Towards efficient authenticated subgraph query service in outsourced graph databases

Fan, Zhe; PENG, Yun; CHOI, Koon Kau; XU, Jianliang; Bhowmick, Sourav S.

Published in:
IEEE Transactions on Services Computing

DOI:
10.1109/TSC.2013.42

Published: 01/10/2014

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 ($\mathcal{VO}$s) 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, chemi-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.

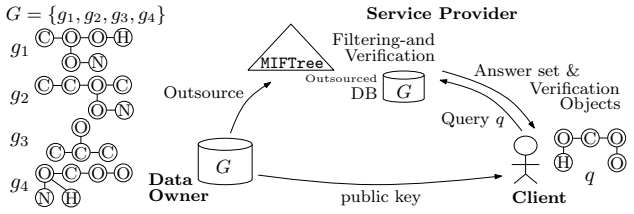


Fig. 1. System model of graph database outsourcing

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

- *Zhe Fan, Yun Peng, Byron Choi and Jianliang Xu are with the Department of Computer Science, Hong Kong Baptist University, China. E-mail: zfan, ypeng, bchoi, xujl@comp.hkbu.edu.hk*
- *Sourav S. Bhowmick is with School of Computer Engineering, Nanyang Technological University, Singapore. E-mail: assourav@ntu.edu.sg*

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 [3], [12], [28], [29], [32], [36] 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 [7], 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 ($\mathcal{VO}$) which stores the processing traces such as index traversals and; By using the answer and $\mathcal{VO}$, 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 $\mathcal{VO}$ 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 $\mathcal{VO}$. 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 $\mathcal{VO}$ 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 $\mathcal{VO}$s 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 `MHT`s 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 `POF`s (*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 $\mathcal{VO}$ 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 $\mathcal{VO}$ 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 $\mathcal{VO}$ 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 `MHT`s 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 approaches* (*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 $V(g)$, $E(g)$, Σ and l are the set 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 $\texttt{subIso}(g, g') = \texttt{true}$. 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 $R_q = \{g_i | \texttt{subIso}(q, g_i), g_i \in G\}$.

Subgraph query paradigms. Two query paradigms for subgraph queries have been proposed: feature-based (*e.g.*, [3], [12], [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.

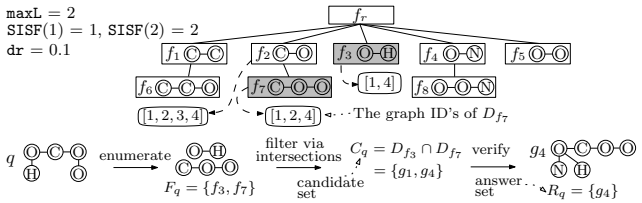


Fig. 2. Illustration of gIndex and its query processing

Feature-based approaches index graphs by their individual features. The term *individual feature* is used to refer to those proposed previously, as the one we put forward comprise individual features that form "higher-order" features. An index of this approach uses these features as the search keys for the graphs that contain them.

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 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 maxL, where maxL is a user-defined *maximum feature size*;
- a *frequent feature* that $|D_f| \geq \texttt{SISF}(|f|)$, where $D_f = \{f |\ f \subseteq g, g \in G\}$, $|D_f|$ is called the *support* of f and SISF is a user-defined *Size-Increasing-Support-Function* of each f; and
- discriminative, *s.t.*, $\frac{|\cap_{f' \in F \wedge f' \subset f} D_{f'}|}{|D_f|} \geq \texttt{dr}$, where dr is a user-defined *discriminative ratio*.

The function SISF returns a support that increases with the input feature size. gIndex sets $\texttt{SISF}(1) = 1$ by default. 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 $F_q = \{f | f \subseteq q, f \in F, \nexists f', s.t., f \subseteq f', f' \subseteq q\}$, 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:

$$C_q = \bigcap_{f \in F_q, F_q \subseteq F} D_f \tag{1}$$

(2) *Verification*: determine the query answers R_q from the candidate set by invoking 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 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, $F = \{f_1, f_2, \cdots, f_8\}$, where $\texttt{SISF}(1) = 1$, $\texttt{SISF}(2) = 2$, maxL and dr are set to 2 and 0.1, respectively. f_r 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 $F_q = \{f_3, f_7\}$ of q and performs *intersections* of the graphs (via IDs) containing the individual feature(s) (D_{f_3} and D_{f_7}) to compute the candidate set $C_q = D_{f_3} \bigcap D_{f_7} = \{g_1, g_4\}$. The verification phase invokes subIso on each graph in C_q, and computes the answers $R_q = \{g_4\}$.

3.2 Background for Query Authentication

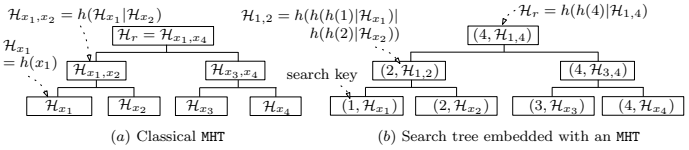
Cryptographic primitives. Similar to other works on authentication, we assume a *one-way collision-resistant hash function* (*e.g.*, SHA and MD5) is denoted as $h(x)$, where x is a data value to be hashed and the hash value $h(x)$ is often referred to as the *digest* of x. It is infeasible to determine 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 $y = \texttt{sign}(x, \texttt{SK})$. 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 $\{x_1, ..., x_4\}$. Each leaf node is associated with the digest (hash) of its data value, *e.g.*, $\mathcal{H}_{x_1} = h(x_1)$. Each internal node contains the digest of the concatenation of the digest of its child nodes, *e.g.*, $\mathcal{H}_{x_1,x_2} = h(\mathcal{H}_{x_1}|\mathcal{H}_{x_2})$. 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 $\mathcal{VO}$ that consists of the digests $\mathcal{H}_{x_1}$ and $\mathcal{H}_{x_3,x_4}$ and the signed root digest of $\mathcal{H}_r$. The client computes from the $\mathcal{VO}$, $\mathcal{H}_{x_2} = h(x_2)$, $\mathcal{H}_{x_1,x_2} = h(\mathcal{H}_{x_1}|\mathcal{H}_{x_2})$, and finally the root digest $\mathcal{H}_{x_1,x_4} = h(\mathcal{H}_{x_1,x_2}|\mathcal{H}_{x_3,x_4})$. The client uses the data owner's public key to compare $\mathcal{H}_{x_1,x_4}$ 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 $\mathcal{VO}$ sizes. Fig. $3(b)$ shows a search tree embedded with an MHT. The data in the MHT are $\{x_1, ..., x_4\}$, 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.*, $(1, \mathcal{H}_{x_1})$ 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.*, $(2, \mathcal{H}_{1,2})$ where $\mathcal{H}_{1,2} = h(h(h(1)|\mathcal{H}_{x_1})|h(h(2)|\mathcal{H}_{x_2}))$.



(a) Classical MHT (b) Search tree embedded with an MHT

Fig. 3. Two kinds of Merkle Hash Trees used

Suppose that *the search of the key* 2 needs to authenticate. The $\mathcal{VO}$ contains (1, $\mathcal{H}_{x_1}$) and (4, $\mathcal{H}_{3,4}$) and the data owner's signature on the root digest $\mathcal{H}_r$. The client computes $\mathcal{H}_{x_2} = h(x_2)$, $\mathcal{H}_{1,2} = h(h(h(1)|\mathcal{H}_{x_1})|\ h(h(2)|\mathcal{H}_{x_2}))$, $\mathcal{H}_{1,4} = h(h(h(2)|\mathcal{H}_{1,2})|\ h(h(4)|\ \mathcal{H}_{3,4}))$, and finally the root digest $\mathcal{H}_r = h(h(4)|\mathcal{H}_{1,4})$. 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 $\mathcal{VO}$.

3.3 Problem Formulation

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

(i) The $\mathcal{DO}$ owns a graph database G. The $\mathcal{DO}$ or $\mathcal{SP}$ first generates an index to support subgraph query processing. Then, $\mathcal{DO}$ signs the root digest of the index. (ii) The $\mathcal{SP}$ receives a query q from a client, processes it on behalf of $\mathcal{DO}$ and returns the answer graphs R_q to the client. Since $\mathcal{SP}$ may not be trusted, it is required to return not only R_q but also a $\mathcal{VO}$ and the $\mathcal{DO}$'s signature to the client. (iii) Upon receiving the $\mathcal{VO}$, the client verifies the R_q the $\mathcal{SP}$ returns. We assume the client has the public key of the $\mathcal{DO}$ 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.*, $\forall g \in R_q,\ g \in G \wedge q \subseteq g$; and
- Completeness: there is no graph that is not in R_q but is an answer, *i.e.*, $\nexists g \notin R_q, g \in G \wedge q \subseteq g$.

Threat Model. In our system model, the $\mathcal{SP}$ 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 $\mathcal{SP}$ 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 — MgIndex

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*,

unless otherwise specified. For detailed algorithm, please refer to Appendix B.

With reference to Formula 1 in Sec. 3.1, in order to authenticate the answer of the query q, the client must authenticate the correctness of (i) the query features F_q and (ii) their graph IDs D_f (for all $f \in F_q$) in order to verify the authenticity of the candidate set C_q. Therefore, the client can examine C_q to obtain the answer R_q.

The baseline approach called MgIndex simply applies MHT to (i) the children of each index node of gIndex; and (ii) the graphs (with IDs) of D_f of each feature f, respectively. The query processing of MgIndex is similar to that of gIndex but incorporates with $\mathcal{VO}$ construction. More specifically, the $\mathcal{VO}$ of MgIndex consists of three main parts:

$$\mathcal{VO} = \mathcal{VO}_{\texttt{index}} \cup \mathcal{VO}_{C_q} \cup \psi_F$$

1) $\mathcal{VO}_{\texttt{index}}$ contains the digests that record the search of each individual feature $f \in F_q$ during query processing and all the graph IDs (and the graphs' hash values if the graphs are not present in R_q) of D_f for all $f \in F_q$;
2) $\mathcal{VO}_{C_q}$ contains the non-answer graphs in the candidate set, *i.e.*, $\mathcal{VO}_{C_q} = C_q - R_q$, denoted as $C_q^{\bar{R}_q}$; and
3) ψ_F is simply the signature of the data owner.

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

1) $\mathcal{VO}_{\texttt{index}}$ contains the digests that record the search of $F_q = \{f_3, f_7\}$. Suppose the search locates f_7 first. The $\mathcal{VO}_{\texttt{index}}$ 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 $\mathcal{VO}_{\texttt{index}}$ 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_7} and D_{f_3} (*i.e.*, {1,2,4} and {1,4} respectively), and the hash value of g_2 are added to $\mathcal{VO}_{\texttt{index}}$; 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 MgIndex using $\mathcal{VO}_{\texttt{index}}$ 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 $\mathcal{VO}_{\texttt{index}}$. 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_4 is the answer but not g_1.

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 $\mathcal{VO}_{\texttt{index}}$ 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.

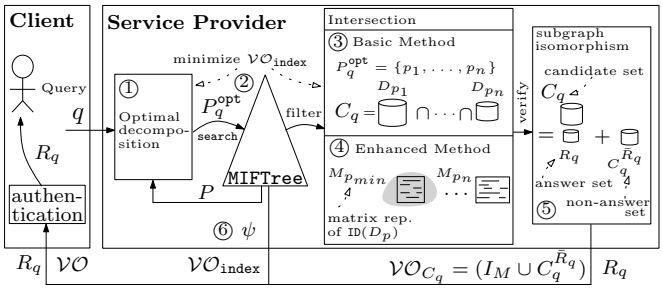


Fig. 4. Overview of efficient authentication method

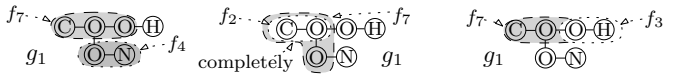
To minimize $\mathcal{VO}_{\texttt{index}}$ while keeping C_q small, we propose essentially to precompute some intersections offline, such that fewer intersections are involved at query time and hence need to be authenticated by clients. In particular, we propose higher-order features (*Partially Overlapping Features* POFs). In a nutshell, a POF consists of a set of overlapping individual features. If a data graph contains a POF, this implies it also contains those individual features in the POF. Hence, POFs are more selective than individual feature, and result in smaller candidate sets.

We propose the Intersection-aware Feature-subgraph Tree (IFTree) to index a graph database by POFs P. Merkle IFTree (MIFTree) is proposed by adopting MHTs on IFTree for basic authentication.

The overview of our solution is depicted in Fig. 4. ① The client issues the query graph q to the $\mathcal{SP}$. The $\mathcal{SP}$ first enumerates all the POFs P_q of q. We then study how to decompose q into an optimal set $P_q^{\texttt{opt}}$ which has *the fewest number of intersections and smallest* C_q. ② $P_q^{\texttt{opt}}$ is then searched on MIFTree to obtain all the graph IDs of D_p, denoted as ID(D_p), where $p \in P_q^{\texttt{opt}}$ and D_p is a set of graphs that contain p. The candidate set C_q is determined by *intersecting* ID(D_p) as shown in Formula 1. ③ We derive a basic method to derive $\mathcal{VO}_{\texttt{index}}$ from MIFTree which is similar to MgIndex. ④ In addition, as the query graph size increases, so does the number of intersections. It is inefficient to include all ID(D_p)s, $p \in P_q^{\texttt{opt}}$ in a $\mathcal{VO}$. Hence, to minimize the $\mathcal{VO}$ needed to authenticate intersections, we propose an enhanced method that uses a compact representation M_p for each D_p of p. We only include the single smallest M_p, namely $M_{p_{min}}$ to $\mathcal{VO}$. $M_{p_{min}}$ itself must be authenticated by the client but the answer graphs indicated by $M_{p_{min}}$ may not fall into a range. We therefore analyze M_p offline to cluster the "intersect-able" graphs in each D_p, $p \in P$, for an optimal ordering of the graphs stored in D_p. ⑤ For $\mathcal{VO}_{C_q}$, we include the non-answer candidate $C_q^{\bar{R}_q}$ and the mappings I_M between the query and its answers. ⑥ $\mathcal{DO}$'s signature ψ is added to $\mathcal{VO}$. The client finally receives the $\mathcal{VO}$ to authenticate the answer.

4 Partially Overlapping Features

In this section, we derive the *partially overlapping features* (POFs) that aim to minimize the number of intersections involved in query time. The benefits are threefold. Fewer intersections are computed in query time; fewer graph IDs are fetched; and more individual features are implicitly involved and often lead to small candidate sets.



(a)co-existing feature (b) overlapping feature (c) partially overlapping feature

Fig. 5. Illustration of higher-order features

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:$\{s_1,\cdots,s_n\}$ 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. $5(a)$, $\{f_4, f_7\}$ is a co-existing feature of g_1. In Fig. $5(b)$, $\{f_2, f_7\}$ 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_{f_7} 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: p: $\{f_1,..., f_n\}$ 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.

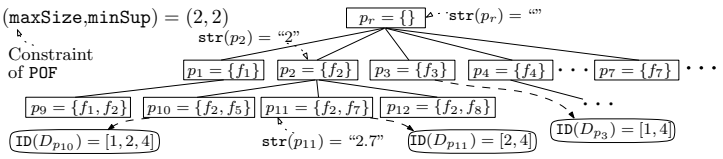


Fig. 6. Illustration of IFTree (partial)

To specify the desired POFs for indexing, we define a user-specified constraint. In particular, POFs should be small in size and have certain minimum support from a database.

Definition 4.4: The *constraint of* POFs P is (maxSize, minSup), where maxSize and minSup are the maximum size and the minimum support of P in a database G, *i.e.,* $\forall p \in P, |p| \leq$ maxSize and $|D_p| \geq$ 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 (maxSize, 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 $f_1, ..., f_n$ are referred to *individual 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 maxSize 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 $\subset$ 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. We use str(p) to denote the string of p. For example, let $p = \{f_1, f_2\}$. str(p) = "1.2". We say p_i precedes p_j, denoted as $p_i \prec p_j$, *iff* str(p_i) is a prefix of str(p_j). With such a representation of POFs, we define a prefix search tree called IFTree.

Definition 5.1: *Intersection-aware Feature-subgraph Tree* (IFTree) is a prefix search tree of POFs P on a graph database G, denoted as T_P : (str, node, V, E, ID, p_r), where

- str is a function that str(p) returns the string of p;
- node takes a POF p and returns the node of p in T_P;
- $V = \{\text{node}(p_i) \mid p_i \in P\}$;
- $E = \{(\text{node}(p_i), \text{node}(p_j)) \mid p_i \prec p_j \wedge (\nexists\ p'_i\ p_i \prec p'_i \wedge p'_i \prec p_j)\}$. The children of a node(p_i) are sorted in lexicographical order w.r.t str;
- ID is a function that ID(D_p) returns the *list* of *ID*s of the graphs in D_p; and
- p_r is an empty POF $\emptyset$ and node(p_r) is an artificial root node of the IFTree.

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 T_P of G
Output: the answer set of q R_q

1: Initialize R_q to $\emptyset$ and C_q to G
2: F_q = find_maxfeatures(q, T_F) //F_q fully cover q
3: P_q = find_POF(q, F_q, T_P) //Enumeration
4: $P_q^{\texttt{opt}}$ = opt_POF_MWSC(F_q, P_q)
5: **for each** $p \in P_q^{\texttt{opt}}$
6: D_p = search(p, T_P); $C_q = C_q \cap D_p$
7: **for each** $g \in C_q$
8: **if** subIso(q, g) = 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 str(p_2) = "2", str(p_9) = "1.2" and therefore $p_2 \not\prec p_9$. To illustrate the processing of existing indexes and IFTree, let's assume that a query contains two individual features f_2 and f_7. gIndex 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 [2]. It takes a query graph q, a graph database G, the prefix tree T_F of features F and the IFTree T_P 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 $P_q^{\texttt{opt}}$ from P_q (Line 4), which shall be discussed shortly. For each POF p in $P_q^{\texttt{opt}}$, 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 subIso(f, f') = true and subIso(f', q) = 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, gIndex 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 $P_q^{\texttt{opt}}$ 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 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.

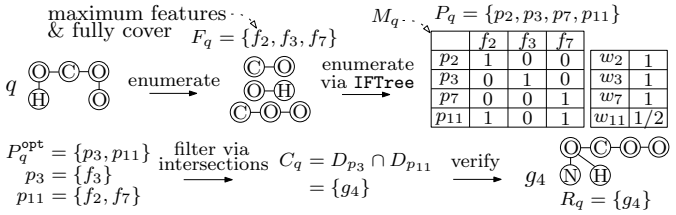


Fig. 7. Subgraph query processing on IFTree

competing objectives in computing $P_q^{\texttt{opt}}$. (i) On one hand, fewer intersections (*i.e.*, fewer POFs in $P_q^{\texttt{opt}}$) on ID(D_p) ($p \in P_q^{\texttt{opt}}$) are desirable to minimize I/O due to graph IDs of D_p's. (ii) On the other hand, larger POFs (*i.e.*, more individual features) in $P_q^{\texttt{opt}}$ cover the query more and reduce the size of the candidate set C_q and the I/O for fetching it. The objectives can be illustrated with an example. Suppose Alg. 1 (find_POF in Line 3) determines $P_q = \{p_2, p_3, p_7, p_{11}\}$. Two possible decompositions are $P_q^1 = \{p_2, p_3, p_7\}$ and $P_q^2 = \{p_3, p_{11}\}$. We may choose P_q^2 since the number of intersections and the candidate set are 2 and $\{g_4\}$, respectively. In contrast, those of P_q^1 are 3 and $\{g_1, g_4\}$, respectively. P_q^2 is in fact the optimal decomposition of q.

Minimization of I/O by using $P_q^{\texttt{opt}}$. The problem discussed above can be formulated as an optimization where both $|P_q^{\texttt{opt}}|$ and I/O are minimized. To present the problem, we define a binary matrix M_q where each row i represents the POF p_i from all possible POFs P_q of q, each column j represents an individual feature $f_j \in F_q$ and each entry $M_q(i, j)$ is 1 if f_j is in p_i, otherwise 0. The weight of each row i of M_q is $w_i = \frac{1}{\texttt{hamming}(M_q(i,*))}$, where hamming$(M_q(i, *))$ is the hamming weight that returns the number of 1s in the row i of M_q. For instance, consider Fig. 7. $w_{11} = \frac{1}{\texttt{hamming}(M_q(11,*))}$ = 1/2 as $p_{11} = \{f_2, f_7\}$.

Definition 5.2: Given a weight value w_i to each row i of M_q, $w_i = \frac{1}{\texttt{hamming}(M_q(i,*))}$, the problem of *optimal decomposition of a query q from P_q* is to determine $P_q^{\texttt{opt}}$, where $P_q^{\texttt{opt}} \subseteq P_q$ and $P_q^{\texttt{opt}}$ fully covers F_q *s.t.* $\sum_{p_i \in P_q^{\texttt{opt}}} w_i$ is minimized.

The optimal decomposition addresses the above two objectives. (i) To minimize $\sum_{p_i \in P_q^{\texttt{opt}}} w_i$, fewer terms are included in the sum, which not only indicates fewer intersections in query processing, but also minimizes I/O due to graph IDs. (ii) For each p_i, the more 1s in $M_q(i, *)$, the more individual features it contains, the smaller w_i and D_{p_i}. Therefore, using p_i leads to a smaller candidate set.

Proposition 5.1: The problem of optimal decomposition of a query q from P_q is NP-hard. □

The hardness can be established from a simple reduction from minimum weighted set cover (MWSC). Due to the space constraint, the proof is presented in Appendix A. We adopt a classical heuristic algorithm for MWSC to solve the problem. The idea is simple: it iteratively chooses the POF with the smallest weight (covering the most number of uncovered features in F_q) and removes the covered features from F_q. It terminates when F_q is empty (fully covered). This heuristic can be exemplified by the example M_q shown in Fig. 7. Initially, $w_2 = w_3 = w_7 = 1$ as the hamming weights of $M_q(2, *)$ $M_q(3, *)$ and $M_q(7, *)$ are 1. $w_{11} = 1/2$ as hamming$(M_q(11, *)) = 2$. In the first iteration, p_{11} is chosen.

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. $P_q^{\texttt{opt}}$ is $\{p_3, p_{11}\}$.

6 MERKLE IFTREE (MIFTREE)

Thanks to the minimization of I/O by using $P_q^{\texttt{opt}}$, the query processing trace needed to be included in $\mathcal{VO}$s is reduced when IFTree is adopted for query authentication. To facilitate efficient authentication, we propose to apply MHTs 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 $\mathcal{VO}$. 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 $\mathcal{DO}$ 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 $\mathcal{H}_{g_i} = h(\texttt{mindfs}(g_i))$.

Graphs are cast into some (publicly known) canonical representation before their digests are computed. In this paper, we adopt the minimum DFS code [31],[3] denoted as mindfs, but other representations may also be adopted.

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

- str(p) is the string of p;
- $\mathcal{H}_{D_p}$ is the root digest of the classical MHT of ID(D_p) : $[j_1, \cdots, j_m]$. The data in the MHT are $\{(j_1,g_{j_1}),\cdots,(j_m,g_{j_m})\}$; and
- $\mathcal{H}_p^r$ is the root digest of the embedded MHT of node(p)'s children. The data in the MHT are $\{\texttt{node}(p_1), \cdots, \texttt{node}(p_m)\}$ and the search keys are $\{p_1, \cdots, p_m\}$, where node(p_1), $\cdots$, 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,O)). The first two digits are the DFS sequence of the vertices of a graph. The following characters are vertices' labels.

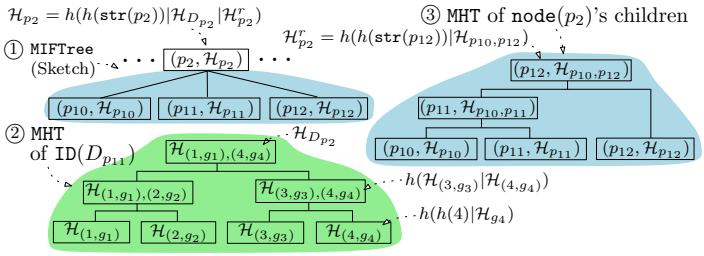


Fig. 8. Illustration of MHTs of node(p_2) of an MIFTree (partial)

Example 6.1: With reference to Fig. 8, we present an example of the digest of node(p_2), denoted as $\mathcal{H}_{p_2} = h(h(\texttt{str}(p_2))|\mathcal{H}_{D_{p_2}}|\ \mathcal{H}_{p_2}^r)$. ① is the sketch of MIFTree. $\mathcal{H}_{D_{p_2}}$ is the root digest of ② the classical MHT of ID(D_{p_2}), which is built on top of the data $\{(1,g_1),\cdots,(4,g_4)\}$. $\mathcal{H}_{p_2}^r$ is the root digest of ③ the embedded MHT of node(p_2)'s children, which are node(p_{10}), node(p_{11}) and node(p_{12}). The data it embeds are $\{$node$(p_{10}),\cdots,$node$(p_{12})\}$, while the search keys are $\{p_{10}, p_{11}, p_{12}\}$.

Definition 6.4: The *signature of the root* node(p_r) of MIFTree is ψ_I = sign$(h(h(\texttt{str}(p_r))|\mathcal{H}_{p_r})$, SK$)$, where SK is the private key of the $\mathcal{DO}$.

It should be remarked that the individual features F_q must be authenticated in order to verify the correctness of POFs. We organize all features F of G with a prefix tree T_F similar to MIFTree. The authentication process of F_q is simpler than that of $P_q^{\texttt{opt}}$.

6.2 Basic Authentication Method

In this subsection, we present the constitution of $\mathcal{VO}$ and a basic authentication method. For a concise exposition, we present the details in *set semantics*, unless otherwise specified.

Verification object. The overview of the constitution of $\mathcal{VO}$ can be given as follows. $\mathcal{VO}$ consists of the $\mathcal{VO}$ for recording the searches of $P_q^{\texttt{opt}}$ on MIFTree ($\mathcal{VO}_{\texttt{index}}$) and the $\mathcal{VO}$ for the candidate set ($\mathcal{VO}_{C_q}$). Informally, $\mathcal{VO}_{\texttt{index}}$ includes the *visited nodes in searching* POFs (denoted as N_I^v) and *some boundaries of the search paths of* POFs (denoted as N_I^b). These are necessary to reconstruct the digest of the root of MIFTree. Moreover, the graphs (not only their IDs) in the candidate set are included in $\mathcal{VO}_{C_q}$ for client's verification. While the query answers R_q must be returned, the non-answers in the candidate set C_q must also be included in $\mathcal{VO}_{C_q}$, denoted as $C_q^{\bar{R}_q}$, where $C_q^{\bar{R}_q} = C_q - R_q$, to verify that no graph in $C_q^{\bar{R}_q}$ is an answer. For verification efficiency, the mappings between the query and the answers are included in $\mathcal{VO}_{C_q}$. To sum up, we define the constitution of $\mathcal{VO}$, presented in Def. 6.6 which consists of the structures and auxiliary structures discussed above.

As discussed in Def. 6.1, we have applied MHTs in MIFTree for small $\mathcal{VO}$. The description of $\mathcal{VO}$ of an MHT is well-known but verbose, which includes the answers, the boundaries and the search keys of search paths. For succinct presentation, we define a term "$\mathcal{VO}$ of MHT" to leverage on the known results from MHT.

Definition 6.5: Suppose an MHT is built on a set of objects O :$\{o_1, \cdots, o_n\}$ and the corresponding search keys are $\{k_1, \cdots, k_n\}$ (if has). Given a set of objects O', $O' \subseteq O$, the *$\mathcal{VO}$ of the* MHT *of* O is the $\mathcal{VO}$ needed to authenticate O'.

Algorithm 2 `Auth_Query_Processing` (q, G, T_F, T_P, ψ)

Input: A query graph q, a graph database G, the prefix tree T_F of features F, the `MIFTree` T_P of G and ψ.
Output: the answer set of q R_q and verification object $\mathcal{VO}$.

1: Initialize R_q and the structures in $\mathcal{VO}$ to $\emptyset$ and C_q to G
2: F_q = `find_maxfeatures`(q, T_F) //F_q fully cover q
3: P_q = `find_POF`(q, F_q, T_P) //Enumeration
4: $P_q^{\texttt{opt}}$ = `opt_POF_MWSC`(F_q, P_q)
/* construct $\mathcal{VO}$ of Case 1 of p of N_I */
5: **for each** $p_i \notin P_q^{\texttt{opt}} \wedge p_i \in P_q \cup \{p_r\}$
6: $N_I^v = N_I^v \cup (p_i, \mathcal{H}_{D_{p_i}})$
7: $N_I^b = N_I^b \cup b_i$ /* the $\mathcal{VO}$ of `MHT` of `node`(p_i)'s children */
8: **for each** $p_i \in P_q^{\texttt{opt}}$
9: D_{p_i} = `search`(p_i, T_P); $C_q = C_q \cap D_{p_i}$
/* construct $\mathcal{VO}$ of Case 2 of p of N_I */
10: **for each** $p_i \in P_q^{\texttt{opt}}$
11: L_{p_i} = []
12: **for each** $g_j \in D_{p_i}$
13: **if** $g_j \in C_q$ **then** $L_{p_i} = L_{p_i} \oplus j$ /* append ID */
14: **else** $L_{p_i} = L_{p_i} \oplus (j, \mathcal{H}_{g_j})$ /* append ID and digest */
15: $N_I^v = N_I^v \cup (p_i, L_{p_i})$
16: $N_I^b = N_I^b \cup b_i$ /* the $\mathcal{VO}$ of `MHT` of `node`(p_i)'s children */
/* construct $\mathcal{VO}$ for features F_q */
17: N_F = `construct_NF`(F_q)
/* construct $\mathcal{VO}_{C_q}$ */
18: **for each** $g \in C_q$
19: **if** `subIso`(q, g) = `true`
20: $R_q = R_q \cup g$
/* construct $\mathcal{VO}$ for answer */
21: $I_M = I_M \cup m$, where m is the mapping from q to g.
22: **else** /* construct $\mathcal{VO}$ for non-answer */
23: $C_q^{\bar{R}_q} = C_q^{\bar{R}_q} \cup g$
24: $\mathcal{VO}$ = $((N_I, N_F, \psi), (I_M, C_q^{\bar{R}_q}))$
25: **return** R_q and $\mathcal{VO}$

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 $\mathcal{VO}$ of the `MHT` are (1, $\mathcal{H}_{x_1}$) and (4, $\mathcal{H}_{3,4}$), with which $\mathcal{H}_r$ can be synthesized.

Definition 6.6: The $\mathcal{VO}$ *constitution* of basic authentication for subgraph query is a tuple ($\mathcal{VO}_{\texttt{index}}$, $\mathcal{VO}_{C_q}$), where $\mathcal{VO}_{\texttt{index}} = (N_I, N_F, \psi)$:

- $N_I = (N_I^v, N_I^b)$ is the digest of `MIFTree` nodes, where
  - $N_I^v : \{n_r, n_1, \cdots, n_m\}$, where $P_q = \{p_1, \cdots, p_m\}$ and p_r is the root of `MIFTree`.
    - Case 1: $p_i \notin P_q^{\texttt{opt}}$, $n_i = (p_i, \mathcal{H}_{D_{p_i}})$.
    - Case 2: $p_i \in P_q^{\texttt{opt}}$: $n_i = (p_i, L_{p_i})$, $L_{p_i} : [l_1, \cdots, l_k]$, where `ID`$(D_{p_i}) : [1, \cdots, k]$, and $l_j = j$, if $g_j \in C_q$; otherwise, $l_j = (j, \mathcal{H}_{g_j})$.
  - $N_I^b : \{b_r, b_1, \cdots, b_m\}$, where b_i is the $\mathcal{VO}$ of `MHT` of `node`(p_i)'s children, $p_i \in P_q \cup \{p_r\}$;
- $N_F = (N_F^v, N_F^b)$ is similar to N_I, as F is also organized in a prefix tree T_F ordered by the `mindfs` order. The only difference from `MIFTree` is that each node of the T_F points a feature but not a list of graph IDs; and
- $\psi = \{\psi_F, \psi_I\}$ is the signature of the $\mathcal{DO}$.

$\mathcal{VO}_{C_q}$ = $(I_M, C_q^{\bar{R}_q})$:

- $I_M : \{m_1, \cdots, m_n\}$ is a set of subgraph isomorphism mappings from q to $R_q : \{g_1, \cdots, g_n\}$; and
- $C_q^{\bar{R}_q}$ are non-answer graphs in the candidate set C_q.

VO construction. The $\mathcal{VO}$ of a query is constructed by Alg. 2 at the $\mathcal{SP}$ side. Alg. 2 is Alg. 1 extended with $\mathcal{VO}$ construction: Lines 5-7 and 10-16 for N_I, Line 17 for N_F, and Lines 21-23 for $\mathcal{VO}_{C_q}$. The extension of `find_maxfeatures` with $\mathcal{VO}$ construction is presented in Appendix B and that of `find_POF` is similar (Lines 2-3). As in Alg. 1, to evaluate q, Alg. 2 determines $P_q^{\texttt{opt}}$ from q and T_F (Lines 2-4). In Lines 5-7, for each p_i in P_q or p_r but not in $P_q^{\texttt{opt}}$, it includes (p_i, $\mathcal{H}_{D_{p_i}}$) in N_I^v (Case 1 of Def. 6.6) and b_i in N_I^b, where b_i is the $\mathcal{VO}$ of `MHT` of `node`(p_i)'s children. A subtle remark is that `node`(p_r) 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 $P_q^{\texttt{opt}}$ (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, $(j, \mathcal{H}_{g_j})$ to L_{p_i}. The $\mathcal{VO}$ for p_i is added to N_I^v (Line 15). The construction of N_I^b 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 `POF`s are indexed by prefix trees. Regarding $\mathcal{VO}_{C_q}$, 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 $C_q^{\bar{R}_q}$. The overall $\mathcal{VO}$ is constructed and returned to the client (Lines 24-25).

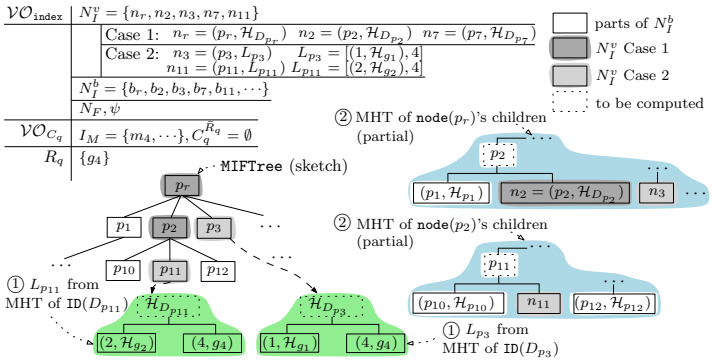


Fig. 9. $\mathcal{VO}$ for basic authentication

Example 6.2: Following up the query processing shown in Fig. 7, Fig. 9 shows the $\mathcal{VO}$ determined by Alg. 2. Recall that $P_q = \{p_2, p_3, p_7, p_{11}\}$ and $P_q^{\texttt{opt}} = \{p_3, p_{11}\}$. Regarding $\mathcal{VO}_{\texttt{index}}$, $N_I^v = \{n_r, n_2, n_3, n_7, n_{11}\}$. $n_r = (p_r, \mathcal{H}_{D_{p_r}})$. $n_2 = (p_2, \mathcal{H}_{D_{p_2}})$ and $n_7 = (p_7, \mathcal{H}_{D_{p_7}})$ since $p_2, p_7 \notin P_q^{\texttt{opt}}$ (Case 1 of Def. 6.6). Since p_3 and p_{11} are in $P_q^{\texttt{opt}}$, $n_3 = (p_3, L_{p_3})$ and $n_{11} = (p_{11}, L_{p_{11}})$ (Case 2 of Def. 6.6). We note that `ID`$(D_3) = [1, 4]$, `ID`$(D_{11}) = [2, 4]$ and $g_4 \in C_q$. Then, ① $L_{p_3} = [(1, \mathcal{H}_{g_1}), 4]$ and $L_{p_{11}} = [(2, \mathcal{H}_{g_2}), 4]$. Since $g_1, g_2 \notin C_q$, only their IDs are needed. Due to space issues, the N_I^b shown is partial. The RHS of Fig. 9 shows ② the (partial) `MHT`s of the children of `node`(p_r) and `node`(p_2). The white boxes indicate the $\mathcal{VO}$ derived from `MHT`s and they are parts of b_r and b_2 in N_I^b. The I_M in $\mathcal{VO}_{C_q}$ is the subgraph isomorphism mapping from q to g_4. Since C_q=R_q=$\{g_4\}$, $C_q^{\bar{R}_q}$ is empty.

Authentication at client. When the client receives R_q and $\mathcal{VO}$, he/she verifies the correctness of R_q. Since the process is similar to Alg. 2 and existing authentication works, we only give an example and highlight the major steps and elaborate Step 4) below, which is unique in `MIFTree`:

1) compute F_q and verify F_q is the maximum individual fully cover features of q by using q, N_F and ψ_F [4];

4. As F is organized in a prefix tree T_F, F_q can be verified by using q, N_F and signature ψ_F in a similar way.

2) compute P_q and verify P_q is consistent to those in N_I by using q, F_q and N_I;
3) determine $P_q^{\texttt{opt}}$ by using F_q and P_q;
4) synthesize $\mathcal{H}_{p_r}$ by using $P_q^{\texttt{opt}}$ and the $\mathcal{VO}$;
5) verify the $\mathcal{H}_{p_r}$ with the signature ψ_I and the public key;
6) determine C_q by intersecting the L_ps from N_I^v, where $p \in P_q^{\texttt{opt}}$; and
7) verify R_q by using I_M; and if I_M is not correct, invokes `subIso`; and verify $C_q^{\bar{R}_q}$ by invoking `subIso`.

In Step 4), the root digest $\mathcal{H}_{p_r}$ 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.* $p \prec p'$. 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* $P_q^{\texttt{opt}}$. n_i of p_i is $(p_i, \mathcal{H}_{D_{p_i}})$. *Case 2* p_i *is in* $P_q^{\texttt{opt}}$. n_i of p_i is (p_i, L_{p_i}). $\mathcal{H}_{D_{p_i}}$ is determined from L_{p_i}, $C_q^{\bar{R}_q}$ and R_q, which contains the IDs, digests and the graphs of D_{p_i}. The remaining part to-be-determined is $\mathcal{H}^r_{p_i}$. In both cases, $\mathcal{H}^r_{p_i}$ is determined from the $\mathcal{VO}$ of `MHT` of $\texttt{node}(p_i)$'s children, in N_I^b. The synthesis must have computed the digest of $\texttt{node}(p_i)$'s children (if it is not already in $\mathcal{VO}$), as the synthesis is defined bottom-up. With $p_i, \mathcal{H}_{D_{p_i}}$ and $\mathcal{H}^r_{p_i}$ (Def. 6.3), the client can recompute $\mathcal{H}_{p_i}$. Then, p_i is removed from P_q. In the recursive step, the synthesis proceeds to another p in P_q with no $p' \in P_q \wedge p \prec p'$. With the $\mathcal{H}_p$ for all $p \in P_q$, $\mathcal{H}_{p_r}$ 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 $\mathcal{H}_{p_r}$ of Fig. 9. To compute $\mathcal{H}_{p_r}$ bottom-up, we may start the synthesis from p_{11} since $p_{11} \in P_q^{\texttt{opt}}$ and $\nexists p' \in P_q$ *s.t.* $p_{11} \prec p'$. We may start at p_3 for a similar reason. Let's start at p_{11}. $n_{11} = (p_{11}, L_{p_{11}})$. $\mathcal{H}_{D_{p_{11}}}$ can be computed from $L_{p_{11}}$ and R_q. The root digest $\mathcal{H}^r_{p_{11}}$ of the `MHT` of $\texttt{node}(p_{11})$'s children can be computed since the `MHT` is empty. $\mathcal{H}_{p_{11}}$ can then be determined from p_{11}, $\mathcal{H}_{D_{p_{11}}}$ and $\mathcal{H}^r_{p_{11}}$. p_{11} is removed from P_q. After that, we may proceed to p_2, since $p_2 \in P_q \wedge p_2 \notin P_q^{\texttt{opt}}$. $n_2 = (p_2, \mathcal{H}_{D_{p_2}})$. We determine $\mathcal{H}^r_{p_2}$ from the computed $(p_{11}, \mathcal{H}_{p_{11}})$ and the $\mathcal{VO}$ of `MHT` of $\texttt{node}(p_2)$'s children such as b_2. In this case, $\mathcal{H}_{p_2}$ is obtained and p_2 is removed from the P_q. We then proceed to p_3. $\mathcal{H}_{p_3}$ is obtained, similar to the synthesis of $\mathcal{H}_{p_{11}}$. $\mathcal{H}_{p_7}$ is synthesized similar to $\mathcal{H}_{p_2}$. With the same logic, $n_r = (p_r, \mathcal{H}_{D_{p_r}})$. With $\mathcal{H}_{p_2}$, $\mathcal{H}_{p_3}$, $\mathcal{H}_{p_7}$ and the $\mathcal{VO}$ of `MHT` of $\texttt{node}(p_r)$'s children, $\mathcal{H}_{p_r}$ 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, $\mathcal{VO}$ sometimes contains excessive graph IDs. In this section, we propose two enhancements on the basic method.

Firstly, *all* graph IDs of each feature $p \in P_q^{\texttt{opt}}$ 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 `MIFTrees` `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 $\mathcal{VO}$s when classical authentication techniques

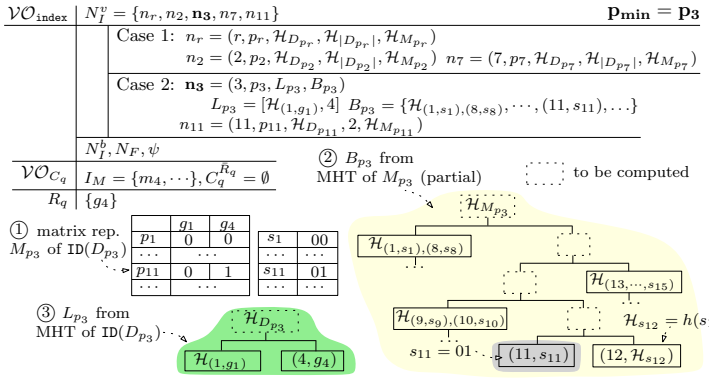


Fig. 10. $\mathcal{VO}$ for enhanced authentication

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 $\mathcal{VO}$.

7.1 Compact Representation of Graph IDs

The main idea to reduce the excessive graph IDs 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* $\texttt{ID}(D_p)$ and M_p to verify the intersections.

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

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

Definition 7.2: The *digest of a node* $\texttt{node}(p)$ is $h(h(id)|h(\texttt{str}(p))|\mathcal{H}_{D_p}|\mathcal{H}_{|D_p|}|\mathcal{H}^r_p|\mathcal{H}_{M_p})$, where

- id is the ID of p, $\texttt{str}(p)$, $\mathcal{H}_{D_p}$ and $\mathcal{H}^r_p$ are the same as in Def. 6.3;
- $\mathcal{H}_{|D_p|}$ is the digest of the size of D_p; and
- $\mathcal{H}_{M_p}$ is the root digest of the classical `MHT` of M_p. The data in the `MHT` are $\{(1, s_i), \cdots, (|P|, s_{|P|})\}$, where for all i, $s_i = M_p(i, *)$.

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

- In Case 2 of N_I^v of $\mathcal{VO}$, for $p_i \in P_q^{\texttt{opt}}$ but $p_i \neq p_{min}$, we include only $n_i = (i, p_i, \mathcal{H}_{D_{p_i}}, |D_{p_i}|, \mathcal{H}_{M_{p_i}})$ in $\mathcal{VO}$, where $|D_{p_i}|$ is used to verify p_{min} in $P_q^{\texttt{opt}}$ 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 $\mathcal{VO}$ of the `MHT` of $\texttt{ID}(D_{p_{min}})$; and (iv) $B_{p_{min}}$ is a set of (i, s_{p_i}), where $(i, s_{p_i}) \in B_{p_{min}}$ if $p_i \in P_q^{\texttt{opt}} \wedge p_i \neq p_{min}$, and the $\mathcal{VO}$ of `MHT` of $M_{p_{min}}$.

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 $D_{p_1} \cap \cdots \cap D_{p_m}$ is equivalent to computing $s_1 \wedge \cdots \wedge s_m$ which very often requires smaller $\mathcal{VO}$. 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 $\mathcal{VO}$ determined by the enhanced method. The differences of $\mathcal{VO}$ from the Example 6.2 are localized in N_I^v. Foremost, $P_q = \{p_2, p_3, p_7, p_{11}\}$ and $P_q^{\texttt{opt}} = \{p_3, p_{11}\}$. Since $|D_{p_3}| = |D_{p_{11}}| = 2$, we just choose p_3 as the 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} is its ② (partial) MHT. Regarding the $\mathcal{VO}$, we first discuss p_{11}. Since p_{11} is in $P_q^{\texttt{opt}}$ but $p_{11} \neq p_{min}$. Thus, $n_{11} = (11, p_{11}, \mathcal{H}_{D_{p_{11}}}, 2, \mathcal{H}_{M_{p_{11}}})$. Next, for p_{min} (*i.e.*, p_3), n_3 of N_I^v is $(3, p_3, L_{p_3}, B_{p_3})$. From previous examples, we have D_{p_3} = $\{g_1, g_4\}$ and C_q = $\{g_4\}$. $L_{p_3} = [\mathcal{H}_{(1,g_1)}, 4]$, where 4 is the graph ID in C_q and $\mathcal{H}_{(1,g_1)}$ is ③ the $\mathcal{VO}$ of the MHT of $\texttt{ID}(D_{p_3})$. Regarding B_{p_3}, only p_{11} is in $P_q^{\texttt{opt}}$ but $p_{11} \neq p_{min}$. Thus, $(11, s_{11})$ is included in B_{p_3}. Finally, the $\mathcal{VO}$ 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 $\mathcal{VO}$ is needed for authenticating the candidate set. In particular, let $\texttt{intv}(M_p, i)$ 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 $\texttt{intv}(M_p, i)$ is the number of ranges needed to be authenticated. To authenticate a range, the upper and lower bounds of the range are needed in $\mathcal{VO}$. 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 $\texttt{ID}(D_p)$ is optimal when intersecting the graphs of other POFs, 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 $\texttt{node}(p)$, the *optimal permutation for* M_p (OPM) is to transform M_p into M_p' by column permutation *s.t.* $\texttt{cost}(M_p') = \sum_{i=1}^{m} \texttt{intv}(M_p, i)$ is minimized, where $|P| = m$ and $|D_p| = n$.

Finding the optimal ordering of graphs of $\texttt{ID}(D_p)$ 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. □

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 $\texttt{ID}(D_p)$) 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 $P_q^{\texttt{opt}}$ is $\{p_i, p_j\}$, $\texttt{ID}(D_{p_i})$ = $[1, 3, 5, 7, 9]$ and $\texttt{ID}(D_{p_j})$ = $[2, 3, 8, 9]$. Assume further p_i and p_j are the only POFs of the database. Then, C_q = $\{g_3, g_9\}$. p_{min} is p_j as $|D_{p_j}|$ = 4 and $|D_{p_i}|$ = 5. Before permutation, the $L_{p_{min}}$ in $\mathcal{VO}$ is $[\mathcal{H}_{(2,g_2)}, 3, \mathcal{H}_{(8,g_8)}, 9]$. In contrast, after the permutation, $\texttt{ID}(D_{p_{min}}) = [2, 8, 3, 9]$. The $L_{p_{min}}$ contains $[\mathcal{H}_{(2,g_2),(8,g_8)}, 3, 9]$.

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.

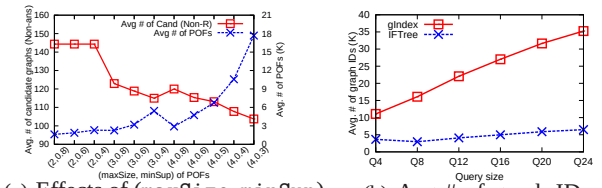
Dataset. Following previous experiments of iGraph, we used the same real-world and synthetic datasets in our experimental evaluation. The real-world dataset consists of 10,000 graphs, all of which are drawn from a real AIDS Antiviral dataset (hereafter denoted as AIDS) [24]. AIDS has been used in many 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.

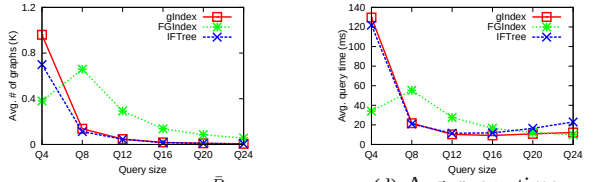
We used gSpan [31] with the default settings [32] on the above two datasets to obtain a set of *discriminative frequent features*, which are served as *individual features* for our experiment.

Query sets. For both AIDS and SYN, the query sets (denoted as Qn) used have been benchmarked in previous works [3], [9], [12], [28], [32], [36]. Each Qn contains 1000 graphs with size (the number of edges) of n, *e.g.*, Q4 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

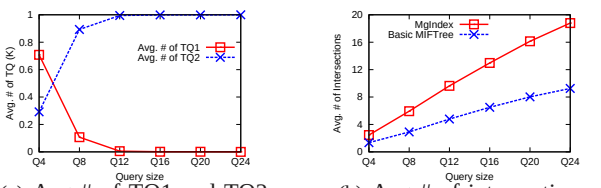


(a) Effects of (maxSize, minSup) (b) Avg # of graph IDs

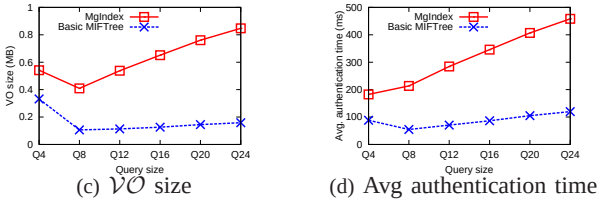


(c) Avg size of $C_q^{\bar{R}_q}$ (d) Avg query time

Fig. 11. I/O cost and query performance



(a) Avg # of TQ1 and TQ2 (b) Avg # of intersections



(c) $\mathcal{VO}$ size (d) Avg authentication time

Fig. 12. Basic authentication method

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 $\mathcal{VO}$, 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 maxSize and minSup of POF. Fig. 11(a) reports the effects of the maximal size (maxSize) and the minimum support (minSup) of POF s by varying maxSize and minSup for Q8 queries. The x-axis is (maxSize, minSup), *e.g.*, (4, 0.5) represents maxSize $= 4$ and maxSup $= 500$. The trends were that when minSup increased *or* 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 $\mathcal{VO}$ size). We set the default values of maxSize and 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 FGIndex 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 $P_q^{\texttt{opt}}$ was small as each $p \in P_q^{\texttt{opt}}$ was chosen by our heuristic discussed in Sec. 5.2. Moreover, the size of each D_p ($p \in P_q^{\texttt{opt}}$) was small.

Average number of non-answer graphs ($C_q^{\bar{R}_q}$) in the candidate set. Fig. 11(c) shows that the average size of $C_q^{\bar{R}_q}$ by varying the query sizes. IFTree produced smaller $C_q^{\bar{R}_q}$ when compared to gIndex and FGIndex in most cases. For example, at Q4, the $C_q^{\bar{R}_q}$ of IFTree contained 27.2% fewer graphs than that of gIndex. At Q24, IFTree resulted in 13.2% fewer graphs. As FGIndex 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, FGIndex produced larger $C_q^{\bar{R}_q}$.

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 $C_q^{\bar{R}_q}$ was large for small queries. The subIso test on those graphs dominated the query time. FGIndex 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 $C_q^{\bar{R}_q}$s of gIndex and IFTree were being similar. However, the benefits of using $P_p^{\texttt{opt}}$ become clear in the experiments on authentication.

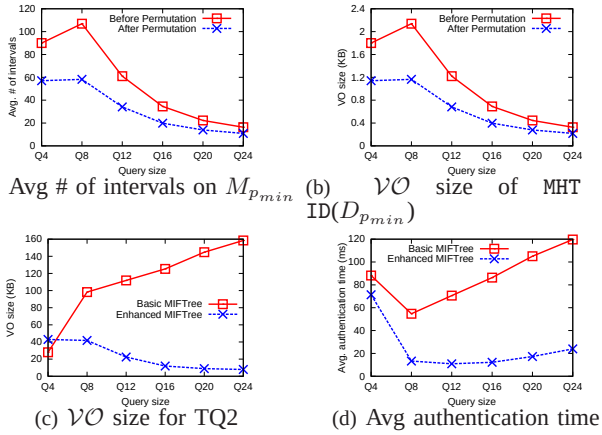
2 Performance of basic authentication

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 $P_q^{\texttt{opt}}$, 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 $\mathcal{VO}$ size. The small number of intersections performed by MIFTree is reflected in the size of $\mathcal{VO}$. Fig. 12(c) shows the $\mathcal{VO}$ 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

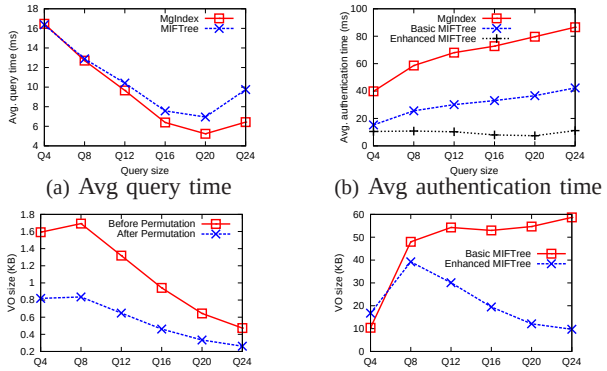
5. Given a query graph q, if q is a feature, *i.e.*, $q = f$, it implies that there is no need to verify the subIso between q and $g \in C_q$ as D_f=C_q=R_q. Such strategy is called "verification-free" [3].



(a) Avg # of intervals on $M_{p_{min}}$ (b) $\mathcal{VO}$ size of MHT of ID($D_{p_{min}}$)

(c) $\mathcal{VO}$ size for TQ2 (d) Avg authentication time

Fig. 13. Enhanced authentication method



(a) Avg query time (b) Avg authentication time

(c) $\mathcal{VO}$ size of MHT of ID($D_{p_{min}}$) (d) $\mathcal{VO}$ size for TQ2

Fig. 14. Authentication performance on synthetic dataset

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 $\mathcal{VO}$ (see Fig. 11(b)). Therefore, $\mathcal{VO}$ enlarged rapidly with query size. For MIFTree, $\mathcal{VO}$ increased with the query size, although at a slower rate. However, since $|P_q^{\texttt{opt}}|$ was often clearly smaller than $|F_q|$ (see Fig. 12(b)) and for each p in $P_q^{\texttt{opt}}$, $|D_p|$ was relatively small (see Fig. 11(b)), MIFTree clearly outperformed MgIndex. Moreover, the $\mathcal{VO}$ of MIFTree did not increase as rapidly as that of MgIndex. We highlight that the $\mathcal{VO}$ size at Q4 was large since the size of non-answers in the candidate set ($C_q^{\bar{R}_q}$) was clearly larger than others, which required some $\mathcal{VO}$ 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.*, $|P_q^{\texttt{opt}}|$ was smaller. Thus, fewer MHTs of ID(D_p) were reconstructed, which is a performance bottleneck during authentication. Further, the sizes of D_ps of $P_q^{\texttt{opt}}$ 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 $\mathcal{VO}$ size due to the MHT of ID($D_{p_{min}}$) 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 $M_{p_{min}}$ for each queries. Recall SubSec. 7.2, the fewer intervals on $M_{p_{min}}$, the smaller $\mathcal{VO}$ size due to the MHT of ID($D_{p_{min}}$). Therefore, Fig. 13(b) reports such $\mathcal{VO}$ size, whose trends were similar to Fig. 13(a). We note that the average size of $\mathcal{VO}$ at Q4 and Q8 increased. The reason was that most of the query features were frequent, then C_q in $D_{p_{min}}$ was relatively large. Therefore, the $\mathcal{VO}$ became larger. At Q12 - Q24, their features contained more infrequent features. Then C_q was relatively small. Hence, the $\mathcal{VO}$ for computing $\mathcal{H}_{D_p}$ decreased with the query size. However, in all queries, the graph permutations of the graph IDs of $D_{p_{min}}$ clearly led to smaller $\mathcal{VO}$ size.

Total $\mathcal{VO}$ size. Fig. 13(c) shows the comparison of $\mathcal{VO}$ sizes between basic method and enhanced method for TQ2. For TQ1, the $\mathcal{VO}$ of enhanced method was almost the same to that of basic method as there was no intersection for TQ1. The figure shows that the enhanced method reduced $\mathcal{VO}$ sizes significantly. For basic method, $\mathcal{VO}$ contained all the graph IDs in ID(D_p) ($p \in P_q^{\texttt{opt}}$) that were needed to be authenticated (see Fig. 11(b)). Instead, $\mathcal{VO}$ for enhanced method contained the $\mathcal{VO}$ of MHT of ID($D_{p_{min}}$) to authenticate. For instance, at Q24, $\mathcal{VO}$ by enhanced method was about 20KB whereas that of the basic method was around 120KB.

Average authentication time. Fig. 13(d) shows the comparison of 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 subiso 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 ID(D_p), $p \in P_q^{\texttt{opt}}$ 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 $M_{p_{min}}$ and ID($D_{p_{min}}$) 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 $\mathcal{VO}$ 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 (maxSize, 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 longer 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 $\mathcal{VO}$s.

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 ID($D_{p_{min}}$). The reasons for the trends were the same to the AIDS. The permutations on ID($D_{p_{min}}$) 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 IFTree that minimizes the I/O cost of the popular filtering-and-verification framework for subgraph query processing. We then proposed MIFTree by extending IFTree to authenticate subgraph query. To optimize the $\mathcal{VO}$ derived from 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

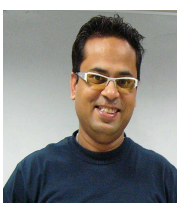
[1] O2I. *http://www.outsource2india.com*, 2013.
[2] Silico. *http://wbbiotech.nic.in/wbbiotech/writereaddata/silicogene.htm*, 2013.
[3] J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: towards verification-free query processing on graph databases. In *SIGMOD*, 2007.
[4] W. Cheng and K.-L. Tan. Query assurance verification for outsourced multi-dimensional databases. *J. Comp. Sec.*, (1):101–126, 2009.
[5] P. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, and S. G. Stubblebine. Flexible authentication of xml documents. In *CCS*, 2001.
[6] M. Y. Galperin and X. M. Fernández-Suarez. The 2012 nucleic acids research database issue and the online molecular biology database collection. *Nucleic Acids Research*, 2012.
[7] P. H. and T. K-L. *Query Answer Authentication*. Morgan & Claypool Publishers, 2012.
[8] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu. igraph: a framework for comparisons of disk-based graph indexing techniques. *PVLDB*, 2010.
[9] H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In *ICDE*, 2006.
[10] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *ICDM*, 2003.
[11] I. Outsourcing. *http://www.informaticsoutsourcing.com*, 2013.
[12] K. Klein, N. Kriege, and P. Mutzel. Ct-index: Fingerprint-based graph indexing combining cycles and trees. In *ICDE*, 2011.
[13] A. Kundu, M. J. Atallah, and E. Bertino. Efficient leakage-free authentication of trees, graphs and forests. *IACR Cryptology ePrint Archive*, 2012:36, 2012.
[14] A. Kundu, M. J. Atallah, and E. Bertino. Leakage-free redactable signatures. CODASPY '12, pages 307–316, 2012.
[15] A. Kundu and E. Bertino. Structural signatures for tree data structures. *PVLDB*, 1, 2008.
[16] A. Kundu and E. Bertino. How to authenticate graphs without leaking. In *EDBT*, 2010.
[17] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *SIGMOD*, 2006.
[18] LKH-2. Lkh-2. *http://www.akira.ruc.dk/ keld/research/LKH/*.
[19] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 2004.
[20] D. A. Menascé. Qos issues in web services. *IEEE Internet Computing*, (6):72–75, 2002.
[21] R. C. Merkle. A certified digital signature. In *CRYPTO*, 1989.
[22] NCBI . Sumit data to NCBI. *http://www.ncbi.nlm.nih.gov/guide/howto/submit-data/*.
[23] NCBI. PubChem. *http://pubchem.ncbi.nlm.nih.gov/*.
[24] NIC. AIDS. *http://dtp.nci.nih.gov/docs/aids/aids_data.html*.
[25] H. Pang, A. Jain, K. Ramamritham, and K.-L. Tan. Verifying completeness of relational query results in data publishing. In *SIGMOD*, 2005.
[26] H. Pang and K. Mouratidis. Authenticating the query results of text search engines. *PVLDB*, 2008.
[27] S. Papadopoulos, Y. Yang, and D. Papadias. Continuous authentication on relational streams. *The VLDB Journal*, (2):161–180, 2010.
[28] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 2008.
[29] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, 2002.
[30] H. Wang, J. Li, J. Luo, and H. Gao. Hash-base subgraph query processing method for graph-structured xml documents. *PVLDB*, 2008.
[31] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM*, 2002.
[32] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *SIGMOD*, 2004.
[33] Y. Yang, S. Papadopoulos, D. Papadias, and G. Kollios. Spatial outsourcing for location-based services. In *ICDE*, 2008.
[34] K. Yi, F. Li, G. Cormode, M. Hadjieleftheriou, G. Kollios, and D. Srivastava. Small synopses for group-by query verification on outsourced data streams. *ACM Trans. Database Syst.*, (3):15:1–15:42, 2009.
[35] M. L. Yiu, Y. Lin, and K. Mouratidis. Efficient verification of shortest path search via authenticated hints. In *ICDE*, 2010.
[36] D. Yuan and P. Mitra. Lindex: a lattice-based index for graph databases. *The VLDB Journal*, (2):229–252, 2012.
[37] L. Zou, L. Chen, J. X. Yu, and Y. Lu. A novel spectral coding in a large graph database. In *EDBT*, 2008.

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 Sourth 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/∼db/).

Yun Peng is a PhD student in the Department of Computer Science, Hong Kong Baptist University. He received his BSci 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/∼db/).

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.*, $S' = P_q^{\texttt{opt}}$. That means finding such collection S', which is the optimal decomposition $P_q^{\texttt{opt}}$ 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 $p = (v_1, v_2, \cdots, v_{n-1}, v_n)$ and $n = |V|$. For all $i, j \leq n$, $\texttt{dist}(M, i, j) = w(v_i, v_j)$, where $\texttt{dist}(M, i, j) = |M(*, i) \oplus M(*, j)|$. Specially, $M(*, 1) = M(*, n) = \mathbf{0}$. 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 $\texttt{cost}(M')$ is the minimized. Thus the permutation of columns of M' is denoted as $p' = (v_1, v'_2, \cdots, v_{n-1}, v_n)$. Then we can get a path $\mathcal{P}$ on K in terms of p', where $\mathcal{P} = (v_1, v'_2, \cdots, v'_{n-1}, v_n)$. Since $\texttt{cost}(M')$ is the minimized, and $\texttt{cost}(M') = \frac{1}{2} \sum_{i=1}^{n-1} \texttt{dist}(M', i, i+1) = \frac{1}{2} \sum_{v'_i \in p', i=1}^{n-1} w(v'_i, v'_{i+1})$. Therefore $\sum_{v'_i \in p', i=1}^{n-1} w(v'_i, v'_{i+1})$ is minimized. In this case, $\mathcal{P}$ 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 $P_q^{\texttt{opt}}$.

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 ψ_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 $P_q^{\texttt{opt}}$. P_q and $P_q^{\texttt{opt}}$ 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 POFs. 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_POF 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 $P_q^{\texttt{opt}}$ using opt_POF_MWSC.

With the correct of F_q, P_q and $P_q^{\texttt{opt}}$, 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:

- $\texttt{subIso}(q, g) = \texttt{false}$: this is detected when the client performs subIso by using the isomorphic mapping I_M (in Step 7) of authentication); or
- $\texttt{subIso}(q, g) = \texttt{true}$: 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 $\mathcal{H}_{p_r}$ is not generated correctly and the client can detect this with the signature ψ_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:

- $g \in C_q^{\bar{R}_q}$: This is detected when the client performs subIso on all graphs in $C_q^{\bar{R}_q}$ (in Step 7) of authentication); or
- $g \notin C_q^{\bar{R}_q}$: Denote $P_q^{\texttt{opt}}$ as $\{p_1, \cdots, p_m\}$. Since $P_q^{\texttt{opt}}$ is correct and g is an answer, g contains an instance of $P_q^{\texttt{opt}}$. There are only two cases that lead to $g \notin C_q^{\bar{R}_q} \wedge g \notin R_q$:
  - $\exists p_i$, $g \notin D_{p_i}$. In this case, $\mathcal{SP}$ had modified D_{p_i}. Hence, $\mathcal{H}_{p_i}$ and $\mathcal{H}_{p_r}$ cannot be synthesized correctly, and this will be detected when comparing the signature ψ_I; or
  - $\forall p_i$, $g \in D_{p_i}$. While the $\mathcal{SP}$ may perform the intersections of all D_{p_i}s incorrectly, the client will performs the intersections on all L_{p_i}, $p_i \in P_q^{\texttt{opt}}$ (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_q, P_q and $P_q^{\texttt{opt}}$ 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:

- $g \in C_q^{\bar{R}_q}$: This is detected when the client performs `subIso` on all graphs in $C_q^{\bar{R}_q}$; or
- $g \notin C_q^{\bar{R}_q}$: Denote $P_q^{\texttt{opt}}$ as $\{p_1, \cdots, p_m\}$. g contains instances of $P_q^{\texttt{opt}}$ as g is an answer. There are again two cases that lead to $g \notin C_q^{\bar{R}_q} \wedge g \notin R_q$:
  - $\exists p_i$, $g \notin D_{p_i}$. In this case, $\mathcal{SP}$ had modified D_{p_i}, *i.e.*, M_{p_i} had been modified. Hence, $\mathcal{H}_{M_{p_i}}$ cannot be synthesized correctly which leads to wrong $\mathcal{H}_{p_r}$, and the client will be alerted when comparing the $\mathcal{DO}$ signature; or
  - $\forall p_i$, $g \in D_{p_i}$. The $\mathcal{SP}$ has performed the intersection incorrectly. However, this is detected when the client performs the conjunctions on all s_i, $p_i \in P_q^{\texttt{opt}}$.

□

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 $\mathcal{VO}$. It first finds all the maximal features F_q by using `find_maxfeatures` (Line 2). `find_maxfeatures` (to be elaborated next) is a traversal on the search tree T_F that constructs $\mathcal{VO}_{\texttt{index}}$. After computing F_q, it then computes the candidate set C_q (Lines 3-4) by intersections. In Lines 5-10, $\mathcal{VO}_{\texttt{index}}$ is modified according to the candidate answer determined in Lines 3-4. The construction of $\mathcal{VO}_{C_q}$ is presented in Lines 11-15. Finally, Alg. 3 generates the total $\mathcal{VO}$ 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_q of q. In Alg. 4, the only difference from [32] is that it needs to record the $\mathcal{VO}_{\texttt{index}}$ while searching for the features, which will be used in Alg. 3. In Line 1, $\mathcal{VO}_{\texttt{index}}{}'$ is the $\mathcal{VO}_{\texttt{index}}$ 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 `MgIndex` (Lines 3-5) in $\mathcal{VO}_{\texttt{index}}$, 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 $\mathcal{VO}_{\texttt{index}}$. 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 $\mathcal{VO}_{\texttt{index}}$(Lines 16-18). Then, `auth_DFS` proceeds to each child c 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_r. The child nodes of f_r are the current boundary nodes and they are recorded in $\mathcal{VO}_{\texttt{index}}{}'$ (Lines 3-5). The search proceeds to the child nodes of f_r with the minimum `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 $\mathcal{VO}_{\texttt{index}}$ (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 $\mathcal{VO}_{\texttt{index}}{}'$.

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 `MIFTree` 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, $\mathcal{VO}$

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

Algorithm 4 find_maxfeatures (q, T_F, $\mathcal{VO}_{\texttt{index}}$)

Input: q is a query graph, the prefix tree T_F of features F, $\mathcal{VO}_{\texttt{index}}$ is the $\mathcal{VO}$ 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 child f of f_r in T_F    /* f_r is the root node of T_F */
4:     VO_index'[f] = (f, H_f)    /* boundary nodes */
5: VO_index[f_r] = (f_r, VO_index' )    /* visited nodes */
6: for each edge e in S^1, e ∉ 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 ($e, s, S, \mathcal{VO}_{\texttt{index}}, q, T_F, U$)

```
11: if s ≠ mindfs(s)
12:     return
13: if VO_index[s] ≠ ∅    /* s is a feature in T_F */
14:     U = U ∪ e    /* mark e as covered */
15:     S = 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 c of s in q ∧ q ≠ U
21:         e' = c − s
22:         auth_DFS (e', c, S, VO_index', q, T_F, U)
```

Software setting. We implemented the seminal 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 $\mathcal{SP}$ or $\mathcal{DO}$ is required to send the whole database and the $\mathcal{DO}$'s signature to the client, for each query. The client can verify the integrity of the database with $\mathcal{DO}$'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 $\mathcal{VO}$. 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 one 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.

TABLE 1
Frequently Used Symbols

Symbol	Description
q, g, G	a query graph, a graph data and a graph database
f	an individual feature
F	a set of all individual features in G or the prefix tree that indexes all features of G
p, P	a partially overlapping feature POF and a set of all POFs in G
D_p	a set of graphs that each of which contains p
ID(D_p)	a list of IDs of the graphs in D_p
P_q, P_q^{opt}	the POFs of q and the optimal decomposition of POFs
C_q, R_q, C_q^{Rq}	the candidate set, the answer set and the non-answer candidate set
$\mathcal{H}_g$, $\mathcal{H}_p$	a digest of graph g and node(p) of MIFTree
$\mathcal{H}_{D_p}$	a root digest of the classical MHT on all graphs in D_p
$\mathcal{H}_p^r$	a root digest of the embedded MHT on node(p)'s child nodes
$\mathcal{VO}_{\texttt{index}}$	the digests that record the search of features
$\mathcal{VO}_{C_q}$	the non-answer graphs in the candidate set
N_I, N_F	the digest of MIFTree nodes and the digest of prefix F nodes
I_M	a set of subgraph isomorphism mappings from q to $g \in R_q$
ψ_I, ψ_F	a signature of MgIndex and MIFTreefrom $\mathcal{DO}$
p_{min}	the minimal POF in P_q^{opt}
M_p	a binary matrix of ID(D_p) of p (for minimizing I/O)
$\mathcal{H}_{M_p}$	a root digest of the classical MHT on ID(D_p)

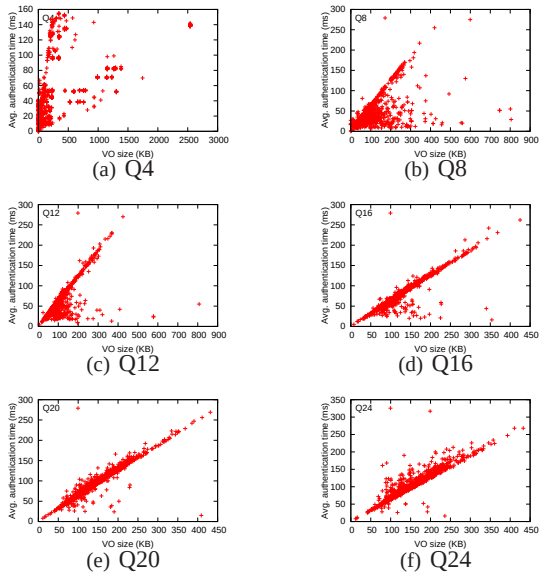
APPENDIX D
FREQUENTLY USED SYMBOLS

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

APPENDIX E
$\mathcal{VO}$ 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 $\mathcal{VO}$ size and authentication time. The reason for not using enhanced authentication method is that its $\mathcal{VO}$ 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



(a) Q4 (b) Q8 (c) Q12 (d) Q16 (e) Q20 (f) Q24

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

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 `IFTree` to address efficient authenticated subgraph query processing.