Towards Efficient Authenticated Subgraph Query Service in Outsourced Graph Databases

Zhe Fan, Yun Peng, Byron Choi, Jianliang Xu, Sourav S Bhowmick

Abstract—Graphs have been a powerful tool that is suitable for a large variety of applications including chemical databases and the Semantic Web, among others. A fundamental query of graph databases is subgraph query: given a query graph \( q \), it retrieves the data graphs from a database that contain \( q \). Due to the cost of managing massive data coupled with the computational hardness of subgraph query processing, outsourcing the processing to a third-party service provider is an appealing alternative. However, security properties such as data integrity and the response time are critical Quality of Service (QoS) issues in query services. Unfortunately, to our knowledge, authenticated subgraph query services have not been addressed before. To support the service, we propose Merkle IFTree (MIFTree) where Merkle hash trees are applied into our Intersection-aware Feature-subgraph Tree (IFTree). IFTree aims to minimize I/O in a well-received subgraph query paradigm namely the filtering-and-verification framework. The structures required to be introduced to verification objects (VOs) and authentication time are minimized. Subsequently, the overall response time is minimized. For optimizations, we propose an enhanced authentication method on MIFTree. Our detailed experiments on both real and synthetic datasets demonstrate that MIFTree is clearly more efficient than a baseline method.

Index Terms—13.0.I Security Concerns of Service-Oriented Solutions, 13.II.I Service-Oriented Security Enablement at Software Level, Subgraph Query Service, Query Answer Authentication, Outsourced Graph Databases

1 INTRODUCTION

There have been a wide range of emerging applications of graph databases, including bio-informatics, chemical-informatics, and web topology [6], [23], [24], whose data are modeled as graphs. To retrieve graphs from large graph databases, many structural queries have been proposed. Among others, subgraph isomorphism query (or simply subgraph query) (e.g., [3], [9], [12], [28]–[30], [32], [36], [37]) has been a fundamental and popular query. Specifically, given a query graph \( q \) and a graph database \( G \), retrieve all graphs in \( G \) that contain \( q \) as a subgraph\(^1\). For example, in biology, there are more than 1,500 online molecular biology databases [6]. In chemistry, PubChem [23] provides public access to numerous chemical compounds. Users can query compounds containing their structures via its web interface.

Due to the cost of hosting the explosive volume of data and performing large-scale computations, the owners of graph databases may not always have the necessary IT infra-structure and expertise to provide the best usage of their data. An appealing solution to address this issue of managing voluminous data is to outsource the owners’ data to a third-party service provider (e.g., Amazon EC2 and Google Cloud Service). Then, the service provider provides query services on the data owners’ behalf. For instance, according to [23], PubChem has managed 19 million unique compound structures. PubChem allows laboratories to submit their data [22]; and PubChem manages the data on the laboratories’ behalf. In addition to PubChem, in drug engineering, many commercial service providers (e.g., [1], [2], [11]) support outsourcing of pharma databases owned by laboratories. Laboratories then focus on the curation of their data.

Security properties such as data integrity are listed as Quality of Service (QoS) issues [20] in (query) services. A reason is that the service provider may be untrusted and/or compromised to attacks and clients may receive tampered results. For instance, Fig. 1 shows an example of a graph database \( G \) and a query graph \( q \). Suppose the service provider stores \( G \) and its index and the client retrieves graphs that contain \( q \). Suppose \( g_4 \) is the answer graph. However, the service provider might return incorrect results, e.g., \( g_1 \), simply abort the computation or return partial answers as some queries may in fact take long to evaluate. In this scenario, the owner/client may never be sure whether the data was outsourced correctly. In practice, the query can be some sensitive chemical compound such as benzopyrene, a carcinogenic substance recently found in some ramen. A compromised service provider might collaborate with some ramen companies and conveniently skip their ingredients that contain benzopyrene. Another important

\(^1\) There are two streams of research work on subgraph queries [8]. One stream handles a very large graph. The other stream concerns a large number of small graphs, which is the focus of this study.
attribute of QoS is the response time of a service. In this paper, it consists of the times for query processing, data transmission and authentication of query results. These two attributes of QoS significantly influence the practicality of outsourcing graph databases. Hence, there is a need for efficient query authentication framework to support the subgraph query services.

Majority of existing querying or indexing algorithms for subgraph queries adopt a filtering-and-verification framework consisting of two key steps. (1) In the filtering phase, the query is decomposed into a set of individual features and an index is searched with those features. The search of each individual feature yields a set of graphs (represented by graph IDs) containing this searched feature. The sets of graphs are intersected to form a candidate set (a superset of answers). (2) In the verification phase, each graph in the candidate set is checked by an exact subgraph isomorphic algorithm to compute the final result set. However, to the best of our knowledge, none of the existing subgraph querying works addresses authentication of such a framework. In this paper, we take the first step towards this goal.

In a typical query authentication system, a data owner publishes his database and signature. A service provider processes queries from a client and transmits to the client both the answer and a verification object (VO) which stores the processing traces such as index traversals and; By using the answer and VO, the client synthesizes the digest of the database/index and compares it with the data owner’s signature to verify the authenticity of the answer.

As the filtering-and-verification framework is not specially designed for authentication, we note at least three problems that may cause large VO to be transmitted to clients and inefficient authentication at clients. Firstly, query features must be authenticated to ensure the correct graph IDs are fetched and intersected. The more the query features, the larger the VO. Unfortunately, none of the previous work minimizes the number of query features used in query processing. Secondly, all graph IDs involved in the intersections must be represented in the VO so that the client can efficiently and correctly verify the intersections. Thirdly, the answer graphs do not generally form a range. In the worst case, each answer graph is authenticated separately. This makes direct applications of classical techniques (e.g., MHT [21] or signature chaining [25]) inefficient. Observe that both the query features and their graph IDs (described in the first two problems) dominate the I/O of the filtering phase and therefore, the problem of minimizing VO is directly related to minimizing I/O of the filtering-and-verification framework.

In this paper, we propose a novel authentication-friendly index called Intersection-aware Feature-subgraph Tree (IFTree) to address the aforementioned technical challenges. We then apply MHTs to IFTree called Merkle IFTree for efficient authentication. Specifically, for the first problem, in order to minimize the number of features used in the filtering phase, we propose a novel higher-order feature called Partially Overlapping Features (POF) which are themselves features composed by individual features. We propose to decompose a query into an optimal POF set such that fewest POFs (i.e., fewest intersections) are used in querying time and meanwhile, more individual features are implicitly used in the filtering phase. As a result, fewer graph IDs are fetched while the candidate set is minimized. As we shall see later, the number of fetched graph IDs in query processing on IFTree is 5 times smaller than that of a baseline. Moreover, the size of candidate set using IFTree is around 25% smaller than that of a baseline. Consequently, the VO size and authentication time are reduced by a factor of 3.6 and 3.3, respectively. For the second problem, we propose a compact matrix representation of intersection of graph IDs on MIFTree to form an enhanced authentication. Our experiments show that the compact representation improves the VO size and the authentication time by a factor around 2.5 and 3.4 (respectively). For the last problem, we determine the optimal ordering of graphs that are “intersect-able”. Our empirical study demonstrates that graphs needed to be authenticated form the fewest number of ranges and the corresponding VO size is reduced by around 40%.

We observe that the overall improvement of the response time over the baseline is often more than an order of magnitude. We show that the energy saving on smartphone by using our proposed techniques is about 27% over the baseline.

In summary, the contributions are listed as follows.

- We propose a novel higher-order feature, called partially overlapping feature for indexing graphs. We leverage these features to propose a novel index, namely Intersection-aware Feature-subgraph Tree (IFTree). For basic authentication, we apply MHTs to various structures of IFTree called MIFTree.
- We propose a novel matrix representation of intersection of graph IDs for enhanced authentication.
- We cluster the graphs that are “intersect-able” by adopting approximation algorithms.
- We conduct extensive experiments with real and synthetic datasets to demonstrate the effectiveness and superiority of our proposed methods.

The rest of the paper is organized as follows. Sec. 2 discusses related works. We present the preliminaries and overview in Sec. 3. We present partially overlapping feature in Sec. 4. We propose IFTree and its query processing in Sec. 5. We propose Merkle IFTree and a basic authentication in Sec. 6. Sec. 7 proposes an enhanced authentication and the optimal ordering of graphs. Sec. 8 presents a detailed experiment. Sec. 9 concludes this paper. We present all the detailed proofs in Appendix A.

2 RELATED WORK

Although there are several efforts in the literature on query authentication for relational and range queries [17], [25], stream queries [27], [34], spatial queries [33], XML queries [5], text search [26], and multi-dimensional queries [4], very few work focus on authentication of graph query processing. Yiu et al. [35] propose authentication of shortest path queries on road networks. However, the ordering of objects in road networks can be determined offline, e.g., by network-based distance. Such ordering is absent in graph databases in general and it is not clear how to adopt this work to subgraph queries. Kundu et al. [13]–[16] propose a series of methods for a
closely related problem. They verify the authenticity of a given portion of data (subtree/subgraph that users’ have the right to access to) without any leakage of extraneous information of the data (tree/graph/forest). They optimize the signature needed and recently propose a scheme that uses one signature [13], [14]. However, in our problem setting, the portion of the data retrieved is the answer of a client’s query, which is yet to be processed by an untrusted service provider. Therefore, the client is required to authenticate both the soundness and completeness (see SubSec. 3.3) of the portion of retrieved data. Search DAGs (Directed Acyclic Graph) [19] is a generalized model for authenticating a large class of data structures, e.g., binary trees, multi-dimensional range trees and tries. However, subgraph query processing can hardly be efficiently cast into a DAG search.

A large number of indexing techniques have been proposed for evaluating subgraph queries. These efforts can be roughly classified into two approaches, namely feature-based (e.g., [3], [12], [28], [29], [32], [36]) and non-feature-based approaches (e.g., [9], [37]). Examples of features are frequent subgraphs, using tools such as gSpan [31] and CAM code [10]. Recently, iGraph [8] implemented these techniques on a common platform and reported that former approaches outperform latter approaches in most cases. Hence, we adopt the feature-based approach in our study.

3 BACKGROUND AND OVERVIEW

In this section, we first discuss the background to subgraph query processing and query authentication. We then formulate the problem studied. A baseline approach and an overview of our solution are discussed.

3.1 Background for Subgraph Query

This paper assumes undirected labeled connected graphs. For simplicity, we may use the term graphs to refer to them. A graph is denoted as $g = (V,E,\Sigma,l)$, where $V(g)$, $E(g)$, $\Sigma$ and $l$ are the sets of vertices, the set of edges, the set of labels of vertices and edges and the function that maps a vertex or edge to a label, respectively. We use $|g|$ to denote the size of graph $g$, where $|g| = |E(g)|$. Following the literature of a popular stream of graph databases [3], [9], [12], [28], [29], [32], [36], [37], we consider graphs of modest sizes.

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

- $\forall u \in V(g), \varphi(u) \in V(g'), l(u) = l'(\varphi(u))$; and
- $\forall (u,v) \in E(g), (\varphi(u),\varphi(v)) \in E(g'), l(u,v) = l'(\varphi(u),\varphi(v))$.

Subgraph query can be formally defined in Def. 3.1. We say a graph $g$ is a subgraph of another graph $g'$ if there exists a subgraph isomorphism from $g$ to $g'$, denoted as $g \subseteq g'$ or $\text{subIso}(g,g') = \text{true}$. It is known that deciding whether $g$ is the subgraph of $g'$ is NP-hard. Subgraph query processing can be described as follows.

**Definition 3.2:** Given a graph database $G = \{g_1,g_2,...,g_n\}$ and a query graph $q$, we want to determine the query answers $R_q = \{g|\text{subIso}(q,g), g \in G\}$.

**Subgraph query paradigms.** Two query paradigms for subgraph queries have been proposed: feature-based (e.g., [3], [28], [29], [32], [36]) and non-feature-based (e.g., [9], [37]) indexes. From Sec. 2 above, iGraph [8] concludes that the former often outperforms the latter. This work contributes to the feature-based approaches.

A well-received query paradigm for feature-based approaches is the filtering-and-verification framework [3], [12], [28], [29], [32], [36]. Early work on subgraph query processing such as Shasha et al. [29] proposes filtering graphs via paths and then verifying the remaining graphs through $\text{subIso}$. Some later works [3], [12], [28], [29], [32], [36] proposed innovative solutions that follow such a framework. To illustrate the filtering-and-verification framework, we present a seminal index called gIndex [32] which is shown efficient in many cases [8]. gIndex proposes discriminative frequent features as individual features, denoted as $F$, for indexing. A discriminative frequent feature $f, f \in F$, is:

- a subgraph whose size is smaller than or equal to $\text{maxL}$, where $\text{maxL}$ is a user-defined maximum feature size;
- a frequent feature that $|D_f| \geq \text{SISF}(|f|)$, where $D_f = \{f | f \subseteq g, g \in G\}$, $|D_f|$ is called the support of $f$ and $\text{SISF}$ is a user-defined $\text{Size-Increasing-Support-Function}$ of each $f$; and
- discriminative, s.t., $\frac{|\cap f \subseteq D_f|}{|D_f|} \geq \text{dr}$, where $\text{dr}$ is a user-defined discriminative ratio.

The function $\text{SISF}$ returns a support that increases with the input feature size. gIndex sets $\text{SISF}(1) = 1$ by default. $\text{SISF}$ gives the flexibility to allow indexing with infrequent features.

The individual features are represented by a canonical string called minimum DFS code [31] and gIndex is a prefix tree of the minimum DFS codes. gIndex processes queries in two phases. (1) **Filtering:** enumerate the maximum individual feature set $F_q$ from $q$, where $F_q = \{f | f \subseteq q, f \in F, \#f, s.t., f \subseteq f', f' \subseteq q\}$, and filter out the graphs that do not contain a feature in $F_q$ to obtain the candidate set $C_q$ by performing the following intersections:

$$C_q = \bigcap_{f \in F_q} D_f$$

(1)

(2) **Verification:** determine the query answers $R_q$ from the candidate set by invoking $\text{subIso}$, where $R_q = \{g|q \subseteq g, g \in C_q\}$.

It is worth noting that the intersections in the filtering phase are performed on graph IDs whereas $\text{subIso}$ in the verification phase is invoked with graph data. Therefore, all
previous indexes (see [3], [12], [28], [29], [32], [36]) propose innovative ideas to filter more non-answer graphs that aim to minimize the candidate set $C_q$.

**Example 3.1:** We illustrate the filtering-and-verification framework with an example in Fig. 2. The upper half of Fig. 2 shows the gIndex constructed from a set of individual features mined from $G$ in Fig. 1, $F = \{f_1, f_2, \ldots, f_4\}$, where SISF(1) = 1, SISF(2) = 2, maxL and dr are set to 2 and 0.1, respectively. $f_i$ is an artificial root node. The lower half of Fig. 2 shows its query processing: Given a query graph $q$, the filtering phase first enumerates all the maximum individual features $F_q = \{f_3, f_4\}$ of $q$ and performs intersections of the graphs (via IDs) containing the individual feature(s) ($D_{f_3}$ and $D_{f_4}$) to compute the candidate set $C_q = D_{f_3} \cap D_{f_4} = \{g_1, g_4\}$. The verification phase invokes subTese on each graph in $C_q$, and computes the answers $R_q = \{g_4\}$.

### 3.2 Background for Query Authentication

**Cryptographic primitives.** Similar to other works on authentication, we assume a **one-way collision-resistant hash function** (e.g., SHA and MD5) is denoted as $h(x)$, where $x$ is a data value to be hashed and the hash value $h(x)$ is often referred to as the **digest** of $x$. It is infeasible to determine the preimage of a digest. We assume a **public-key digital signature scheme**, such as RSA, that guarantees the authenticity of a message or value. The signer has a private key (SK) and can produce a signed message $y = sign(x, SK)$. Any public user has a public key (PK) and can verify the message by decryption.

**Merkle Hash Tree (MHT).** The Merkle Hash Tree [21] is a classical authentication technique. The main idea of MHT is illustrated with an example shown in Fig. 3(a). It is a classical MHT built on data values $\{x_1, \ldots, x_4\}$. Each leaf node is associated with the digest (hash) of its data value, e.g., $H_{x_1} = h(x_1)$. Each internal node contains the digest of the concatenation of the digest of its child nodes, e.g., $H_{x_1, x_2} = h(H_{x_1} | H_{x_2})$. A data owner signs the digest of the root node.

To authenticate a data value, e.g., $x_2$, the service provider sends to the client $x_2$ and a $VO$ that consists of the digests $H_{x_1}$ and $H_{x_3, x_4}$ and the signed root digest of $H_{r}$. The client computes from the $VO$, $H_{x_2} = h(x_2)$, $H_{x_1, x_2} = h(H_{x_1} | H_{x_2})$, and finally the root digest $H_{x_1, x_2} = h(H_{x_1} | H_{x_3, x_4})$. The client uses the data owner’s public key to compare $H_{x_1, x_2}$ and the signed root digest. If they agree, $x_2$ has not been tampered with. MHT can be extended to authenticate a set of data values.

MHT has been generalized to a **multi-way index** (such as Merkle B-tree [17]) for database applications. Moreover, it has been **embedded** into index nodes (see the Embedded Merkle B-tree (EMB-tree) [17]) to minimize $VO$ sizes. Fig. 3(b) shows a search tree embedded with an MHT. The data in the MHT are $\{x_1, \ldots, x_4\}$, the search keys are $\{1, 2, 3, 4\}$.

- Each leaf node is associated with the search key and the digest (hash) of its data value, e.g., $(1, H_{x_1})$ where 1 is the search key of $x_1$; and
- Each internal node contains the search key and the digest of the concatenation of the digest of its child nodes, e.g., $(2, H_{1,2})$ where $H_{1,2} = h(h(h(1) | H_{x_1}) | h(h(2) | H_{x_2}))$.

**Fig. 3.** Two kinds of Merkle Hash Trees used

Suppose that the search of the key 2 needs to authenticate. The $VO$ contains $(1, H_{x_1})$ and $(4, H_{x_4})$, and the data owner’s signature on the root digest $H_r$. The client composites $H_{x_2} = h(x_2)$, $H_{1,2} = h(h(h(1) | H_{x_1}) | h(h(2) | H_{x_2}))$, and finally the root digest $H_r = h(h(4) | H_{x_4})$. Similarly, by comparing the synthesized root digest and the data owner’s signature, the client verifies the authenticity of the data from the service provider. From the boundaries (i.e., 1 and 4) of the search, the client verifies that the search is correct.

In this paper, we apply both kinds of MHTs (Figs. 3(a) and (b)) to various structures of our index to minimize $VO$.

### 3.3 Problem Formulation

**System Model.** The system model follows the existing authentication framework, that comprises three parties — (i) **data owner DO**, (ii) **service provider SP** and (iii) querying client.

(i) The DO owns a graph database $G$. The DO or SP first generates an index to support subgraph query processing. Then, DO signs the root digest of the index. (ii) The SP receives a query $q$ from a client, processes it on behalf of DO and returns the answer graph $R_q$ to the client. Since SP may not be trusted, it is required to return not only $R_q$ but also a $VO$ and the DO’s signature to the client. (iii) Upon receiving the $VO$, the client verifies the $R_q$ the SP returns. We assume the client has the public key of the DO for authentication. In particular, the client verifies the following:

- **Soundness:** all graphs in $R_q$ are answers and they are not tampered with, i.e., $\forall q \in R_q$, $g \in G \land q \subseteq g$; and
- **Completeness:** there is no graph that is not in $R_q$ but is an answer, i.e., $\\exists g \not\in R_q, g \in G \land q \subseteq g$.

**Threat Model.** In our system model, the SP may not always be trusted. It may be a potential adversary or have been tampered with by attackers. In either case, we assume that the SP may alter the graph data or the index structure, introduce wrong answers, skip certain answers or abort the computation. An authentication framework is considered secure if attacking it under this threat model is as hard as inverting a one-way collision-resistant hash function.

Given the above preliminaries, we are ready to formally present the problem statement.

**Problem statement.** Given the above system and threat models, we seek an efficient authentication framework where the client may submit a subgraph query and verify the soundness and completeness of the answers returned by the service provider.

### 3.4 Baseline Authentication — MIndex

In this subsection, we derive a baseline technique from gIndex. We sketch the main ideas of this naïve authentication approach and discuss the drawbacks of such an approach. For a concise exposition, we present the details in set semantics,
Overview of our Approach

In response to the drawbacks of the baseline approach, we propose more efficient authentication techniques. The frequently used symbols of our discussions are listed in Appendix E.

Example 3.2: We use Example 3.1 to illustrate the \( \mathcal{VO} \).

1. \( \mathcal{VO}_{\text{index}} \) contains the digests that record the search of \( F_q = \{ f_3, f_5 \} \). Suppose the search locates \( f_3 \) first. The \( \mathcal{VO}_{\text{index}} \) includes the digests of the nodes \( f_1, f_3, f_4, \) and \( f_5 \). The digest of node \( f_2 \) is computed by the client. When the search locates \( f_3 \), the digest of \( f_3 \) in \( \mathcal{VO}_{\text{index}} \) is replaced by the actual content of the node \( f_3 \). Thus, the client can verify \( f_3 \). The graph IDs for each graph in \( D_f \) and \( D_{f_3} \) (i.e., \( \{1, 2, 4\} \) and \( \{1, 4\} \), respectively), and the hash value of \( g_2 \) are added to \( \mathcal{VO}_{\text{index}} \); and
2. \( \mathcal{VO}_{C_q} \) contains the non-answer graphs in the candidate set, i.e., \( \mathcal{VO}_{C_q} = C_q - R_q = \{ g_1 \} \), where \( C_q = \{ g_1, g_4 \} \) and \( R_q = \{ g_4 \} \).

Regarding the authentication at the client side, firstly, the client rebuilds the root digest of \( \text{MgIndex} \) using \( \mathcal{VO}_{\text{index}} \) and \( \mathcal{VO}_{C_q} \) to verify that \( F_q \) and \( C_q \) are not tampered with. Secondly, it enumerates the query again to verify that \( F_q \) is exactly \( f_3 \) and \( f_7 \) by using \( \mathcal{VO}_{\text{index}} \). Thirdly, the client performs intersections on \( \{1, 4\} \) and \( \{1, 2, 4\} \) to verify the correctness of \( C_q \). Finally, the client performs the subIso tests to verify \( g_1 \) is the answer but not \( g_3 \).

The sketch of the baseline approach reveals the performance bottlenecks of subgraph query authentication. The more features (i.e., more intersections) are used to determine \( C_q \) (Formula 1), the more \( \mathcal{VO}_{\text{index}} \) is needed to authenticate \( F_q \) and the more graph IDs of \( D_f \) are introduced. This not only leads to large \( \mathcal{VO} \), but also requires high time costs to authenticate them. Similar to query processing, query authentication also requires to minimize \( C_q \) as the non-answer graphs (not the IDs) are included in \( \mathcal{VO}_{C_q} \).

3.5 Overview of our Approach

In response to the drawbacks of the baseline approach, we propose more efficient authentication techniques. The frequently used symbols of our discussions are listed in Appendix E.
4.1 Types of Overlapping Features

Features can be composed in various ways. We derive POFs and call them higher-order features as they themselves are features and composed by individual features.

To describe POFs, we first present a few notations needed: Individual features $F$ can be features proposed by any existing works. We adopt discriminative frequent feature [32] as the individual feature in this paper. We use $g$ and $F_g$ to denote a graph and its individual features. We call the subgraph of $g$ that is isomorphic to $f$ as an instance of $f$, i.e., $g \in D_f$. With these notations, we derive POFs. We start with the feature of multiple individual features.

**Definition 4.1:** A feature $\{f_1, \cdots, f_n\}$ is a co-existing feature of $g$ if $g$ contains an instance of $f_i$ for all $i \in \{1, \cdots, n\}$, where $\{f_1, \cdots, f_n\} \subseteq F_g$.

The definition above can be trivially extended to a database $G$. Let $\{f_1, \cdots, f_n\}$ be a co-existing feature of $G$, a graph $g$ in $G$ contains it iff $g \in D_{f_1} \cap \cdots \cap D_{f_n}$.

The next feature, namely overlapping feature, concerns not only the existence of features but also the overlapping of features.

**Definition 4.2:** $\{f_1, \cdots, f_n\}$ is an overlapping feature of $g$ if it is a co-existing feature of $g$ and there is a set $S:\{s_1, \cdots, s_r\}$ in $g$, where $s_i \in S$ is an instance of $f_i$, and $S$ forms a connected graph.

We remark that singleton sets $\{f_1\}$ (i.e., $n = 1$) are considered as “overlapping” features since each of their instance definitely forms a connected graph.

**Example 4.1:** Fig. 5 illustrates Defs. 4.1 and 4.2. In Fig. 5(a), $\{f_4, f_5\}$ is a co-existing feature of $g_1$. In Fig. 5(b), $\{f_2, f_3\}$ is an overlapping feature of $g_1$, as the instances of $f_2$ and $f_7$ not only exist but also overlap.

One may be tempted to derive more sophisticated features, e.g., by exploiting the topology graph of an overlapping feature. However, such features may introduce a high complexity in query processing. In this paper, we adopt overlapping features. Moreover, consider overlapping features e.g., in Fig. 5(b). The instances of $f_2$ and $f_7$ are completely overlapped. In practice, $D_{fp}$ is often a subset of $D_{f_2}$. Indexing graphs with both $f_2$ and $f_7$ are often redundant. Hence, we propose partially overlapping features defined in Def. 4.3. An example is shown in Fig. 5(c).

**Definition 4.3:** $\{f_1, \cdots, f_n\}$ is a partially overlapping feature (POF) of $g$, if (1) it is an overlapping feature of $g$ and (2) there does not exist $f_i, f_j \in p$, s.t., for each instance $s_i$ of $f_i$ and $s_j$ of $f_j$, $s_i$ is completely overlapping with (i.e., contained in) $s_j$.

Singleton sets are considered POFs since (1) they are special cases of overlapping features and (2) no two features whose instances are completely overlapping. This subtle case has a practical implication: Clients may issue queries with exactly one feature and it may be indexed.

**Fig. 5.** Illustration of higher-order features

5 Intersection-aware Feature-subgraph Tree (IFTree)

In this section, we present IFTree that indexes a graph database $G$ with all POFs that satisfy maxSup and minSup. We present the querying processing of IFTree, which is authenticated in Sec. 6.

5.1 IFTree

IFTree is a prefix tree on POFs where each node represents a POF and points to a list of graph IDs. Recall from Def. 4.2 that each POF is a set of individual features. The subset operator $\subseteq$ over all the POFs is a partially ordered set. To derive a search tree on the set, we assume that each individual feature has an ID and a POF $p$ is represented by a string of IDs of its individual features sorted in ascending order. The string $\text{str}(p)$ to denote the string of $p$. For example, let $p = \{f_1, f_2\}$. An example in a database $G$ is shown in Def. 4.3. An example is shown in Fig. 5(c).

**Definition 4.4:** The constraint of POFs $P$ is $(\text{maxSize}, \text{minSup})$, where $\text{maxSize}$ and $\text{minSup}$ are the minimum size and the minimum support of $P$ in a database $G$, i.e., $\forall p \in P, \text{str}(p) \leq \text{maxSize}$ and $|\text{str}(p)| \geq \text{minSup}$.

The number of all POFs of a database $G$ is exponential to the number of features in worst case. In practice, many POFs do not have sufficient support. We adopt an enumeration algorithm to compute all POFs that satisfy the user-defined $(\text{maxSize}, \text{minSup})$.

It is worth mentioning that the graphs indexed by a POF $p : \{f_1, \cdots, f_n\}$ (denoted as $D_p$) are a proper subset of the graphs in $D_{f_1} \cap \cdots \cap D_{f_n}$. Indexing with $p$ may be viewed as precomputing the intersections. In the rest of the paper, we use the term features $P$ to refer to POFs, whereas $f_1, \cdots, f_n$ are referred to individual features.

**Fig. 6.** Illustration of IFTree (partial)
Algorithm 1 Query Processing ($q$, $G$, $T_F$, $T_P$)

**Input:** A query graph $q$, a graph database $G$, the prefix tree $T_F$ of features $F$ and the $IFTree$ of $G$

**Output:** the answer set of $q$ $R_q$

1. Initialize $R_q$ to $\emptyset$ and $C_q$ to $G$
2. $F_q = \text{find max features}(q, T_F)$ // Full cover $q$
3. $P_1 = \text{find POF}(q, F_q)$ // Enumeration
4. $P^{opt}_q = \text{opt POF}\_\text{MWSC}(P_1, P_q)$
5. for each $p \in P^{opt}_q$
6. $D_p = \text{search}(p, T_P)$ \quad $C_q = C_q \cap D_p$
7. for each $g \in C_q$
8. if $\text{subIso}(q, g) = \text{true}$ then $R_q = R_q \cup g$
9. return $R_q$

**Example 5.1:** Fig. 6 shows the $IFTree$ of the POFs $P$ of $G$ of Fig. 1. Due to space constraints, we skip the enumeration that yields $P:\{p_1 \cdots p_{15}\}$. Each box of the tree represents a POF. The constraint of POF (maxSize, minSup) is (2, 2). Consider $p_9$. The $IFTree$ has an edge between $p_1$ and $p_9$ but not $p_2$ and $p_9$ as $\text{str}(p_2) = "2"$, $\text{str}(p_9) = "1.2"$ and therefore $p_2 \not \approx p_9$.

To illustrate the processing of existing indexes and $IFTree$, let’s assume that a query contains two individual features $f_2$ and $f_7$. $g$Index retrieves and intersects $D_{f_2}$ and $D_{f_7}$ whereas $IFTree$ simply retrieves $D_{p_{11}}$.

### 5.2 Query Processing on $IFTree$

The query processing on $IFTree$ is detailed in Alg. 1. It takes a query graph $q$, a graph database $G$, the prefix tree $T_F$ of features $F$ and the $IFTree$ of $G$ as input. It determines all maximum individual features $F_q$ that fully cover $q$ (Line 2).

From $F_q$, it computes all possible POFs $P_q$ from $F_q$ (Line 3) and determines the optimal POFs $P^{opt}_q$ from $P_q$ (Line 4), which shall be discussed shortly. For each POF $p$ in $P^{opt}_q$, the graphs of $D_p$ are retrieved by searching $IFTree$ and maintained in a candidate set $C_q$ (Lines 5-6). For each graph in $C_q$, the algorithm verifies if it is in fact an answer (Lines 7-8). Following up Example 3.1, we use Fig. 7 to illustrate the query processing on $IFTree$ (shown in Fig. 6) in the following discussion.

It is worth noting that Alg. 1 involves two optimizations. The first one is similar to an existing work [32] — the query $q$ is decomposed into maximum individual features $F_q$ by using an enumeration method. $f$ is maximum in terms of $q$ if and only if there does not exist a larger $f'$ such that $\text{subIso}(f, f') = \text{true}$ and $\text{subIso}(f', q) = \text{true}$. Unlike previous work, we determine $F_q$ that fully covers $q$. When compared to non-covers, a cover $F_q$ is expected to be more selective and yields a small candidate set in the filtering phase. $F_q$ is then used to enumerate POFs, as indicated in the RHS of Fig. 7. For example, as in Example 3.1, $g$Index computes $F_q$ as $\{f_3, f_7\}$. However, Alg. 1 determines $F_q$ as $\{f_2, f_3, f_7\}$ (in Line 2). Without $f_2$, $F_q$ does not fully cover $q$.

The second optimization is that an optimal decomposition $P^{opt}_q$ is determined from $F_q$. In the filtering-and-verification framework (e.g., Fig. 7), graph data are fetched from disk mainly in two steps: (i) when graph IDs of $D_p$’s are fetched from disk for performing intersections; and (ii) when candidate graphs are fetched for $\text{subIso}$ tests. This leads to two

---

2. Some pseudocode in Alg. 1, e.g., Lines 1 and 2, are straightforward but verbose. Hence, for concise presentation, we present their main ideas in text.
Since $f_2$ and $f_7$ are covered by $p_{11}$, they are removed from $F_q$ and the weights $w_2, w_3$ and $w_7$ are updated accordingly. In the second iteration, $p_3$ is chosen. All features in $F_q$ are covered and the algorithm terminates. $F_q^{opt}$ is $\{p_3, p_{11}\}$.

6 Merkle IFTree (MIFTree)

Thanks to the minimization of I/O by using $F_q^{opt}$, the query processing trace needed to be included in $VO$s is reduced when IFTree is adopted for query authentication. To facilitate efficient authentication, we propose to apply MIFTree to IFTree to obtain Merkle IFTree (MIFTree). Recall that IFTree is a prefix tree for the string representations of POFs. The index nodes near the root of IFTree often have large fanouts, as those POFs may overlap with many other individual features to form larger POFs. Therefore, an MHT is embedded to the children of each index node to minimize $VO$. In addition, in practice, some POFs may index a large number of graphs. For instance, in the dataset AIDS, the number of graphs containing the POF of an index node near the root of the IFTree is 12% of the total number of graphs. When some of these graphs are selected into the candidate set in the filtering phase, a classical MHT is needed to efficiently authenticate these graphs. Hence, we propose the Merkle IFTree (MIFTree) as follows.

Definition 6.1: MIFTree is an IFTree extended with two kinds of MHTs: (i) An MHT is embedded to the child nodes of each node$(p)$ of MIFTree; and (ii) A classical MHT is built on top of all graphs (with graph IDs) in $D_p$ for each node$(p)$.

The rest of this section describes the signing of MIFTree in detail and a basic authentication of MIFTree.

6.1 Signing MIFTree

Similar to the majority of search trees for query authentication, we associate hash values/digests to the nodes of IFTree. The data owner $DO$ signs the root of the digest of MIFTree. Specifically, we formalize the digests and signatures of MIFTree below.

Definition 6.2: The digest of a data graph $g_i$ is defined as $\mathcal{H}_g_i = h(\text{mindfs}(g_i))$.

Graphs are cast into some (publicly known) canonical representation before their digests are computed. In this paper, we adopt the minimum DFS code [31] denoted as mindfs, but other representations may also be adopted.

Definition 6.3: The digest of a node node$(p)$ of MIFTree is $\mathcal{H}_p = h(h(\text{str}(p))|\mathcal{H}_{D_p}|\mathcal{H}_p^r)$, where

- $\text{str}(p)$ is the string of $p$;
- $\mathcal{H}_{D_p}$ is the root digest of the classical MHT of $\text{ID}(D_p): \{j_1, \ldots, j_m\}$. The data in the MHT are $\{(j_1, g_{j_1}), \ldots, (j_m, g_{j_m})\}$; and
- $\mathcal{H}_p^r$ is the root digest of the embedded MHT of node$(p)$’s children. The data in the MHT are $\{\text{node}(p_1), \ldots, \text{node}(p_m)\}$ and the search keys are $\{p_1, \ldots, p_m\}$, where node$(p_1), \ldots, \text{node}(p_m)$ are the children of node$(p)$.

3. Due to space constraints, we have to omit the details of mindfs. As an example, mindfs($g_3$) = $(1,2,C,C), (2,3,C,C), (2,4,C,C), (2,5,C,C)$. The first two digits are the DFS sequence of the vertices of a graph. The following characters are vertices’ labels.
Algorithm 2 Auth Query Processing $(q, G, T_F, T_p, \psi)$

**Input:** A query graph $q$, a graph database $G$, the prefix tree $T_F$ of features $F$, the MIFTREE $T_p$ of $G$ and $\psi$.

**Output:** The answer set of $R_q$ and verification object $V_O$.

1. Initialize $R_q$ and the structures in $V_O$ to $\emptyset$ and $C_q$ to $G$
2. $F_q = \text{find_maxfeatures}(q, T_F)$ //F_q fully cover q
3. $P_q = \text{find_POF}(q, F_q, T_p)$ //Enumeration
4. $P^\text{opt} = \text{opt_POF_MHT}(F_q, P_q)$ //\* construct $V_O$ of Case 1 of $p$ of $N_I$ */
5. for each $p_i \notin P^\text{opt}$ and $p_i \in P_q \cup \{p_v\}$
6. $N_j = N_j \cup \{p_i, \mathcal{H}_{p_i}\}$ //app and digest */
7. $N_j \cup b_i$ //\* the $V_O$ of MHT of node($p_i$)'s children */
8. for each $p_i \in P^\text{opt}
9. $D_p_i = \text{search}(p_i, T_F), C_q = C_q \cap D_p_i$
10. //\* construct $V_O$ of Case 2 of $p$ of $N_I$ */
11. $L_{p_i} = []$
12. for each $g_j \in D_p_i$
13. \* if $g_j \in C_q$ then $L_{p_i} = L_{p_i} + j$ //\* append ID */
14. else $L_{p_i} = L_{p_i} \cup \{j, \mathcal{H}_{g_j}\}$ //\* append ID and digest */
15. $N_j = N_j \cup \{p_i, L_{p_i}\}$ //\* construct V_O for features $F_q$ */
16. $N_j = N_j \cup b_i$ //\* construct $V_O$ for non-answer */
17. $\mathcal{M} = \text{compute}(F_q)$
18. for each $g \in C_q$
19. if subIso($g, q, \psi$) = true
20. $R_q \cup q \cup g$
21. //\* construct $V_O$ for answer */
22. $\mathcal{M} = \mathcal{M} \cup m$, where $m$ is the mapping from $q$ to $g$.
23. else \* construct $V_O$ for non-answer */
24. $C_q = C_q \cap q$
25. return $R_q$ and $V_O$

For example, recall from Sec. 3 that Fig. 3(b) shows an embedded MHT where \{x_1, x_2, x_3, x_4\} are data values and \{1,2,3,4\} are the search keys. The search of the key is 2 and the answer is x_2. The $V_O$ of the MHT are (1, x_2) and (4, x_3, 4), with which $H_c$ can be synthesized.

**Definition 6.6:** The $V_O$ construction of basic authentication for subgraph query is a tuple ($V_O$\_index, $V_O$\_cq), where $V_O$\_index = \{N_I, N_F, \psi\}:

- $N_I = (N_I^v, N_I^p)$ is the digest of MIFTREE nodes, where $N_I^v = \{n_1, n_2, \ldots, n_m\}_v$, where $P_q = \{p_1, \ldots, p_m\}$
- $N_I^p = \{p_1, \ldots, p_m\}$
- $N_F = \{F_1, F_2, \ldots, F_q\}$ is similar to $N_I$.
- $F_q$ is also organized in a prefix tree $T_F$ ordered by the $\mathcal{M}_I$s. The only difference from MIFTREE is that each node of the $T_F$ points a feature but not a list of graph IDs; and
- $\psi = \{\psi_F, \psi_p\}$ is the signature of the $DO$.

$V_O$\_cq = (I_M, C_q^R)$:

- $I_M = \{i_1, \ldots, i_n\}$ is a set of subgraph isomorphism mappings from $q$ to $R_q$;
- $C_q^R$, $\mathcal{M}_I$ = $\emptyset$.

**$V_O$ Construction:** The $V_O$ of a query is constructed by Alg. 2 at the $SP$ side. Alg. 2 is Alg. 1 extended with $V_O$.

**For construction:** Lines 5-7 and 10-16 for $N_I$, Line 17 for $N_F$, and Lines 21-23 for $V_O$\_cq. The extension of $\text{find_maxfeatures}$ with $V_O$ construction is presented in Appendix B and that of $\text{find_POF}$ is similar (Lines 2-3). As in Alg. 1, to evaluate $q$, Alg. 2 determines $P^\text{opt}$ from $q$ and $T_F$ (Lines 2-4). In Lines 5-7, for each $p_i$ in $P_q$ or $P_p$ but not in $P^\text{opt}$, it includes ($p_i, H_{p_i}$) in $N_I^v$ (Case 1 of Def. 6.6) and $b_i$ in $N_I^p$, where $b_i$ is the $V_O$ of MHT of node($p_i$)'s children. A subtree is marked that node($p_i$) is the root of MIFTREE and it is always visited and considered in $N_I^v$. $C_q$ is computed in Lines 8-9 (same as Lines 5-6 in Alg. 1). Then, in Lines 10-14, for each $p_i$ in $P^\text{opt}$ (Case 2 of Def. 6.6), and for each $g_j \in D_{p_i}$, if $g_j$ is in $C_q$, it adds $j$ to $L_{p_i}$; otherwise, (j, $H_{g_j}$) to $L_{p_i}$. The $V_O$ for $p_i$ is added to $N_I^v$ (Line 15). The construction of $N_I^v$ in Line 16 is the same as that of in Line 7. In Line 17, $N_F$ for $F_q$ is constructed similar to $N_I$, as both $F$ and POFs are indexed by prefix trees. Regarding $V_O$\_cq, in Lines 18-23, if a graph $g$ in $C_q$ is an answer, its mapping between the query is added to $I_M$; otherwise, $g$ is added to $C_q^R$. The overall $V_O$ is constructed and returned to the client (Lines 24-25).

**Example 7.6:** Following up the query processing shown in Fig. 7, Fig. 9 shows the $V_O$ determined by Alg. 2. Recall that $P_q = \{p_2, p_3, p_7, p_11\}$ and $P^\text{opt} = \{p_3, p_11\}$. Regarding $V_O$\_index, $N_I^v = \{n_1, n_2, n_3, n_4\}_v$, $n_1 = (p_1, H_{p_1}), n_2 = (p_2, H_{p_2}), n_3 = (p_3, H_{p_3}), n_4 = (p_4, H_{p_4})$, and $N_I = N_I^v \cup \{p_i, H_{p_i}\}$ for $i = 1, 2, 3, 4$. Since $p_1, p_2 \notin P^\text{opt}$, $I_M = \{\emptyset\}$. $F_q \cap C_q = \emptyset$. Since $C_q = R_q \cap \{g_1\}$, $C_q^R$ is empty.
2) compute $P_q$ and verify $P_q$ is consistent to those in $N_f$ by using $q$, $F_q$ and $N_f$;
3) determine $I_q^{opt}$ by using $F_q$ and $P_q$;
4) synthesize $H_p$, by using $P_q^{opt}$ and the VO;
5) verify the $H_{p_r}$ with the signature $\psi_p$ and the public key;
6) determine $C_q$ by intersecting the $L_s$ from $N_f^7$, where $p \in P_q^{opt}$; and
7) verify $R_q$ by using $I_M$; and if $I_M$ is not correct, invokes $\text{subISO}$; and verify $C_q^{R_k}$ by invoking $\text{subISO}$.

In Step 4), the root digest $H_{p_r}$ is synthesized bottom-up: We start the synthesis from the $p$ in $P_q$ that do not have a $p' \in P_q$ s.t. $p < p'$. At each synthesis step, $p_i$ can only be in one of the two cases: Case 1 $p_i$ is in $P_q$ but not in $P_q^{opt}$, $n_i$ of $p_i$ is $(p_i, H_{D_{p_i}})$. Case 2 $p_i$ is in $P_q^{opt}$, $n_i$ of $p_i$ is $(p_i, L_{p_i})$. $H_{D_{p_i}}$ is determined from $L_{p_i}$, $C_q^{R_k}$ and $R_q$, which contains the IDs, digests and the graphs of $D_{p_i}$. The remaining part to-be-determined is $H_{p_i}$. In both cases, $H_{p_i}$ is determined from the VO of MHT of node($p_i$)'s children, in $N_f^7$. The synthesis must have computed the digest of node($p_i$)'s children (if it is not already in VO), as the synthesis is defined bottom-up. With $p_i$, $H_{D_{p_i}}$, and $H_{p_i}^{r_i}$ (Def. 6.3), the client can recompute $H_{p_i}$. Then, $p_i$ is removed from $P_q$. In the recursive step, the synthesis proceeds to another $p$ in $P_q$ with no $p' \in P_q \land \lhd p'$. With the $H_{p_i}$ for all $p \in P_q$, $H_{p_r}$ is synthesized. We present the formal proofs of the soundness and completeness of the basic authentication in Appendix A.

Example 6.3: To illustrate Step 4), we present the major steps of the synthesis of $H_{p_r}$ of Fig. 9. To compute $H_{p_r}$, bottom-up, we may start the synthesis from $p_{11}$ since $p_{11} \in P_q^{opt}$ and $\overline{p}_{11} \in P_q$ s.t. $p_{11} \prec p_{11}$. We may start at $p_3$ for a similar reason. Let's start at $p_{11}$, $n_{11} = (p_{11}, L_{p_{11}})$. $H_{D_{p_{11}}}$ can be computed from $L_{p_{11}}$ and $R_q$. The root digest $H_{p_{11}}^{r_{11}}$ of the MHT of node($p_{11}$)'s children can be computed since the MHT is empty. $H_{p_{11}}$ can then be determined from $p_{11}$, $H_{D_{p_{11}}}$, and $H_{p_{11}}^{r_{11}}$, $p_{11}$ is removed from $P_q$. After that, we may proceed to $p_2$, since $p_2 \in P_q \land p_2 \notin P_q^{opt}$, $n_2 = (p_2, H_{D_{p_2}})$. We determine $H_{p_2}^{r_2}$ from the computed $(p_2, H_{D_{p_2}})$ and the VO of MHT of node($p_2$)'s children such as $b_2$. In this case, $p_2$ is obtained and $p_2$ is removed from the $P_q$. We then proceed to $p_3$. $H_{p_3}$ is obtained, similar to the synthesis of $H_{p_{11}}$. $p_3$ is synthesized similar to $p_2$. With the same logic, $n_r = (p_r, H_{D_{p_r}})$. With $H_{p_2}$, $H_{p_3}$, $H_{p_3}$ and the VO of MHT of node($p_r$)'s children, $H_{p_r}$ is synthesized.

7 Enhanced Authentication

While the basic method presented in Sec. 6 is natural to authenticate the filtering-and-verification framework of subgraph query, VO sometimes contains excessive graph IDs. In this section, we propose two enhancements on the basic method.

Firstly, all graph IDs of each feature $p \in P_q^{opt}$ are returned and in Step 6) of authentication, intersected at the client side to ensure the correctness of $C_q$. To optimize this, we propose a compact representation of graph IDs. Secondly, graph IDs are needed to synthesize the digests of MHTTrees nodes, as elaborated in Step 4) of authentication. As motivated in Sec. 1, graph IDs of $C_q$ do not fall into a range in general which may lead to large VO$s when classical authentication techniques are adopted. Hence, we propose to cluster graphs with similar feature sets offline. As a result, when a query is retrieved by using a set of features, the IDs of $C_q$ may be clustered and represented by a smaller VO.

7.1 Compact Representation of Graph IDs

The main idea to reduce the excessive graph ID$s for verifying the intersections is to encode all the features of each graph in a $D_p$ in a binary matrix $M_p$. The data owner signs the matrix. Hence, the client requires one ID($D_p$) and $M_p$ to verify the intersections.

Definition 7.1: For each node node($p$), the matrix representation $M_p$ of ID($D_p$) is a $n \times m$ binary matrix, where $n = |D_p|, m = |P|$. $M_p(i, j) = 1$ if $g_j \in D_p$, and $M_p(i, j) = 0$, otherwise.

Next, we build a classical MHT to each $M_p$ (defined with $M_{M_p}$ in Def. 7.2). The authentication process can then be described as follows. Consider $P_{q_{opt}} = \{p_1, \ldots, p_m\}$. To authenticate $C_q = D_{p_1} \cap \cdots \cap D_{p_m}$, instead of using $L_p$ in $N_f$ of VO for all $p \in P_q^{opt}$ (Case 2 of N_f), we only use $M_{p_{min}}$, where $p_{min} \in P_q^{opt}$ and $|D_{p_{min}}|$ is the smallest among all $|D_p|, p \in P_q^{opt}$. The digest $H_{p_r}$ of each node node($p$) in MHTTree includes $H_{M_p}$ and the VO includes only the graph IDs of $p_{min}$.

Definition 7.2: The digest of a node node($p$) is $h(h(id)h(str(p))h(D_{p_r})h(D_{p_r})h(M_{M_p}))$, where

- $id$ is the ID of $p$, str($p$), $H_{D_{p_r}}$, and $H_{p_r}$ are the same as in Def. 6.3;
- $H_{D_{p_i}}$ is the digest of the size of $D_{p_i}$; and
- $H_{M_p}$ is the root digest of the classical MHT of $M_p$. The data in the MHT are $\{(i, s_i), \ldots, (|P|, s_{|P|})\}$, where for all $i, s_i = M_p(i, \_)$.

The modifications on VO constitution are then described as follows. Other parts of VO are identical to those in Def. 6.6.

- In Case 2 of $N_f$ of VO, for $p_i \in P_q^{opt}$ but $p_i \neq p_{min}$, we include only $n_i = (i, p_i, H_{D_{p_i}}|D_{p_i}|H_{M_{p_{min}}})$ in VO, where $|D_{p_i}|$ is used to verify $p_{min}$ in $P_q^{opt}$ at the client side.
- For $p_{min}, n_{min} = (\min(p_{min}, L_{p_{min}}, B_{p_{min}}))$, where (i) $\min$ is the ID of $p_{min}$; (ii) $p_{min}$ is the POF itself; (iii) $L_{p_{min}}$ contains the IDs of graphs in $C_q$ and the VO of the MHT of ID($D_{p_{min}}$); and (iv) $B_{p_{min}}$ is a set of $(i, s_i)$, where $(i, s_i) \in B_{p_{min}}$ if $p_i \in P_q^{opt}$ and $p_i \neq p_{min}$, and the VO of MHT of $M_{p_{min}}$. 

![Fig. 10. VO for enhanced authentication](image-url)
We remark that $B_{p_{\min}}$ records the bit strings of $s_i$ of $M_{p_{\min}}$, where $i \neq \min$, $s_{\min}$ is not needed as $s_{\min}$ can be derived from $L_{p_{\min}}$. Finally, determining $D_{p_1} \cap \cdots \cap D_{p_n}$ is equivalent to computing $s_1 \wedge \cdots \wedge s_m$ which very often requires smaller VØ. We provide the formal proofs of soundness and completeness of the enhanced method in Appendix A.

**Example 7.1:** Following up Example 6.2. Fig. 10 shows the major parts of the VØ determined by the enhanced method. The differences of VØ from the Example 6.2 are localized in $N_i$. Foremost, $P_1 = \{p_2, p_3, p_7, p_1\}$ and $L_0^{opt} = \{p_3, p_1\}$. Since $|D_{p_3}| = |D_{p_1}| = 2$, we just choose $p_3$ as $p_{\min}$. We show \( \{1\} \) the (partial) matrix $M_{p_3}$ in the LHS of the figure. The bit strings are shown next to the matrix. On the RHS of $M_{p_3}$, it is \( \{0\} \) (partial) MHT. Regarding the VØ, we first discuss $p_{11}$. Since $p_{11}$ is in $L_0^{opt}$ but $p_{11} \neq p_{\min}$. Thus, $n_{11} = (11, p_{11}, H_{D_{p_{11}}, 2}, H_{M_{p_{11}}})$. Next, for $p_{\min}$ (i.e., $p_3$), $n_3$ of $N_i$ is \( \{3, 3, L_{p_3}, B_{p_3}\} \). From previous examples, we have $D_{p_3} = \{g_1, g_4\}$ and $C_q = \{g_1, g_4\}$. $L_{p_3} = [H_{(1, g_4)}, 4]$, where 4 is the graph ID in $C_q$ and $H_{(1, g_4)}$ is \( \{0\} \) the VØ of the MHT of $ID(D_{p_3})$. Regarding $B_{p_3}$, only $p_{11}$ is in $L_0^{opt}$ but $p_{11} \neq p_{\min}$. Thus, $(11, s_{11})$ is included in $B_{p_3}$. Finally, VØ of MHT of $M_{p_3}$ is included in $B_{p_3}$. We remark that the ID of $g_1$ is not needed in $L_{p_3}$, since $s_{11}[1] = 0$ and $s_{11}$ will be authenticated in $B_{p_3}$. Thus, $g_1$ is certainly not in $C_q$.

### 7.2 Clustering Intersect-able Graphs

The matrix $M_p$ (defined in Def. 7.1) not only minimizes the number of graph IDs by using $M_{p_{\min}}$, but also indicates how much VØ is needed for authenticating the candidate set. In particular, let $intv(M_p, i)$ denote the number of intervals in the row of $p_i$, where all entries in each interval are 1s. The 1s in $M_p(i, *)$ correspond to the graphs in $D_p \cap D_{p_i}$ and $intv(M_p, i)$ is the number of ranges needed to be authenticated. To authenticate a range, the upper and lower bounds of the range are needed in VØ. This argument can be generalized to the intersections of multiple sets.

In this subsection, we define the problem of optimal permutation (of columns) of $M_p$. The ordering of graphs in $ID(D_{p})$ is optimal when intersecting the graphs of other P0Fs, the number of the intervals is minimized. We remark that the ordering is optimal in the absence of queries.

**Definition 7.3:** Given a $m \times n$ binary matrix $M_p$ for node $(p)$, the optimal permutation for $M_p$ (OPM) is to transform $M_p$ into $M'_p$ by column permutation s.t. cost($M'_p$) = $\sum_{i=1}^{m}$ intv($M_p, i$) is minimized, where $|P| = m$ and $|D_{p_i}| = n$.

Finding the optimal ordering of graphs of $ID(D_{p})$ is to determine the optimal column permutation of $M_p$. Its hardness is established by a reduction from Shortest Hamiltonian Path (SHP). Details are shown in Appendix A.

**Proposition 7.1:** The problem of OPM is NP-hard. \( \square \)

The OPM problem can be solved by heuristics of SHP. We cast an instance of OPM into that of SHP. Specifically, given an instance of OPM $M_p$, we generate a complete graph in terms of $M_p$. Each column (graph ID in $ID(D_{p}))$ of $M_p$ is a vertex and the weight of the edge between two vertices is the total number of different 1s between the two respective columns. The difference of the row of $p_i$ states that one graph has $p_i$ but the other does not. That is, one graph appears in $D_p \cap D_{p_i}$ and the other does not. A final trick is to add an artificial node $s_0$ as the source and sink of the graph being constructed. We extend $M_p$ with a column of zeros for $s_0$. The SHP of such a complete graph encodes a permutation of columns of $M_p$. We have proved that the total sum of the weight of the optimal SHP is twice of the number of intervals in $M_p$ after the optimal permutation. One of the most efficient approximation algorithms for SHP LKH-2 [18] is adopted. The algorithm is $K$-opt and the approximation ratio is preserved under the above conversion.

**Example 7.2:** To illustrate the effect of the permutation, we create a small artificial example. Suppose that $P_0^{opt} = \{p_1, p_3\}$, $ID(D_{p_1}) = \{1, 3, 5, 7, 9\}$ and $ID(D_{p_3}) = \{2, 3, 8, 9\}$. Assume further $p_1$ and $p_3$ are the only P0Fs of the database. Then, $C_q = \{g_3, g_9\}$. $p_{\min}$ is $p_j$ as $|D_{p_j}| = 4$ and $|D_{p_1}| = 5$. Before permutation, the $L_{p_{\min}}$ in VØ is $[H_{(2, g_2)}, 3, H_{(8, g_8)}, 9]$. In contrast, after the permutation, $ID(D_{p_{\min}}) = \{2, 3, 8, 9\}$. The $L_{p_{\min}}$ contains $[H_{(2, g_2)}, (8, g_8), 3, 9]$.

### 8 Experimental Evaluation

In this section, we present a detailed experimental evaluation that verifies the performance of our proposed techniques and the effectiveness of our optimizations.

#### 8.1 Experimental Setup

**Running Platform.** We conducted all our experiments on a machine with an Intel Core 2 Quad 2.4GHz CPU and 4 GB memory running Windows 7 OS. All our techniques were implemented using C++. We implemented our algorithms on top of iGraph [8]. SHA and RSA were used as our cryptographic signing schemes.

**Dataset.** Following previous experiments of iGraph, we used the same real-world and synthetic datasets in our experimental evaluation. The real-world dataset consists of 10,000 graphs, all of which are drawn from a real AIDS Antiviral dataset (hereafter denoted as AIDS) [24]. AIDS has been used in many studies of subgraph queries [3], [9], [12], [28], [29], [32], [36], [37]. On average, AIDS has 25.42 vertices and 27.40 edges. The number of distinct vertex labels and distinct edge labels are 51 and 4.

For the synthetic dataset, we used SYN.10K.E30.D3.L50 (denoted as SYN). It contains 10,000 graphs of which the average size (the number of edges) is 30; the average density is 0.3; and the number of distinct vertex/edge labels is 50.

We used gSpan [31] with the default settings [32] on the above two datasets to obtain a set of discriminative frequent features, which are served as individual features for our experiment.

**Query sets.** For both AIDS and SYN, the query sets (denoted as $Q_n$) used have been benchmarked in previous works [3], [9], [12], [28], [32], [36]. Each $Q_n$ contains 1000 graphs with size (the number of edges) of $n$, e.g., $Q_4$ represents 1000 graphs sized 4.

**I/O cost and query time comparison.** We used two representative indexes, namely gIndex [32] and FGIndex [3], to compare the I/O cost (number of graph IDs and graph data fetched) and query time of IFTree. We used the same settings for gIndex and FGIndex as in previous experiment [8]. We note that gIndex often outperformed FGIndex except
for small queries and hence we concentrated on comparisons using gIndex.

**Baseline comparison.** Since there is no existing work on subgraph query authentication, we implemented the authentication on gIndex [32] as a baseline, denoted as MgIndex (see SubSec. 3.4). For MgIndex, we also used the same settings as gIndex. Since it is known that binary MHTs yield smaller \( \mathcal{V}O \), in our implementation, the MHTs used are binary MHTs.

**Offline computation and memory overhead.** The offline computation mainly involves (1) the selection of individual features, which takes around 0.5min and 1min for AIDS and SYN, respectively; (2) the selection of POFs, which takes around 30min and 1min for each of the dataset; and (3) the clustering of the intersect-able graphs, which takes around 24h for each of the dataset. For both basic authentication and enhanced authentication, the memory consumptions at the server side and the client side are always smaller than 300MB and 8MB, respectively.

### 8.2 Experiments on AIDS

**Effects of \( \text{maxSize} \) and \( \text{minSup} \) of POF.** Fig. 11(a) reports the effects of the maximal size (\( \text{maxSize} \)) and the minimum support (\( \text{minSup} \)) of POFs by varying \( \text{maxSize} \) and \( \text{minSup} \) for Q8 queries. The x-axis is (\( \text{maxSize}, \text{minSup} \)), e.g., (4, 0.5) represents \( \text{maxSize} = 4 \) and \( \text{minSup} = 500 \). The trends were that when \( \text{minSup} \) increased or \( \text{maxSize} \) decreased, the number of POFs of the IFTree (i.e., the index nodes needed by IFTree) decreased and the candidate size increased (which is directly related to \( \mathcal{V}O \) size). We set the default values of \( \text{maxSize} \) and \( \text{minSup} \) to 4 and 500 to strike a balance between pruning and IFTree size.

1 I/O cost and query performance.

**Average number of graph IDs.** Fig. 11(b) shows the average number of graph IDs fetched at query time by varying the query sizes. Since the numbers for FGIIndex were over 70K, we could not show them here. In Figure 11(b), we can see that IFTree had significantly fewer graph IDs than gIndex, especially when the query size was large. The reason was because the size of \( P_q^{\text{opt}} \) was small as each \( p \in P_q^{\text{opt}} \) was chosen by our heuristic discussed in Sec. 5.2. Moreover, the size of each \( D_p \ (p \in P_q^{\text{opt}}) \) was small.

**Average number of non-answer graphs \( (C_q^{\text{Rq}}) \) in the candidate set.** Fig. 11(c) shows that the average size of \( C_q^{\text{Rq}} \) by varying the query sizes. IFTree produced smaller \( C_q^{\text{Rq}} \) when compared to gIndex and FGIIndex in most cases. For example, at Q4, the \( C_q^{\text{Rq}} \) of IFTree contained 27.2% fewer graphs than that of gIndex. At Q24, IFTree resulted in 13.2% fewer graphs. As FGIIndex was verification-free\(^5\), Q4 queries were small graphs. Most of them were features already and in such cases, there was no non-answer graph in the candidate set. However, when queries were larger than 4, FGIIndex produced larger \( C_q^{\text{Rq}} \).

**Average query time.** Fig. 11(d) reports the average query time at the service provider. At Q4, the average query times on gIndex and IFTree were large since the size of \( C_q^{\text{Rq}} \) was large for small queries. The subIso test on those graphs dominated the query time. FGIIndex was verification-free and Q4 queries in most cases did not require to verify. When the query size increased after Q12, the query time on IFTree became slightly larger. The reason was that the size of \( P_q \) became large and finding the optimal decomposition from \( P_q \) incurred relatively large overhead, while their \( C_q^{\text{Rq}} \)'s of gIndex and IFTree were being similar. However, the benefits of using \( P_q^{\text{opt}} \) became clear in the experiments on authentication.

2 Performance of basic authentication

**Query composition.** Prior to a detailed performance analysis, we show the composition of queries of AIDS, presented in Fig. 12(a). TQ1 are queries that contain exactly one POF in their \( P_q^{\text{opt}} \). In this case, MIFTree does not perform intersections at query time. TQ2 are queries decomposed into multiple POFs and all proposed algorithms in MIFTree affect the performances. From Fig. 12(a), we note that TQ2 dominated the query sets as the query size increased.

**Average number of intersections.** Fig. 12(b) shows us the average number of intersections needed versus the query size. MIFTree required significantly fewer intersections at query time compared to MgIndex. For instance, at Q4 and Q24, MIFTree required 45.2% and 50.8% fewer intersections, respectively, than MgIndex.

**Total \( \mathcal{V}O \) size.** The small number of intersections performed by MIFTree is reflected in the size of \( \mathcal{V}O \). Fig. 12(c) shows the \( \mathcal{V}O \) sizes of MIFTree and MgIndex with varying query sizes. Looking at MgIndex, when the query size increased, the size of the feature set \( |F_q| \) rapidly became larger and the

---

5. Given a query graph \( q \), if \( q \) is a feature, i.e., \( q = f \), it implies that there is no need to verify the subIso between \( q \) and \( g \in C_q \) as \( D_f = C_q = R_q \). Such strategy is called “verification-free” [3].
Fig. 13. Enhanced authentication method
number of intersections performed at query time also increased accordingly. For each addition of feature, \( f \), all the graph IDs of \( D_f \) were added to \( \mathcal{V}O \) (see Fig. 11(b)). Therefore, \( \mathcal{V}O \) enlarged rapidly with query size. For MIFTree, \( \mathcal{V}O \) increased with the query size, although at a slower rate. However, since \( |P_q^{\text{opt}}| \) was often clearly smaller than \( |F_q| \) (see Fig. 12(b)) and for each \( p \) in \( P_q^{\text{opt}} \), \( |D_p| \) was relatively small (see Fig. 11(b)), MIFTree clearly outperformed MgIndex. Moreover, the \( \mathcal{V}O \) of MIFTree did not increase as rapidly as that of MgIndex. We highlight that the \( \mathcal{V}O \) size at Q4 was large since the size of non-answers in the candidate set \( (c_F^q) \) was clearly larger than others, which required some \( \mathcal{V}O \) to authenticate them.

Average authentication time. Fig. 12(d) reports the average authentication time at client side. We observed that the authentication time of MIFTree was often 4 times faster than that of MgIndex. The number of intersections, \( i.e., |P_q^{\text{opt}}| \) was smaller. Thus, fewer MHTs of \( \text{ID}(D_p) \) were reconstructed, which is a performance bottleneck during authentication. Further, the sizes of \( D_p \)s of \( P_q^{\text{opt}} \) were smaller (refer to Fig. 11(b)). These factors made MIFTree clearly more efficient than MgIndex.

3 Performance of enhanced authentication.

While the basic authentication already outperformed MgIndex, in this part, we verify the enhanced method further optimizes authentication performances.

Performance on clustered graphs. We study the \( \mathcal{V}O \) size due to the MHT of \( \text{ID}(D_{\min}) \) in Fig. 13(a) and Fig. 13(b). The queries used were TQ2. Fig. 13(a) first shows the average number of the intervals on \( M_{\min} \) for each queries. Recall SubSec. 7.2, the fewer intervals on \( M_{\min} \), the smaller \( \mathcal{V}O \) size due to the MHT of \( \text{ID}(D_{\min}) \). Therefore, Fig. 13(b) reports such \( \mathcal{V}O \) size, whose trends were similar to Fig. 13(a). We note that the average size of \( \mathcal{V}O \) at Q4 and Q8 increased. The reason was that most of the query features were frequent, then \( C_q \) in \( D_{\min} \), was relatively large. Therefore, the \( \mathcal{V}O \) became larger. At Q12 - Q24, their features contained more infrequent features. Then \( C_q \) was relatively small. Hence, the \( \mathcal{V}O \) for computing \( H_{q, p} \), decreased with the query size. However, in all queries, the graph permutations of the graph IDs of \( D_{\min} \) clearly led to smaller \( \mathcal{V}O \) size.

Total \( \mathcal{V}O \) size. Fig. 13(c) shows the comparison of \( \mathcal{V}O \) sizes between basic method and enhanced method for TQ2. For TQ1, the \( \mathcal{V}O \) of enhanced method was almost the same to that of basic method as there was no intersection for TQ1.

Fig. 14. Authentication performance on synthetic dataset
The figure shows that the enhanced method reduced \( \mathcal{V}O \) sizes significantly. For basic method, \( \mathcal{V}O \) contained all the graph IDs in \( \text{ID}(D_p) \) (\( p \in P_q^{\text{opt}} \)) that were needed to be authenticated (see Fig. 11(b)). Instead, \( \mathcal{V}O \) for enhanced method contained the \( \mathcal{V}O \) of MHT of \( \text{ID}(D_{\min}) \) to authenticate. For instance, at Q24, \( \mathcal{V}O \) by enhanced method was around 120KB whereas that of the basic method was around 20KB.

Average authentication time. Fig. 13(d) shows the comparison of the authentication time of the basic and enhanced methods. At Q4, since the candidate set contained a large number of non-answer graphs (shown in Fig. 11(c)), the subbiso test dominated the authentication time. When the query size went beyond Q4, more queries required the basic method to re-build the root digest of the MHT of each \( \text{ID}(D_p) \), \( p \in P_q^{\text{opt}} \) and the graph IDs were intersected to determine the candidate set. Thus, the authentication time increased rapidly as the query size increased. In comparison, while the authentication time for the enhanced method increased with the query size, it increased in a much slower rate. The reason was that only \( M_{\min} \) and \( \text{ID}(D_{\min}) \) were needed to authenticate.

Overall response time. The overall response time consists of the time for query processing, data transmission and authentication. Although the query times of different methods (Fig. 11(d)) were close, the improvements of our methods of \( \mathcal{V}O \) size (Figs. 12(c) and 13(c)) and authentication times (Figs. 12(d) and 13(d)) were often an order of magnitude more than those of the baseline, which led to better response times.

8.3 Experiments on Synthetic Dataset
Finally, we tested our techniques on SYN. We varied (\text{maxSize}, \text{minSup}) and observed the same trends as those from AIDS. We chose (5, 300) as default. Since the results are similar to those from AIDS, we present some major results in this subsection.

Average query time and authentication time. Fig. 14(a) and Fig. 14(b) show the query time and authentication time, respectively. In Fig. 14(a), we note that the query time of MIFTree was slightly larger than that of the MgIndex. Importantly, Fig. 14(b) shows that the authentication time of MIFTree of basic and enhanced method were at least 3 and 4 times faster than the MgIndex, respectively. The speedup of the enhanced method was up to 8 times. These results were due to smaller \( \mathcal{V}Os \).

Performance on clustered graphs. Fig. 14(c) shows the clustering of graphs of TQ2 queries reduced at least 50% of
the $\mathcal{VO}$ size due to the MHT of $\mathbb{I}(D_{p_m})$. The reasons for the trends were the same to the AIDS. The permutations on $\mathbb{I}(D_{p_m})$ of SYN performed even better than that of AIDS.

**Total $\mathcal{VO}$ size.** We compared $\mathcal{VO}$ size for TQ2 queries between basic method and enhanced method, shown in Fig. 14(d). The figure shows that the enhanced method consistently generated smaller $\mathcal{VO}$s when the query sizes were larger than 4.

## 9 Conclusions

We investigated the authentication of subgraph query services of outsourced graph databases. We proposed an index $\text{IFTree}$ that minimizes the $\mathcal{I/O}$ cost of the popular filtering-and-verification framework for subgraph query processing. We then proposed $\text{MIFTree}$ by extending $\text{IFTree}$ to authenticate subgraph query. To optimize the $\mathcal{VO}$ derived from $\text{MIFTree}$, we proposed a compact $\mathcal{VO}$ representation and a clustering of graphs having similar subset of features. We conducted a detailed experiment to evaluate the performance of our proposed techniques and the effectiveness of the enhancements. For future work, we are investigating the authentication of subgraph similarity query.

**Acknowledgements.** Z. Fan, Y. Peng and B. Choi are partially supported by GRF210510 and FRG2/12-13/079.

**References**


**Zhe Fan** is a PhD student in the Department of Computer Science, Hong Kong Baptist University. He received his BEng degree in Computer Science from South China University of Technology in 2011. His research interests include graph-structured databases. He is a member of the Database Group at Hong Kong Baptist University. (http://www.comp.hkbu.edu.hk/~dbi/).

**Yun Peng** is a PhD student in the Department of Computer Science, Hong Kong Baptist University. He received his BSc and MPhil degrees in Computer Science from Shandong University in 2006 and Harbin Institute of Technology (HIT) in 2008, respectively. His research interests include graph-structured databases. He is a member of the Database Group at Hong Kong Baptist University. (http://www.comp.hkbu.edu.hk/~dbi/).

**Byron Choi** received the bachelor of engineering degree in computer engineering from the Hong Kong University of Science and Technology (HKUST) in 1999 and the MSE and PhD degrees in computer and information science from the University of Pennsylvania in 2002 and 2006, respectively. He is now an assistant professor in the Department of Computer Science at the Hong Kong Baptist University.

**Jianliang Xu** is an associate professor in the Department of Computer Science, Hong Kong Baptist University. He received his BEng degree in computer science and engineering from Zhejiang University, Hangzhou, China, in 1998 and his PhD degree in computer science from Hong Kong University of Science and Technology in 2002. He held visiting positions at Pennsylvania State University and Fudan University.

**Sourav S Bhowmick** is an Associate Professor in the School of Computer Engineering, Nanyang Technological University. He is a Visiting Associate Professor at the Biological Engineering Division, Massachusetts Institute of Technology. He held the position of Singapore-MIT Alliance Fellow in Computation and Systems Biology program (05'-12'). He received his Ph.D. in computer engineering in 2001.
APPENDIX A

PROOFS

In this appendix, we present all the proofs of the propositions in this paper. We then present the proofs of the correctness of our proposed authentication techniques.

A.1 Proof of Prop. 5.1

Proposition 5.1: The problem of the optimal decomposition of a query q from \( P_q \) is NP-hard.

Proof: The proof is established from a simple reduction from minimum weighted set cover problem. For the given query graph \( q \), the universe set \( U \) contains all the individual features in \( F_q \), i.e., \( U = F_q \). The set of subsets of \( U \) is denoted as \( S \), in our cases, \( S = P_q \). For each \( p_i \) in \( S \), its weight is \( w(p_i) = \frac{1}{|p_i|} \). A collection \( S' \) of sets from \( S \), which covers all the individual features in \( U \) and minimizes the \( \sum_{p_i \in S'} w(p_i) \), is the optimal decomposition of \( q \) from \( P_q \), i.e., \( S' = P_q^{opt} \).

That means finding such collection \( S' \), which is the optimal decomposition \( P_q^{opt} \) of \( q \) from \( P_q \), is finding the minimum weighted set cover of \( S \). Therefore, the problem of finding the optimal decomposition of \( q \) from \( P_q \) is NP-hard.

A.2 Proof of Prop. 7.1

Proposition 7.1: The problem of OPM is NP-hard.

Proof: The proof is established from a reduction from Shortest Hamiltonian Path (SHP) problem. Let \( K = (V, E, W) \) be an undirected weighted complete graph with \( n \) vertices. \( V, E, W \) are the set of vertices, edges and weight values, respectively. In particular, \( w(v_i, v_j) \in W \) where \( v_i, v_j \in V \) and \( e_{i,j} \in E \). \( M \) is a \( m \times n \) binary matrix generated from \( K \), where the \( j \)-th column of \( M \) represents \( v_j, v_j \in V \), i.e., the permutation of columns of \( M \) is \( p = (v_1, v_2, \ldots, v_{n-1}, v_n) \) and \( n = |V| \). For all \( i, j \leq n \), \( \text{dist}(M, i, j) = w(v_i, v_j) \), where \( \text{dist}(M, i, j) = |M(*, i) \oplus M(*, j)| \). Specially, \( M(*, 1) = M(*, n) = 0 \). Fix the 1-st and \( n \)-th column of \( M, M' \) is the OPM for \( M \) by columns permutation \( s.t. \) for all possible matrices permuted from \( M \) \( \text{cost}(M') \) is the minimized. Thus the permutation of columns of \( M' \) is denoted as \( p' = (v_1', v_2', \ldots, v_{n-1}', v_n') \).

Then we can get a path \( P \) on \( K \) in terms of \( p' \), where \( P = (v_1, v_2', \ldots, v_{n-1}', v_n) \). Since \( \text{cost}(M') \) is the minimized, and \( \text{cost}(M') = \frac{1}{2} \sum_{i=1}^{n-1} w(v_i', v_{i+1}') = \frac{1}{2} \sum_{i=1}^{n-1} \text{dist}(M', i, i + 1) = \frac{1}{2} \sum_{v_i' \in p', i=1}^{n-1} w(v_i', v_{i+1}') \). Therefore we can then analyze \( \sum_{v_i' \in p', i=1}^{n-1} w(v_i', v_{i+1}') \) is minimized. In this case, \( P \) is exactly the SHP of \( K \). Because finding a SHP from a weighted complete graph is NP-hard, thus the problem of OPM is NP-hard.

A.3 Proof of Soundness and Completeness of the Basic Authentication

Theorem A.3: The basic authentication method is sound and complete.

Proof: We establish our theorem with reference to the authentication steps presented in the end of Sec. 6.2. In order to prove the soundness and completeness of the query answers of the basic authentication, we first comment how we establish the correctness of \( F_q \), \( P_q \) and \( P_q^{opt} \).

Proof of correctness of \( F_q \). \( F \) is organized in a prefix tree. We can establish the correctness of \( F \) from the authentication of prefix trees, studied in [19]. More specifically, in Step 1 of the basic authentication \( F_q \) is computed by the client from \( q \) and \( N_F \). (Recall that \( N_F \) contains the digests of the nodes for the prefix tree of \( F \)). The client has \( q \). The client can verify the features in \( N_F \) are sound and complete with respect to the prefix tree of \( F \) by synthesizing the root digest of the prefix tree and the data owner’s signature \( \psi_F \) [19]. After verifying \( N_F \) is not forged, the client computes if \( F_q \) is the maximum individual fully cover features of \( q \).

Proof of correctness of \( P_q \) and \( P_q^{opt} \). \( P_q \) and \( P_q^{opt} \) are computed from \( q \) and \( F_q \) (in Step 2) and (3) of the authentication).

The MIFTree is a prefix tree of the string representations of P0Fs. The soundness of \( N_I \) can be established by the correctness of the authentication of prefix trees [19]. The client can compute \( P_q \) by applying find_P0F on \( q \), \( F_q \) and \( N_I \). If \( N_I \) is not consistent to the \( P_q \) computed, then the client can be alerted that \( N_I \) is tampered with. The client can compute the correct \( P_q^{opt} \) using opt_P0F_MWSC.

With the correct of \( F_q \), \( P_q \) and \( P_q^{opt} \), we can then analyze the soundness and completeness of the query answers.

Proof of soundness of \( R_q \). Assume that a graph \( g \) in \( R_q \) is modified or bogus. There are only two possible cases:

- \( \text{subIso}(q, g) = \text{false} \): this is detected when the client performs subIso by using the isomorphic mapping \( I_M \) (in Step 7) of authentication).
- \( \text{subIso}(q, g) = \text{true} \): since \( g \) is a bogus, the digest of the node that \( g \) belongs to cannot be synthesized because \( h \) is a one-way collision-resistant function. Subsequently the digest \( H_p \) is not generated correctly and the client can detect this with the signature \( \psi_I \) and the public key (in Steps 4 and 5) of authentication).

Proof of completeness of \( R_q \). Assume a graph \( g \) is an answer but not in \( R_q \). There are two possible cases:

- \( g \in C_q^{R_q} \): This is detected when the client performs subIso on all graphs in \( C_q^{R_q} \) (in Step 7) of authentication).
- \( g \notin C_q^{R_q} \): Denote \( P_q^{opt} \) as \( \{p_1, \ldots, p_m\} \). Since \( P_q^{opt} \) is correct and \( g \) is an answer, \( g \) contains an instance of \( F_q^{opt} \).

There are only two cases that lead to \( g \notin C_q^{R_q} \) and \( g \notin R_q \):

- \( \exists p_i \), \( g \notin D_{p_i} \). In this case, \( S_P \) had modified \( D_{p_i} \). Hence, \( H_p \) and \( H_{p_i} \) cannot be synthesized correctly, and this will be detected when comparing the signature \( \psi_I \); or
- \( \forall p_i \), \( g \in D_{p_i} \). While the \( S_P \) may perform the intersections of all \( D_{p_i} \), \( p_i \in P_q^{opt} \) (in Step 6) of authentication). Hence, the client will be able to detect that \( g \) is an answer.
A.4 Proof of Soundness and Completeness of Enhanced Authentication

Theorem A.4: The enhanced authentication method is sound and complete.

Proof: The proof of correctness of $F_{oi}$, $P_{oi}$ and $P_{opt}$ is the same as the one presented in Theorem A.3. The proof of the soundness of $R_q$ is the same as that of Theorem A.3. Here, we focus on the proof of the completeness.

Proof of completeness of $R_q$. Assume a graph is an answer but not in $R_q$. There are two possible cases:

- $g \in C_q^{R_q}$: This detected when the client performs subiso on all graphs in $C_q^{R_q}$; or
- $g \notin C_q^{R_q}$: Denote $P_{opt}^g$ as $\{p_0, \ldots, p_m\}$. $g$ contains instances of $P_{opt}^g$ as $g$ is an answer. There are again two cases that lead to $g \notin C_q^{R_q} \land g \notin R_q$:
  - $\exists p_i, g \notin D_{pi}$. In this case, $SP$ had modified $D_{pi}$, i.e., $M_{pi}$ had been modified. Hence, $H_{M_{pi}}$ cannot be synthesized correctly which leads to wrong $H_{pi}$, and the client will be alerted when comparing the $DO$ signature; or
  - $\forall p_i, g \in D_{pi}$. The $SP$ has performed the intersection incorrectly. However, this is detected when the client performs the conjunctions on all $s_i, p_i \in P_{opt}^g$.

\[ \square \]

APPENDIX B

DETAILS OF THE BASELINE APPROACH – MgINDEX

In this section, we provide the details of the baseline approach for authenticated subgraph query processing algorithm (Alg. 3). These details supplement the verbose pseudo-code, that is used to construct the running example in Example 3.2.

The overall authentication algorithm (auth_MgIndex). The overall algorithm can be described as follows. The inputs of Alg. 3 are the query graph $q$ and $T_F$, where $T_F$ is the prefix tree of features $F$. The outputs are the query result $R_q$ and their $VO$. It first finds all the maximal features $F_{opt}$ by using find_maxfeatures (Line 2). find_maxfeatures (to be elaborated next) is a traversal on the search tree $T_F$ that constructs $VO_{index}$. After computing $F_{opt}$, it then computes the candidate set $C_q$ (Lines 3-4) by intersections. In Lines 5-10, $VO_{index}$ is modified according to the candidate answer determined in Lines 3-4. The construction of $VO_{C_q}$ is presented in Lines 11-15. Finally, Alg. 3 generates the total $VO$ and returns with $R_q$ (Lines 16-17).

Enumeration of features (find_maxfeatures). Alg. 4 presents the algorithm for find_maxfeatures. The algorithm is presented in the style of gIndex [32], which is a depth first search of minimum DFS order [32] to enumerate all the maximal features $F_{opt}$ of $q$. In Alg. 4, the only difference from [32] is that it needs to record the $VO_{index}$ while searching for the features, which will be used in Alg. 3. In Line 1, $VO_{index}'$ is the $VO_{index}$ at the boundary of the search; $S$ is a set of features enumerated so far; $F_q$ is the maximal features of $q$; and $U$ is the edges of $q$ covered by $S$. The algorithm first sorts all the (individual) edges in $q$ ordered by minimum DFS order (Line 2). It records the feature and the associated digest for each child node of the root of $R_q$ (Lines 3-5) in $VO_{index}'$ as they are at the boundary of the current search. Alg. 4 then invokes the traversal algorithm auth_DFS (Lines 6-8) to enumerate the features of $q$. After generating all the features of $q$, the algorithm then computes and returns the maximal features $F_q$ (Lines 9-10).

The traversal pseudo-code of a prefix search tree (auth_DFS). The Procedure 4.1 is a depth first search procedure with generating the features of $q$ and the $VO_{index}$. At each traversal step, if the current feature $s$ is not the minimum [32], then the current recursion is terminated (Lines 11-12). If $s$ is in $T_F$ (Line 13), then $s$ is a feature of $q$ (Line 13). The edge $e$ is covered by $s$ and $s$ is added to $S$ (Lines 14-15). The feature and the associated digest for each child node of $s$ is involved in $VO_{index}$ (Lines 16-18). Then, auth_DFS proceeds to each child $e$ of $s$ in $q$ (Lines 20-22), again in the minimum DFS order.

Example B.1: We use Example 3.1 to illustrate the search of the query features in Alg. 4. The search find_maxfeatures starts at the artificial root node $f_1$. The child nodes of $f_1$ are the current boundary nodes and they are recorded in $VO_{index}'$ (Lines 3-5). The search proceeds to the child nodes of $f_1$ with the minimal DFS order $S^1$ (Lines 6-8).

According to the example, the first edge $e$ in $S^1$ is $(C, O)$. In the sub-procedure auth_DFS, since $s$ is a minimum (Lines 11-12) and $s$ is exactly $f_2$ (Line 13), where $s = e$, $e$ is added in $U$ as covered (Line 14) and $s$ is added in $S$ as a feature of $q$ (Line 15). $s$ (i.e., $f_2$) is the current visited node, and the child nodes of $f_2$ are the boundary nodes which are recorded in $VO_{index}$ (Lines 16-19). The auth_DFS then recursively expands the search to the child of $s$ in $q$ (Lines 20-22), i.e., expands $(O, H)$ and $(O, O)$ respectively. However, $s$ expanded with $(O, H)$ is not a feature of $q$ (Line 13). The traversal then proceeds to $s$ expanded with $(O, O)$ (i.e., $f_7$) and modifies $VO_{index}'$.

After auth_DFS finishes traversing the search tree rooted at $f_2$, the only edge of $q$ not covered by $S$ is $(O, H)$. find_maxfeatures then proceeds to search $(O, H)$ as it is the next edge in $S^1$ (Line 6). Similarly, auth_DFS traverses the subtree rooted at $f_3$ (Lines 7-8). After the traversal, $S$ covered $q$ (i.e., $U = q$) and the traversal terminates (Line 10).

Finally, find_maxfeatures determines $F_q = \{f_7, f_3\}$ from $S$ and returns $F_q$ (Lines 9-10).

APPENDIX C

AUTHENTICATED SUBGRAPH QUERIES ON LIGHTWEIGHT DEVICES

This experiment verifies that MIFTreed is a practical approach that enables clients to access authenticated subgraph query services via lightweight devices. We chose an extreme hardware setting where the client uses a commodity smartphone. We report both the time for query processing and energy consumption of the subgraph queries on the smartphone.

Hardware setting. The smartphone used in this experiment has an 1GHz processor, 1GB internal memory and 3.7 Volt, 1500 mAh battery running the Android 2.2 system.
**Algorithm 3 auth_MgIndex(q, T_F)**

**Input:** q is a query graph, the prefix tree T_F of features F

**Output:** R_q, VO

1: Initialize R_q = { }, C_q = G, VO_index = { }, VO_C_q = { }
2: F_q = find_maxfeatures(q, T_F, VO_index)
3: /* compute C_q */
4: for each f ∈ F_q
5: /* construct VO_index */
6: Initialize a list L = [ ]
7: for each g_j ∈ D_f
8: if g_j ∈ C_q then L = L ⊔ g_j /* append ID */
9: else L = L ⊔ (j, H_g_j) /* append ID and digest */
10: VO_index[f] = (f, L)
11: /* construct R_q and VO_C_q */
12: for each g ∈ C_q
13: if subGIs(q, g) = true
14: R_q = R_q ⊔ g
15: else /* construct VO for non-answer */
16: VO_C_q = VO_C_q ⊔ g
17: return R_q and VO

**Algorithm 4 find_maxfeatures(q, T_F, VO_index)**

**Input:** q is a query graph, the prefix tree T_F of features F, VO_index is the V O records the search

**Output:** F_q

1: Initialize VO_index = { }, S = { }, F_q = { }, U = { }
2: S^1 is a list of edges e of q ordered by the minimum DFS order
3: for each f of f_e in T_F /* f_e is the root node of T_F */
4: VO_index[f] = (f, H_f) /* boundary nodes */
5: VO_index[f_e] = (f_e, VO_index') /* visited nodes */
6: for each edge e in S^1, e ∈ U \ U \ q \ U
7: s = e
8: auth_DFS(e, s, S, VO_index', q, T_F, U)
9: compute F_q ⊆ S, s.t., F_q is a set of maximal features
10: return F_q

**Procedure 4.1 auth_DFS(e, s, S, VO_index', q, T_F, U)**

11: if s \неmind ﬂash(s)
12: return
13: if VO_index[s] \neq \emptyset /* s is a feature in T_F */
14: U = U \ e /* mark e as covered */
15: if s ∈ S
16: initialize VO_index' = [ ]
17: for each child f of s in T_F
18: VO_index'[f] = (f, H_f) /* boundary nodes */
19: VO_index'[s] = (s, VO_index') /* visited nodes */
20: for each child e of s in q \ q \ U
21: e' = e \ s
22: auth_DFS(e', e, S, VO_index', q, T_F, U)

Software setting. We implemented the semimal subgraph isomorphism algorithm, namely the Ullman’s algorithm, by Java on Android. The dataset we used is the benchmark dataset AIDS [24], which was used in Sec. 8. The queries we tested are Q4, i.e. the query graphs of the size 4.

Results and discussions. In our experiment, we consider three cases as follows. (1) Prior to our work, there is no indexing technique that supports authenticated subgraph queries. In the absence of indexing techniques, the SP or DO is required to send the whole database and the DO’s signature to the client, for each query. The client can verify the integrity of the database with DO’s signature. Next, the client scans the graphs to compute the answers. For each Q4 on the dataset AIDS (containing 10K), the smartphone took around 32.6 minutes to determine the answers and 11.7% of the battery was consumed. It is not surprising that the bottleneck is the subgraph isomorphism computation. Moreover, large queries of the AIDS dataset exhibited similar or worse performances. Hence, it is imperative to propose an efficient authentication mechanism on top of indexing techniques. (2) Suppose the client can access to an authenticated subgraph query service using the baseline method. We simulated the evaluation of Q4 again. The authentication on the smartphone required to invoke subgraph isomorphism on 960 graphs. The main reason is that the baseline method also minimizes the number of candidate graphs in the V O. It took 3.3 minutes and drained around 1.1% of the battery for one Q4 query. (3) We tested our MIFTree approach. Subgraph isomorphism was then invoked on 700 graphs only. This further reduced the battery consumption of the Q4 query to 0.81%. The answers were authenticated in around 2.4 minutes. In this case, the battery saved by using the MIFTree approach is about 27% of the battery consumption of the baseline approach.

**APPENDIX D**

**FREQUENTLY USED SYMBOLS**

We present the list of frequently used symbols of our discussions in Tab. 1.

**APPENDIX E**

**V O SIZE VS. AUTHENTICATION TIME**

In this appendix, we report a supplementary experiment of the basic authentication method on AIDS dataset in order to show the relationship between V O size and authentication time. The reason for not using enhanced authentication method is that its V O size and authentication time are affected by several non-trivial optimizations, e.g., the matrix representation of graph IDs and the clustering of intersect-able graphs. Therefore, we opt to use the basic method for this supplementary experiment.

The experimental results of Q4-Q24 are reported in Figs. 15(a)-(f), respectively. Each figure is obtained from an experiment of a specific query set on the AIDS dataset. All the query sets used (i.e., Q4-Q24) are the same to those in
Sec. 8. Each dot in the figure represents a query; the x-axis of the figure represents the \( \mathcal{VO} \) size due to the query; and the y-axis stands for its authentication time. From the figures, we can easily observe that there are (roughly) linear correlations between \( \mathcal{VO} \) size and authentication time. Therefore, a major portion of this paper discusses \( \mathcal{VO} \) minimization techniques for \( \text{IFTree} \) to address efficient authenticated subgraph query processing.

Fig. 15. \( \mathcal{VO} \) size vs. authentication time of various query sizes of the basic authentication method on the AIDS dataset.