid.

**Algorithm for SPVerify.** The intuitive idea of SPVerify is to transform the verification steps into mathematical operations on \(M_{Q_k}\) and \(CM_s,\) where (1) the violation (Formula 1) can be detected; (2) only matrix additions and multiplications are involved; and (3) the result can be aggregated with one message or multiple messages.

Algo. 3 shows the detailed algorithm. The inputs are a candidate mapping map, an encrypted query graph \(Q_k,\) a candidate matching \(CM_s\) and an encrypted result \(R_k.\) We remark that \(R_k\) is to record the aggregated result for \(CM_s,\) where \(R_k\) is initialized to 1 in Line 1 Algo. 1.

We initialize an intermediate result \(R_i\) with a value 0 (Line 1). For each pair of vertices \((u_i, u_j)\) in \(V(Q)\) and the mapped vertex pair \((v_{i'}, v_{j'})\) in \(CM_s\) (Lines 2-3), the following two steps are performed:

1. **Additions (Lines 4-7):** if \((v_{i'}, v_{j'}) \notin E(CM_s), R_i\) is set to \((M_{Q_k}(u_i, u_j) + R_i) \pmod{p}\). This indicates that if \((u_i, u_j)\) is an edge in \(Q, R_i\) must not contain a factor of \(q,\) and the decryption value of \(R_i\) is non-zero (i.e., the current mapping map contains a violation (by Formula 1), which is not a valid mapping). Otherwise, no violation is caused by \((u_i, u_j).\) This sets \(R_i\) to the value \(I + R_i \pmod{p},\) where \(I\) is an encrypted value with a factor \(q\) issued by the client, \(I = Enc(q);\) and

2. **Multiplications (Line 8):** it aggregates \(R_i\) into \(R_k,\) by \(R_k = R_k \times R_i \pmod{p}\). If there is at least one valid mapping from \(Q\) to \(G, i.e.,\) at least one \(R_i\) whose decryption value is zero. The decryption value of \(R_k\) must also be zero. Otherwise, it is non-zero. We remark that CGBE leads to errors if the number of \(R_i\) aggregated in \(R_k\) is larger than a predetermined value \(M.\)
Algorithm 3 SPVerify(map, $Q_k$, $CM_s$, $R_k$)

1: Initialize $R_i = 0$
2: for each $u_i, v_i \in V(Q), i < j$
3: \hspace{1em} $v'_i = \text{map}[u_i], v_j = \text{map}[u_j]$
4: \hspace{1em} /* Additions */
5: \hspace{1em} if $(v'_i, v'_j) \notin E(CM_s)$
6: \hspace{1em} \hspace{1em} $R_i += M_{Q_k}(u_i, u_j)$ (mod $p$) /* Aggregate violation */
7: \hspace{1em} else
8: \hspace{1em} \hspace{1em} $R_i = I$ (mod $p$) /* No violation, $I = \text{Enc}(q)$ */
9: \hspace{1em} /* Multiplications */
10: \hspace{1em} $R_i \times R_j$ (mod $p$) /* Decompose $R_k$ after aggregating $M, R_i$ */

Example 6.2: Fig. 3(a) depicts an example of SPVerify between $Q_k$ and $CM_{s1}$. There are two mappings from $Q_k$ to $CM_{s1}$ in Fig. 4(b). In map1, all the factors in $R_1$ contain $q$ since map1 is a valid mapping. However, in map2, since there exists a violation between $(u_1, u_3)$ and $(s, v_4)$, there is a factor in $R_2$ that has no prime $q$. $R_k = R_1 \times R_2$ (mod $p$).

Decryption at the client. After receiving all the encrypted messages $R_k$, the client performs two main steps:

- For each $R_k$, the client computes the plaintext of $R_k$ by $R_k' = \text{Dec}(R_k, x, g)^{M_i}$;
- The client computes the final result by $R = R_k' (\text{mod } q)$. If $R$ equals zero if and only if there is at least one valid mapping from $Q$ to $G$, and thus, $\text{subiso}(Q, G) = \text{true}$.

Algorithm 4 SPVerify* (map, $Q_k$, $CM_s$, $R_k$)

1: Initialize $R_i = 1$
2: for each $u_i, v_i \in V(Q), i < j$
3: \hspace{1em} $v'_i = \text{map}[u_i], v'_j = \text{map}[u_j]$
4: \hspace{1em} /* Additions */
5: \hspace{1em} if $(v'_i, v'_j) \notin E(CM_s)$
6: \hspace{1em} \hspace{1em} $R_i += M_{Q_k}(u_i, u_j)$ (mod $p$) /* Aggregate violation */
7: \hspace{1em} else
8: \hspace{1em} \hspace{1em} $R_i = I$ (mod $p$) /* No violation, $I = \text{Enc}(1)$ */
9: \hspace{1em} /* Multiplications */
10: \hspace{1em} $R_i \times R_j$ (mod $p$) /* Decompose $R_k$ after aggregating $M, R_i$ */

Proposition 6.1: The probability of false positives in $R_i$ is $\frac{1}{q}$, which is negligible.

Proposition 6.2: The probability of false positives in $R_k$ is $1 - e^{-\frac{R_k}{q}}$, which is negligible, in each decomposed number.

B. Optimized SPVerify for Queries of Bounded Sizes

Each encrypted message $R_k$ sent by SPVerify aggregates at most $M$ messages $R_i$s. In this subsection, we propose SPVerify*, which significantly reduces the number of messages returned, and in turn reduces both the communication and computational costs at the client. The main idea behind SPVerify* is to use multiplications to detect violations since queries are often small and use additions to aggregate $R_i$s. Hence, the value of $R_k$ may not exceed $p$ even after many aggregations. However, a tradeoff of SPVerify* is that the query size must be bounded by a pre-determined constant $c$.

Similiar to SPVerify, SPVerify* also detects the violation by multiplications and additions. In order to achieve that, we first define a complement encoding of the query (see Def. 6.1).

Definition 6.1: The encoding of the entries of $M_Q$ are:

$$\forall u_i, u_j \in V(Q),
\begin{cases}
M_Q(u_i, u_j) = 1 & \text{if } M_Q(u_i, u_j) = 0 \\
M_Q(u_i, u_j) = q & \text{otherwise}
\end{cases}
$$

where $q$ is a large prime number.

In relation to Def. 6.1, we adopt Formula 1 to state the violation:

$$M_Q(u_i, u_j) = q \land (v'_i, v'_j) \notin E(G)$$

where $v'_i, v'_j \in V(G)$, $u_i \mapsto v'_i$ and $u_j \mapsto v'_j$.

Algorithm for SPVerify*. For ease of comparison, we present the pseudo-code of SPVerify* (shown in Alg. 4) in the style of SPVerify. The inputs and the initialized data structures are the same as SPVerify, except that $R_k$ is initialized to 0. The two main steps of SPVerify* can be highlighted as follows:

1. Multiplications (Lines 4-7): according to the violation (by Formula 3), if $(v'_i, v'_j) \notin E(CM_s)$, set $R_i$ as the value $M_{Q_k}(u_i, u_j) \times R_i$ (mod $p$). This indicates that as long as $(u_i, u_j)$ is an edge in $Q$, $R_i$ must contain the factor $q$, and the decryption value is zero (i.e., the current mapping map contains a violation). Otherwise, $R_i$ is set to a value $I \times R_i$ (mod $p$), where $I$ is an encrypted value without factor $q$ issued by the client, $I = \text{Enc}(1)$; and

2. Additions (Line 8): it aggregates $R_i$ to $R_k$, where $R_k = R_k + R_i$ (mod $p$). If there is at least one valid mapping from $Q$ to $G$ (i.e., at least one $R_i$ whose plain text is
Example 6.4: Fig. 5 (b) illustrates an example of SPVerify*. Similarly, since there is no violation in map1, all the factors in $R_1$ do not contain $q$. Regarding map2, since there is no violation, $R_2$ contains a factor $q$. $R_k = R_1 + R_2$ (mod $p$).

Example 6.5: We show the decryption in Fig. 5 (b). For simplicity, we assume that $R_k$ only contains $R_1$ and $R_2$. The client generates $g^{-6x}$, computes $R_k' = R_k \times g^{-6x}$ (mod $p$), and finally computes $R = R_k'$ (mod $q$). The result is non-zero which indicates that $Q$ is a subgraph of $G$.

Determining the constant $c$ to decide when to use SPVerify or SPVerify*. In SPVerify*, multiplications are used to aggregate violations by edges in $CM_s$ (Line 4 in Alg. 4), instead of aggregating numerous mapping results ($R_i$ in Line 8 of Alg. 3). Similarly, when $R_i$ (Lines 4-7) in Alg. 4 exceeds $p$, the client cannot recover the plaintext. The number of multiplications for each $R_i$ is directly related to the size of the query ($|V(Q)|$). We can determine the maximum size of the query, denoted as $c$, using the following inequality.

$$\text{Len}(p) \geq \frac{c(t-1)}{2}(\text{Len}(q) + \text{Len}(r))$$

Putting these together, in Lines 7-8 of Alg. 4, once $|V(Q)| \leq c$, the $SP$ uses SPVerify*. Otherwise, it uses SPVerify.

False positives. Since both SPVerify and SPVerify* use CGBE, we can obtain that the probabilities of false positives of SPVerify* are also negligible. Their proofs are almost identical to those of Props. 6, 1 and 6, 2 and 8, and hence, omitted.

VII. PRIVACY ANALYSIS

In this section, we prove the privacy of the encryption method and SPsubIs. The attack model is the one defined in Sec. 11. The attackers or $SP$s are eavesdroppers and can adopt chosen plaintext attack (CPA) [20].

Privacy of the encryption method. CGBE is adopted to encrypt the query graph in this paper. The privacy of CGBE and $M_{Q_k}$ can be recalled from [8].

Lemma 7.1: CGBE is secure against CPA. $M_{Q_k}$ is preserved from the $SP$ against the attack model under CGBE.

Then, based on Lemma 7.1, we have the following.

Proposition 7.1: The structure of the query is preserved from the $SP$ against the attack model under CGBE.

Proof: (Sketch) The proof can be derived from Lemma 7.1. After receiving $Q_k$, the $SP$ cannot break the $M_{Q_k}$ since they are secure against CPA. $V$, $\Sigma$ and $L$ do not contain structural information. Thus, the structure of query is preserved from the $SP$ against the attack model.

Privacy of SPsubIs. SPsubIs mainly consists of five steps: (1) GenCandSubGraph; (2) MinCandSubGraph; (3) GenCandMatch; (4) GenAllMap; and (5) SPVerify (or SPVerify*). We now analyze the privacy of each step as follows. However, first, the analysis requires some notations. We denote a function $P(m, h, \Sigma)$ that returns all possible graphs of $m$ vertices with a minimum height $h$ and the labels $\Sigma$. $|P(m, h, \Sigma)|$ is exponential to the value $m$ and the size of $\Sigma$. Let $A$ is a function that returns 1 if $SP$ is able to determine the exact structure of $Q$, and 0 otherwise. The probability that the $SP$ can determine the structure of the query $Q$ is denoted as $Pr[A(Q) = 1]$. Given a query $Q$ and $(m, h, \Sigma)$, the probability of determining its structure is $Pr[A(Q) = 1] = 1/|P(m, h, \Sigma)|$.

Proposition 7.2: Under GenCandSubGraph, MinCandSubGraph, GenCandMatch and GenAllMap, $Pr[A(Q) = 1] = 1/|P(m, h, \Sigma)|$.

Proof: (Sketch) The proof is established by one main fact: $SP$ does not utilize any structural information of the query, except the value $h$ in the algorithm.

- GenCandSubGraph utilizes $\ell_s$, $h$, $Q_k$ and $G$ to generate all the $CS_s$.
- MinCandSubGraph minimizes the size of each $CS_s$ by using only the structure of $CS_s$ itself.
- GenCandMatch utilizes $Q_k$ and $CS_s$ to generate $CM_s$;
- GenAllMap enumerates all the possible mappings between $Q_k$ and $CM_s$.

The $SP$ cannot learn the structure of $Q$ by invoking them, and thus the probability of determining a structure remains $Pr[A(Q) = 1] = 1/|P(m, h, \Sigma)|$.

In SPVerify and SPVerify*, $SP$ sends messages to the clients. The clients may terminate the algorithm when a mapping is found, which may leak information to the $SP$. Such a leak can be quantified in the following proposition.

Proposition 7.3: Under SPVerify or SPVerify*, the following hold for:

- If $Q$ is a subgraph of $G$, $Pr[A(Q) = 1] = 1/|S|$, where $S = \{ G \in P(m, h, \Sigma), G \subseteq CM_s \}$, where $CM_s \in \text{Cache} \}$; and
- If $Q$ is not a subgraph of $G$, $Pr[A(Q) = 1] = 1/|P(m, h, \Sigma)|$.

Proof: (Sketch) Since the algorithm SPVerify* is similar to that of SPVerify, due to the space constraint, we prove it with SPVerify only. The proof involves two aspects: (1) $SP$ can never determine any structural information from the mathematical computations in each steps of SPVerify;

Recall that SPVerify comprises a fixed number of mathematical operations in the encrypted domain in Alg. 3.

- Lines 4-7 invoke a constant number of additions of $M_{Q_k}$ and $R_i$, and only structure of $CM_s$ is considered.

More specifically, $\forall i,j, m^2$ additions are invoked for $M_{Q_k}(i, j)$ and $R_i$; and

$^3$We remark that if $h = 1$, the $SP$ is able to infer that the vertex with $\ell_s$ must connect to other vertices in $Q$. To avoid this special case, as mentioned in Sec. IV, we choose the starting vertex where $h$ equals or larger than 2.
TABLE I
STATISTICS OF THE REAL-WORLD DATASETS

<table>
<thead>
<tr>
<th>Graph</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td>311,080</td>
<td>13,049,866</td>
<td>6.62</td>
<td>199</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>3,997,962</td>
<td>34,681,189</td>
<td>17.34</td>
<td>1335</td>
</tr>
</tbody>
</table>

• Line 8 requires one multiplication on each \( R_i \) and \( R_k \).

Based on Lemma 7.1 all the intermediate computations results are securely protected against the attack model. Moreover, each step of SPVerify has a constant number of operations in the encrypted domain. \( SP \) cannot learn any information from them.

(2) \( SP \) may only infer some structural information from the message communications:

Recall that once \( M \) \( R_i \)s are aggregated into \( R_k \), \( R_k \) is returned to the client, the client may decide to terminate SPVerify after receiving \( R_k \)s. There are two cases:

• Suppose there is at least one valid \( R_k \) such that \( Q \) is a subgraph of \( G \). In this case, \( Q \) must be graph (or subgraph) isomorphic to one of \( CM_k \) in Cache. Therefore, \( Pr[A(Q) = 1] = 1/|S| \), where \( S = \{G|G \in P(m, h, \Sigma), G \subseteq CM_k, CM_k \in Cache\} \); and

• If the client does not terminate the algorithm, \( SP \) does not know if there is a valid \( R_k \) or not. Thus, the probability of determining the structure of \( Q \) is still \( Pr[A(Q) = 1] = 1/|P(m, h, \Sigma)| \).

Based on Prop. 7.3 we note that the client can make a tradeoff between privacy and response times by terminating the algorithm as late as acceptable.

VIII. EXPERIMENTAL EVALUATION

This section presents an experimental evaluation of our techniques with popular real datasets. The results show that our techniques are efficient and our optimizations are effective.

The platform. We conducted all our experiments on a machine with an Intel Core i7 3.4GHz CPU and 16GB memory running Windows 7 OS. All techniques were implemented on C++, and CGBE was implemented on the GMP library. We simulate the bandwidth as 10Mbit/s.

Data and query sets. We benchmarked real-world datasets: DBLP, Amazon, Youtube, and LiveJournal. Due to space limitations, we opt to report the performance of DBLP and LiveJournal, since others exhibit similar performance characteristics. Other performance details of Amazon and Youtube are reported in 9. Since the vertices do not have labels, we adopt the approach that uses the degree of the vertex as its label 11. (We tested to assign vertex labels by randomly choosing labels from predefined domains. We noted similar trends. Due to space limits, we skip reporting them.) Some statistics of the datasets are shown in Table II.

For each dataset, we generated two types of queries 26: (1) BFS queries (BFS) and (2) DFS queries (DFS) by random BFS and DFS methods, respectively. Both BFS and DFS contain query sets \( Q_3-Q_8 \), wherein each \( Q_n \) contains 1,000

query graphs, and \( n \) is the number of vertices of each query of the query set. \( h \) of the query sets are around 3-4 on average.

Default values of the parameters. In CGBE, the prime \( p \) and \( q \) are 2048 bits and 32 bits, respectively. The random number \( r \) is 32 bits. The largest value \( c \) is 12 by Formula 9. However, to study the performance of both SPVerify* and SPVerify, we first set \( c \) to 6, by default. That is, if \( |V(Q)| \leq 6 \), we used SPVerify*. Otherwise, we used SPVerify. We finally investigated the effectiveness of SPVerify* with \( c = 11 \). For SPVerify*, we set \( M = 100 \) by default (i.e., we aggregated \( 100 \) \( R_i \)s into each \( R_k \)). For SPVerify, we set \( M = 10 \) only. Unless specified otherwise, \( \delta = 5 \). Under these settings, no false positives was detected from the entire experiments.

Fig. 6. Average preprocessing time at the client.

A. Performance at the Client Side

Preprocessing time at the client. We report the average preprocessing time of the query \( Q \) at the client side on all datasets in Fig. 6. Specifically, the preprocessing of \( Q \) includes (1) the computation for \( \ell_s \) and \( h \); and (2) the encryption of \( Q \) by CGBE. We observe that the average times for each query on all datasets are around 4ms, which shows that the preprocessing is in cognitively negligible.

Fig. 7. Average received encrypted message size at the client.

The sizes of messages received by the client. We report the sizes of the encrypted messages \( R_k \)s that the client received in Fig. 7. Due to the optimizations by SPSubIso, the largest sizes of \( R_k \)s (at \( Q_6 \)) are around 13KB on LiveJournal, which can be efficiently transmitted via today’s networks. For \( Q_7-Q_8 \), as we set \( c \) to 6 (by default), SPSubIso uses SPVerify. The number of \( R_i \)s aggregated in each \( R_k \) is 10. Thus, the message sizes for \( Q_7-Q_8 \) are larger. Since the maximum value of \( c \) is 11 in the current configuration, SPVerify* can be used to produce much smaller messages (to be discussed with Fig. 14).

Fig. 8. Average decryption time at the client.

The decryption time at the client. After receiving the encrypted messages \( R_k \)s, the client decrypts \( R_k \)s. The decryption time is shown in Fig. 8. Since the sizes of \( R_k \)s are small and the decryption method is simple, the average decryption times at the client are correspondingly fast at most 16ms.
pruned by Cache in GenCandMatch. We note that as the query size increases, the effectiveness of Cache increases. For Q3-Q4 of all datasets, the percentage of the elimination of redundant $CM_s$ increases to 80%. For Q5-Q8 on DBLP, the percentages are stable at around 80%. We note the graph structures of LiveJournal are diverse and there are many distinct $CM_s$. The effectiveness of Cache then decreases from 80% to 50% for Q5-Q8. This is also reflected by the fact that the sizes of the encrypted messages $R_k$s are the largest for LiveJournal (see Fig. 7).

The memory consumption of Cache. We report the memory consumption of Cache in Fig. 12. As we only store the hash code of the canonical labeling of each distinct $CM_s$, the memory consumption is very small (at most 25MB).

The number of aggregated messages by SPVerify*. In Fig. 7 since $c$ was set to 6 by default, we used SPVerify for Q7-Q8, where each $R_k$ is an aggregate of $M$ messages and $M = 10$. As discussed, the messages are small. To study SPVerify*, we then set $c = 11$. We used Q8 with DFS and varied the values of $M$ from 10 to 100. Fig. 14 shows the detailed performance of all datasets. We report that for $M = 10$, the message size is the same as those values of Q8 DFS in Fig. 7. Importantly, as $M$ increases, the message size decreases accordingly.

Summary. The experiments show that the computation times at the client were in the order of milliseconds. The messages transferred were small. Most computation was outsourced to the SP. Further, the proposed optimizations were effective.

IX. RELATED WORK

While there has been some work on privacy-preserving query processing, due to space limitations, we can only include the work relevant to graph queries.
Privacy-preserving graph queries. Cao et al. [3] proposed supporting subgraph queries over an encrypted database with a number of small graphs. Their work protects the privacy of the query, index and data features. However, this work does not address the subgraph isomorphism verification of candidate graphs. Cao et al. [2] studied tree pattern queries over encrypted XML documents. The traversal order for each query (required by their method) is predetermined. In the context of graphs, the order cannot be predetermined. He et al. [15] analyzed vertex reachability, while preserving edge privacy. Gao et al. [12] proposed neighborhood-privacy protection for the shortest distance. It aims to preserve the neighborhood connections and the shortest distances between vertices. Mouratidis et al. [24] proposed a shortest path computation with no information leakage using the PIR protocol [6], whose high computational cost is a known concern. Karwa et al. [19] addressed subgraph counts by satisfying the differential privacy of edges. Yin et al. [31] studied private reachability queries. Fan et al. [10], [11] proposed authenticated subgraph query services under the classical data outsourcing setting. A subgraph isomorphism verification method that keeps both query and data graphs secret was proposed [8]. To our knowledge, this is the first work that subgraph queries are protected, whereas the data graph is publicly known.

Subgraph isomorphism. Ullmann [27] proposed a seminal algorithm for subgraph isomorphism. The basic idea is a search with backtracking with respect to the matrix that represents possible isomorphic relationships. In the last decade, several algorithms (e.g., VF2 [7], QuickSI [25] and Turboiso [14]) have been proposed to enhance performance significantly. They all require to traversing the query on graph data. For instance, VF2 [7] relies on a set of state transitions and traversals on the graph and query. QuickSI [25] optimizes the ordering in traversals of graphs. Turboiso [14] exploits neighborhood information and local regions of vertices. Turboiso also involves determining an optimal traversal in query processing. However, the traversals themselves carry topological information, which makes privacy preservation complicated if it is possible at all. Wu et al. [30] supports structureless graph queries, as the query structure is automatically formulated. However, the queries are known to the SP.

X. Conclusion

This paper studies the first practical private approach for subgraph query service: asymmetric structure-preserving subgraph query processing. Our techniques include deriving minimized candidate subgraphs to significant reduce the number of candidate mappings, generating candidate matchings and then candidate mappings without redundancies and verifying candidate mappings without leaking query structures. Our experiments confirm that our techniques are efficient and effective. A future work is to support data values associated with the graphs. We also plan to apply distributed computing once candidate subgraphs are generated.

Acknowledgments. This work was partially supported by the HKBU grants FRG2/12-13/079 and FRG2/13-14/073, and GRF HKBU12200114.

REFERENCES