

Article

DFB-PPGSQ: Dynamic Forward-Private and Epochal Backward-Private Graph Similarity Matching Query over Encrypted Sensor Graph Databases

Huiying Hou ^{1,2,3,*} , Yucong Ma ², Zisu Zhao ² and Xinrui Ge ⁴ ¹ The Key Laboratory of Grain Information Processing and Control, Henan University of Technology, Zhengzhou 450001, China² The School of Artificial Intelligence and Big Data, Henan University of Technology, Zhengzhou 450001, China; yema@stu.haut.edu.cn (Y.M.); 251210200819@stu.haut.edu.cn (Z.Z.)³ Henan Province Key Laboratory of Cyberspace Situation Awareness, Zhengzhou Institute of Information Science and Technology, Zhengzhou 450001, China⁴ School of Computer Science and Technology, Qingdao University, Qingdao 266071, China; xinrui@qdu.edu.cn

* Correspondence: hyhou@haut.edu.cn

Abstract

In applications pertaining to sensor network, the Internet of Things, industrial monitoring, and cyber-physical security, graphs are increasingly being outsourced to clouds, where similarity search should be supported without exposing graph content, query graphs, update contents, or database evolution. Existing privacy-preserving graph similarity schemes mainly target static encrypted databases, and as such struggle to handle insertions, deletions, label updates, and long-running index maintenance. This paper proposes DFB-PPGSQ, a dynamic forward-private and epochal backward-private graph similarity matching scheme that moves branch-based lower-bound filtering into a structured encryption framework. DFB-PPGSQ uses epoch-local feature tokens, per-record occurrence handles, one-time update labels, update buffers, deletion tombstones, and shuffle-based branch-tree re-randomization to preserve pruning efficiency while making same-epoch tombstone, traversal, size, timing, and refresh leakage explicit. We formalize the system model, leakage functions, algorithms, and security interpretation, then implement a reproducible Python prototype with HMAC-SHA256 token generation and multi-profile dynamic sensor-topology workloads. Across five random seeds, DFB-PPGSQ keeps server-side filtering latency close to the static branch-tree baseline (46.60 ms versus 44.72 ms at 4000 graphs), avoids immediate full-rebuild updates (0.091 ms insertion and 0.092 ms label update), and keeps metadata-assisted cross-epoch token linkage below 5.6% attack success after refresh in additional industrial and campus IoT stress workloads. Storage, communication, exact GED refinement, side-channel hardening, and verifiable-result protection are treated as deployment costs and limitations rather than being included in the headline latency.

Keywords: encrypted graph database; graph similarity matching; structured encryption; searchable encryption; forward privacy; epochal backward privacy; dynamic graph update; IoT security; sensor cloud



Academic Editor: Alexios Mylonas

Received: 18 June 2026

Revised: 16 July 2026

Accepted: 20 July 2026

Published: 28 July 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and

conditions of the [Creative Commons Attribution \(CC BY\) license](https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Graphs provide a natural data structure for describing entities and their relationships. In sensor networks, vertices can represent deployed sensors and edges can describe communication links. In Industrial IoT contexts, vertices can denote controllers, actuators, events, or data streams, while edges can express operational dependencies. In cyber-physical security, a graph may capture provenance paths, alert correlations, or lateral movement traces. Such graph data are valuable, but are also sensitive. Sensor applications are often resource-constrained and security-sensitive, meaning that privacy mechanisms must remain lightweight enough for practical IoT deployments [1].

Cloud computing provides an attractive platform for storing and querying large graph databases. A client can outsource the graph database and obtain scalable computation. However, the cloud and the client usually belong to different trust domains; the cloud can be honest in executing protocols, but curious about graph structures, labels, queries, and query results. Therefore, graph data are often encrypted before outsourcing. A meaningful encrypted graph service should still support useful operations such as graph similarity matching queries.

Graph similarity matching retrieves data graphs for which the distance to a query graph is no larger than a user-specified threshold, a more tolerant approach than exact graph matching. This tolerance is necessary in practical sensing and monitoring applications where data collection is noisy, device states are incomplete, labels can change over time, and inter-sensor dependencies may drift in dynamic environments [2]. Where a strict exact match might miss useful results, a similarity query can recover them.

Recent privacy-preserving graph similarity matching query schemes have made important progress. PGSim and PrigSim protect graph similarity search over encrypted data by combining graph filters and secure query processing [3,4]. The recent PPGSQ scheme improves efficiency by adopting a branch-based lower bound of edit distance and a branch-tree index [5]. Its core idea is to encode each local neighborhood as a branch consisting of a center vertex, incident edge labels, and neighboring vertex labels. Since a single graph edit operation affects a limited number of branches, branch-based filtering gives a tighter lower bound than a global label filter and can prune more mismatched data graphs.

However, current PPGSQ-style systems are primarily static. This is a narrow assumption for real cloud graph services. Sensor cloud databases change continuously; the database evolves as new devices join, old devices disappear, labels are corrected, abnormal events are reclassified, and communication edges are removed.

This dynamic setting changes the security problem. If encrypted branch tags remain stable, then the cloud can link updates to previous queries. Deleted graphs can leave stale index traces, future queries may reveal information about removed objects, and label updates implemented as delete-and-insert operations without privacy control require the cloud to infer which branch features have changed. In long-running services, even small leakages accumulate. Therefore, the resulting challenge involves not just computational efficiency but temporal privacy.

This paper asks the following question:

Can we support efficient graph similarity matching over encrypted graph databases while allowing insertions, deletions, label updates, and periodic index re-randomization with forward privacy and epochal delay-bounded backward privacy?

We answer this question by proposing DFB-PPGSQ. The scheme keeps the useful branch-based filtering principle but redesigns the encrypted index as a dynamic structured encryption object. The index evolves across epochs. Update tokens are one-time and unlinkable across time. Deleted records are represented by bounded tombstone information,

and are removed or re-randomized during epoch compaction. The branch–tree index is preserved for efficient pruning, but its labels and counters are refreshed by epoch keys. The design does not claim volume hiding, result size hiding, oblivious traversal, or strong hidden-deletion backward privacy; instead, it states the leakage profile explicitly and evaluates the cost of reducing stable cross-epoch token equality.

The main contributions of this paper are summarized as follows:

1. We formulate dynamic forward-private and epochal backward-private PPGSQ. We extend the static graph similarity matching query model to support graph insertion, graph deletion, vertex/edge label updating, and periodic re-randomization. We define the syntax, threat model, leakage functions, and privacy goals for dynamic encrypted graph similarity services.
2. We design a dynamic branch-based encrypted index. DFB-PPGSQ constructs epoch-local feature tokens, per-record occurrence handles, encrypted branch-count vectors, update buffers, deletion tombstones, and a re-randomizable branch–tree index. The construction supports efficient filtering without forcing the cloud to traverse the whole database after each update.
3. We integrate forward privacy and epochal backward privacy with graph filtering. Search tokens generated before an update cannot be used to identify future inserted or updated branch features. Future queries reveal only bounded information about deleted graphs. Periodic re-randomization further reduces long-term linkability of branch identifiers and tree nodes.
4. We provide a reproducible prototype evaluation. We implement the protocol-relevant operations of DFB-PPGSQ in Python, including branch extraction, HMAC-based token generation, update buffering, tombstone filtering, epoch refresh, and generated dynamic sensor topology workloads. The evaluation compares DFB-PPGSQ with several baselines: static scan, static branch–tree, feature-only dynamic SSE, rebuild-after-update, and no re-randomization.

The remainder of this paper is organized as follows: Section 2 reviews related work; Section 3 presents the system model, threat model, notation, and security definitions; Section 4 introduces preliminaries; Section 5 gives the detailed construction of DFB-PPGSQ; Section 6 provides the security analysis; Section 7 presents the prototype implementation and experimental results; finally, Section 8 concludes the paper.

2. Related Work

2.1. Privacy-Preserving Graph Similarity Query

A graph similarity query is a fundamental graph database operation. Given a query graph and an edit distance threshold, the server retrieves data graphs that approximately match the query. Graph edit distance has a long history in attributed graph matching and pattern recognition [6,7]. Since exact graph edit distance computation is expensive, practical systems usually follow a filtering-and-refinement framework [8–11]. A lower bound is first computed to prune impossible candidates. The remaining graphs are then refined by a more expensive distance computation.

Encrypted graph similarity querying brings an additional constraint in which the cloud should not see plaintext labels, graph structures, or query graphs. PGSim uses pivot-based filtering to support privacy-preserving graph similarity queries over encrypted outsourced data [3]. PrigSim improves the service model and considers graph similarity search in cloud settings [4]. The recent PPGSQ design replaces heavy pivot computations with branch-based lower-bound filtering and a secure branch–tree index, which improves pruning ability and query efficiency [5].

These schemes are important foundations, yet they mainly assume that the encrypted graph database is static after setup. In practice, sensor and IoT graphs are dynamic. A static index must be rebuilt after each update or must accept privacy leakage caused by stable encrypted identifiers, neither of which provides a satisfying solution.

2.2. Dynamic Searchable Symmetric Encryption

Searchable encryption allows a client to outsource encrypted data while retaining search capability [12–15]; later systems provided improved scalability and expressive queries [16]. Dynamic searchable symmetric encryption further supports document insertion and deletion [17–20]. Forward privacy, introduced to prevent newly added documents from being linked to previous search tokens, has become a standard requirement for dynamic encrypted search [21]. Backward privacy limits what future searches reveal about deleted documents [22]. Stronger backward privacy notions reduce leakage from deleted records, but often require higher update cost, server-side cleanup, or more complex key evolution.

The DFB-PPGSQ problem is related to dynamic searchable symmetric encryption, but is not a direct keyword search problem. A graph similarity query touches many branch features, computes a lower bound, and traverses a tree-shaped index. Insertions, deletions, and label updates modify structured objects rather than independent documents. Therefore, dynamic searchable encryption ideas must be adapted to graph features, branch counts, candidate filtering, and tree pruning.

2.3. Secure Graph Query Processing

Secure graph query processing covers shortest path queries, reachability queries, sub-graph matching, and graph analytics over encrypted graphs. Privacy-preserving encrypted graph query processing was explored by Cao et al. [23]. GRECS and later graph encryption schemes show that specialized encrypted graph indexes can support useful graph operations [24]. GraphShield considers dynamic secure graph queries with forward privacy [25]. The recent PathGES improves secure shortest-path query processing [26]. These systems demonstrate the value of graph-specific encrypted indexes.

Recent work on privacy-preserving multi-user graph intersection in cloud-assisted IoT published in *Sensors* further shows that graph operations over outsourced wireless/IoT data are becoming a concrete sensor cloud requirement [27]. However, shortest path querying, graph intersection, and graph similarity matching expose different computation patterns. A similarity query compares a query graph with many candidate graphs under an edit distance threshold, which requires feature-level counting, lower-bound computation, and pruning. DFB-PPGSQ targets this specific operation.

2.4. Leakage Attacks and Motivation for Re-Randomization

Encrypted search systems often leak access patterns, search patterns, result sizes, update sizes, and sometimes order information. Leakage abuse and inference attacks show that such information can be exploited with auxiliary knowledge [28–31]. Order-revealing encryption and deterministic encrypted identifiers may further expose ordering or equality relations [32]. For static encrypted graph indexes, repeated queries may reveal stable branch or node correlations.

Dynamic graph services intensify this risk. The same branch feature may appear across many epochs, while a deleted graph may still influence future result sizes. A label update may reveal a before-and-after relationship. Therefore, DFB-PPGSQ adds periodic re-randomization, as shown in Table 1. This does not eliminate all leakage, but does reduce persistent linkability while giving the data owner a tunable mechanism for limiting leakage accumulation. The refresh operation itself also leaks migration size, bucketed

record sizes, refreshed tree shape, and timing; these observables are treated as part of the re-randomization leakage rather than as hidden quantities.

Table 1. Functional and privacy comparison with related encrypted graph querying schemes.

Scheme	Mechanism Coverage						Residual Risk
	Sim.	Dyn.	Label	Re-Rand.	Fwd.	Bwd.	
PGSim [3]	•	×	×	×	×	×	stable index leakage
PrigSim [4]	•	×	×	×	×	×	full-database traversal leakage
PPGSQ [5]	•	×	×	○	×	×	static branch-token linkage
GraphShield [25]	×	•	○	○	•	○	not designed for similarity filtering
Dynamic SSE index [17,21,22]	○	•	○	○	•	○	weak graph-structure pruning
DFB-PPGSQ	•	•	•	•	•	○	same-epoch tombstone leakage

•: directly supported; ○: bounded, indirect, or assumption-dependent support; ×: outside the current mechanism. Sim.: graph similarity query; Dyn.: insertion/deletion; Label: vertex/edge label updates; Re-Rand.: periodic re-randomization or compaction.

3. Problem Formulation

3.1. System Model

The system model contains two primary entities: the client and the cloud. It can also include authorized users in a multi-user deployment, but this paper focuses on the single-client setting first. This scope matches the basic SSE model and is intentionally narrower than multi-owner, multi-gateway, and multi-policy sensor cloud deployments. Extending the construction to those deployment models requires additional authorization, revocation, and policy enforcement layers, and is left as future work.

3.1.1. Client

The client owns a dynamic graph database. They extract graph features, generates cryptographic keys, encrypts graph records, constructs secure indexes, and issues update/query tokens. The client is trusted, and may be a sensor data owner, IoT service operator, or security monitoring platform.

3.1.2. Cloud

The cloud stores encrypted graphs and secure indexes, receives update tokens and query tokens from the client, executes update, filtering, pruning, and candidate retrieval procedures on encrypted data, and returns encrypted candidate graphs to the client.

Figure 1 summarizes the system-level interaction. The trusted client keeps plaintext graphs, master keys, and workload semantics locally, uploading only encrypted graph records and protected index material. Later, the client sends update tokens, query tokens, and epoch re-randomization material to the cloud. The cloud is honest-but-curious. It stores the encrypted graph database, branch-tree index, update buffer, tombstone set, and re-randomization state while performing token-authorized filtering and returning encrypted candidates for local decryption and optional refinement.

The dynamic graph database at logical time t is denoted by

$$\mathcal{G}^t = \{G_1^t, G_2^t, \dots, G_{N_t}^t\}. \quad (1)$$

Each graph $G = (V, E, L_V, L_E)$ contains a vertex set V , edge set E , vertex label function L_V , and edge label function L_E . A query graph is denoted by Q . The query threshold is denoted by τ . The query result consists of encrypted graphs for which the graph edit distance to Q is no larger than τ .

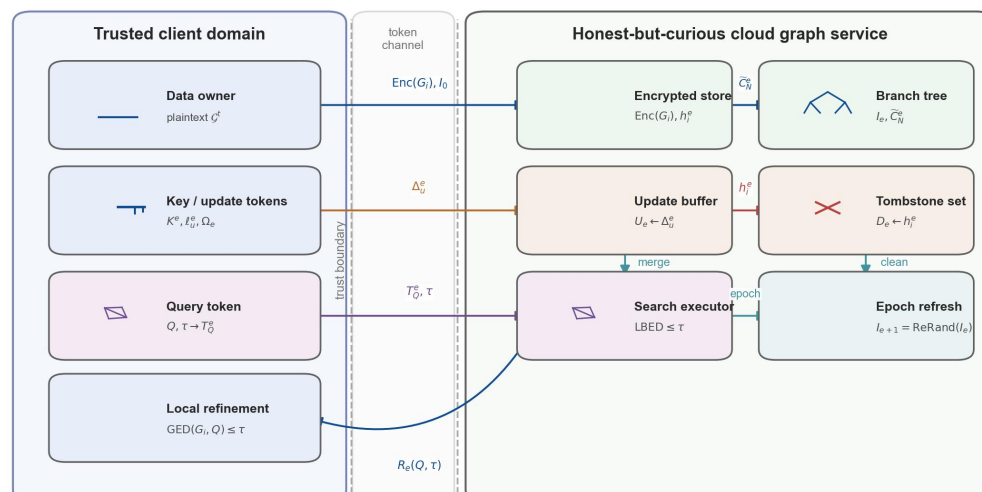


Figure 1. System model of DFB-PPGSQ. The figure separates the trusted client domain from the honest-but-curious cloud graph service and shows the the setup, update, query, results returning, and epoch refreshing flows.

DFB-PPGSQ supports the following operations:

- **Insert:** Adds a new graph G to the encrypted database.
- **Delete:** Removes an existing graph G_i from the active database.
- **UpdateLabel:** Modifies one or more vertex labels or edge labels in an active graph.
- **Search:** Retrieves encrypted candidate graphs that satisfy the similarity threshold.
- **ReRand:** Re-randomizes feature tokens, occurrence handles, tree nodes, counters, and active-record handles at an epoch boundary.

Topology-changing updates such as edge insertion, edge deletion, or local device neighborhood changes can be represented by recomputing the affected one-hop branches and issuing the same branch-delta update when the affected local neighborhood is available to the client. If a deployment cannot identify the local affected set, the conservative fallback is delete-and-insert, which preserves correctness but increases update cost and weakens continuity. The prototype evaluation focuses on whole-graph insertion/deletion and label updates; explicit edge update engineering is a limitation of the current implementation.

3.2. Threat Model

The cloud is honest-but-curious. It follows the protocol, but attempts to infer information from encrypted graphs, secure indexes, update tokens, query tokens, search traces, access patterns, result sizes, tree traversal paths, and timing metadata. This is the same broad server model used by many searchable encryption systems.

The client is trusted. It generates keys correctly and does not leak plaintext graph data. Network channels between the client and authorized users can be protected by standard authenticated encryption. Denial-of-service attacks, malicious result omission, collusion between users, and side-channel leakage from implementation details are outside the main model. Accordingly, the present security analysis should be read as honest-but-curious leakage-based security, not as a guarantee of resistance to malicious server, verifiable computation, or side-channel attacks.

The cloud may possess auxiliary knowledge, and may know public graph distributions, common sensor labels, partial plaintext graphs, or historical workload statistics. This knowledge makes leakage control more important.

Side-channel attacks, including timing, power, and electromagnetic leakage, are outside the current threat model. The honest-but-curious adversary is assumed to observe

only the protocol messages and the leakage functions defined in Section 6.1, not physical side channels.

3.3. Notation

Table 2 lists the main symbols used in this paper.

Table 2. Notation used in the DFB-PPGSQ construction and security analysis.

Symbol	Description
λ	Security parameter
$\mathcal{G}^t$	Dynamic graph database at time t
G_i	The i -th data graph
Q	Query graph
τ	Graph edit distance threshold
$B(G)$	Branch multiset extracted from graph G
b	A branch feature
e	Current epoch
K_e	Epoch key
θ_b^e	Epoch-local query-matchable feature token of branch b
$\eta_{b,i,c}^e$	Per-record occurrence handle for the c -th occurrence of branch b in graph G_i
I_e	Encrypted branch–tree index in epoch e
U_e	Encrypted update buffer in epoch e
D_e	Deletion tombstone set in epoch e
LBED	Branch-based lower bound of edit distance
op	Dynamic operation type

3.4. Scheme Definition

A DFB-PPGSQ scheme consists of the following polynomial-time algorithms:

$$\Pi = (\text{KeyGen}, \text{Setup}, \text{UpdateToken}, \text{ApplyUpdate}, \text{TrapGen}, \text{Query}, \text{ReRand}). \quad (2)$$

3.4.1. $\text{KeyGen}(1^\lambda) \rightarrow K$

Given the security parameter, the client generates master keys for encryption, pseudo-random functions, pseudo-random permutations, order or comparison modules, and epoch evolution.

3.4.2. $\text{Setup}(K, \mathcal{G}^0) \rightarrow (\text{EDB}_0, I_0, \text{st}_0)$

The client encrypts the initial graph database, extracts branch multisets, builds the initial secure branch–tree index, and initializes the client state.

3.4.3. $\text{UpdateToken}(K, \text{st}, \text{op}) \rightarrow \Delta$

For an insertion, deletion, or label update, the client generates an update token Δ . The token contains only encrypted feature handles and masked metadata needed by the cloud.

3.4.4. $\text{ApplyUpdate}(\text{EDB}, I, U, D, \Delta) \rightarrow (\text{EDB}', I', U', D')$

The cloud applies the update token. It inserts encrypted records into the update buffer, marks deleted records, or updates encrypted label-derived branch features.

3.4.5. $\text{TrapGen}(K, \text{st}, Q, \tau) \rightarrow T_Q$

The client extracts the branch multiset of the query graph and generates a query token for the current epoch and active update buffers.

3.4.6. $\text{Query}(\text{EDB}, I, U, D, T_Q) \rightarrow R$

The cloud computes encrypted branch lower bounds, prunes candidate graphs using the branch-tree index, checks active-record states, and returns encrypted candidates.

3.4.7. $\text{ReRand}(K, \text{st}, e) \rightarrow \Omega_e$

At an epoch boundary, the client generates a shuffled refresh package. The cloud replaces the old branch tree, active-record handles, and update buffers with the new package, but it does not receive an old-handle to new-handle correspondence table.

3.5. Privacy Goals

DFB-PPGSQ aims to achieve content privacy, query privacy, forward privacy, epochal backward privacy, and bounded re-randomization leakage.

3.5.1. Content Privacy

The cloud should not learn plaintext graph structures, vertex labels, edge labels, query graphs, or update contents beyond the defined leakage functions.

3.5.2. Query Privacy

The cloud should not learn the plaintext query graph. Depending on the selected security performance configuration, it may learn bounded query leakage such as result size and a protected traversal trace.

3.5.3. Forward Privacy

Search tokens issued before an insertion or label update should not enable the cloud to identify whether the newly inserted or newly updated branch features match previous queries. In other words, future updates are protected from past searches.

3.5.4. Backward Privacy

After a graph is deleted, future search tokens should not reveal information about that deleted graph beyond bounded deletion leakage. The strongest form hides the deleted graph from future results and search traces after compaction. A practical form allows temporary tombstone leakage within the current epoch and removes it at re-randomization.

3.5.5. Re-Randomization Privacy

Across epochs, refreshed feature tokens, occurrence handles, tree-node handles, and active-record handles should not be linkable by the cloud except through explicitly leaked size buckets, update timing, and migration counts.

4. Preliminaries

4.1. Branch-Based Lower Bound of Edit Distance

A branch is a local graph feature centered at a vertex. It contains the center vertex label, the labels of incident edges, and the labels of neighboring vertices. The branch multiset of graph G is denoted by $B(G)$.

For a branch b and a vertex label ℓ , the branch and vertex multiplicities of G are written as

$$C_B(G, b) = |\{b' \in B(G) : b' = b\}|, \quad C_V(G, \ell) = |\{v \in V(G) : L_V(v) = \ell\}|. \quad (3)$$

Let $\mathcal{B}_{G,Q} = B(G) \cup B(Q)$ and $\mathcal{L}_{G,Q} = L_V(G) \cup L_V(Q)$ be the union domains induced by a data graph and a query graph. The branch and vertex residuals are

$$\Delta_B(G, Q) = \sum_{b \in \mathcal{B}_{G,Q}} |C_B(G, b) - C_B(Q, b)|, \quad (4)$$

$$\Delta_V(G, Q) = \sum_{\ell \in \mathcal{L}_{G,Q}} |C_V(G, \ell) - C_V(Q, \ell)|. \quad (5)$$

Following the branch-based filtering principle, a computable lower-bound score is

$$\text{LBED}(G, Q) = \max \left\{ \left\lceil \frac{\Delta_B(G, Q)}{2} \right\rceil, \left\lceil \frac{\Delta_V(G, Q)}{2} \right\rceil \right\}. \quad (6)$$

If $\text{LBED}(G, Q) > \tau$, then G cannot be a valid answer and can be pruned before exact refinement. This score is not used as a final graph edit distance estimator but as a safe filtering signal.

This filtering principle is central to PPGSQ-style systems. DFB-PPGSQ retains it. The new challenge is to maintain branch features under dynamic encrypted updates without exposing temporal correlations.

4.2. Structured Encryption

Structured encryption allows a client to outsource encrypted data structures while enabling controlled server-side operations [33]. A leakage function specifies what the server may learn during setup, update, and querying. Security is defined by simulation; whatever the real server observes should be simulatable from the allowed leakage.

DFB-PPGSQ treats the encrypted branch index as a structured encryption object. Branch identifiers are produced by pseudo-random functions, counters are masked or encrypted, tree nodes are re-randomized across epochs, and update tokens are generated from evolving keys.

The construction separates two token namespaces. In epoch e , the query-matchable feature token of branch b is

$$\theta_b^e = F_{K_F^e}(\text{canon}(b)), \quad (7)$$

where $\text{canon}(b)$ is a canonical encoding of the branch. This token is deterministic only inside one epoch and is the object that query tokens can match. For storage, the client additionally derives a per-record occurrence handle

$$\eta_{b,i,c}^e = F_{K_O^e}(\theta_b^e \parallel h_i^e \parallel c \parallel r_i), \quad h_i^e = F_{K_D^e}(\text{id}_i \parallel e), \quad (8)$$

where r_i is a graph-local nonce and c is an occurrence counter. The occurrence handle hides repeated occurrences and record membership; it is never used as the query key. This separation keeps feature matching well-defined while preventing the graph-local nonce from breaking query–token equality. The protected count stored at bucket $j = H_{K_H^e}(b) \bmod m$ is

$$\tilde{C}_G^e[j] = C_G^e[j] + F_{K_C^e}(\text{id}_G \parallel j) \pmod{p}. \quad (9)$$

Only query-authorized masks can be canceled or compared. Thus, the cloud obtains enough information to run the filtering predicate but does not receive plaintext branches or stable cross-epoch identifiers.

4.3. Forward and Backward Privacy

Forward privacy prevents a server from linking future inserted data to previous search tokens. A common method is to use key evolution or one-time update labels. In DFB-PPGSQ, the update key for epoch e is derived through a one-way key chain

$$K_U^{e,u} = F_{K_U^e}(\text{upd} \parallel u), \quad K_U^{e+1} = F_{K_R}(K_U^e \parallel e), \quad (10)$$

where u is a monotone update counter and K_R is the re-randomization master key. The server cannot derive the label of a future update from a past search token.

Backward privacy controls leakage from deleted data. If a document, graph, or branch feature is deleted, later searches should not reveal it. Strong backward privacy can be expensive; therefore, DFB-PPGSQ offers an epochal design: within the current epoch, a deletion may leak a tombstone handle and active-state check, while stale handles are removed from the active searchable structure after re-randomization. Formally, for a deleted graph set $\mathcal{D}_e$, query output is filtered by

$$R_e(Q, \tau) = \{G_i : \text{LBED}(G_i, Q) \leq \tau \wedge h_i^e \notin \mathcal{D}_e\}. \quad (11)$$

At re-randomization, only $G_i \notin \mathcal{D}_e$ is migrated to I_{e+1} .

4.4. Periodic Re-Randomization

Periodic re-randomization refreshes encrypted identifiers and index nodes at epoch boundaries. This breaks long-term equality links and provides a natural moment for compaction at which deleted entries can be removed, update buffers can be merged, and counters can be regenerated. In the stronger refresh mode used by DFB-PPGSQ, the client prepares the next-epoch active summaries and uploads them in a fresh random order:

$$\Omega_e = \pi_e \left(\left\{ \left(h_i^{e+1}, \{\theta_b^{e+1}, \eta_{b,i,c}^{e+1}\}_{b \in B(G_i)}, \tilde{C}_{G_i}^{e+1} \right) : h_i^e \in \mathcal{M}_e \right\} \right), \quad (12)$$

where π_e is a random permutation over active records. The cloud learns migration-size buckets, refreshed tree shape, and bucketed record sizes but does not receive a pairwise map from h_i^e to h_i^{e+1} or from θ_b^e to θ_b^{e+1} beyond the leakage function.

The epoch length controls a tradeoff in which a short epoch reduces temporal leakage but increases maintenance cost. A long epoch reduces maintenance cost but permits more accumulated leakage. Section 7 includes this tradeoff as an experimental variable.

5. The Proposed DFB-PPGSQ Scheme

5.1. Overview

DFB-PPGSQ follows the filtering-and-refinement philosophy. The client extracts branch features from data graphs and query graphs, the cloud uses an encrypted branch-tree index to filter candidates, and the client decrypts returned candidates; when exactness is required, the client refines the final graph edit distance locally.

At epoch e , the outsourced state is

$$\text{State}_e = (\text{EDB}_e, I_e, U_e, D_e, \text{meta}_e), \quad (13)$$

where EDB_e stores encrypted graph records, I_e is the compact encrypted branch-tree index, U_e is the recent update buffer, D_e is the tombstone set, and meta_e contains non-secret size and epoch metadata allowed by the leakage function. The active encrypted graph handles satisfy the invariant

$$\text{Act}_e = (\text{Leaves}(I_e) \cup \text{Handles}(U_e)) \setminus D_e. \quad (14)$$

The search procedure returns only handles in Act_e .

The state invariant is maintained as follows: for every active logical graph ID, exactly one current encrypted record is addressable either through a compact index leaf or through the update buffer; every deletion handle in D_e suppresses the corresponding active handle until refresh; every positive update label is generated with a non-reused counter; and every ReRand operation migrates only handles in Act_e into the next epoch and then clears U_e and D_e . Thus, arbitrary insertion, deletion, label update, search, and refresh sequences preserve the relation among EDB_e , I_e , U_e , and D_e so long as the trusted client maintains the logical ID map and update counters.

The construction evaluated in this paper is the lightweight configuration based on PRF tokens, masked counters, update buffers, tombstones, and shuffle refresh. Stronger tools such as homomorphic counters, ORAM traversal, volume padding, or oblivious compaction are compatible design directions but are not constructed or evaluated here, and are not claimed as part of the implemented scheme. As shown in Figure 2, the four dynamic mechanisms are as follows::

1. Epoch-local feature tokens and per-record occurrence handles for compact active data in I_e .
2. One-time update labels for recent insertions and label updates in U_e .
3. Epoch-specific deletion handles in D_e .
4. Re-randomization material Ω_e for epoch migration.

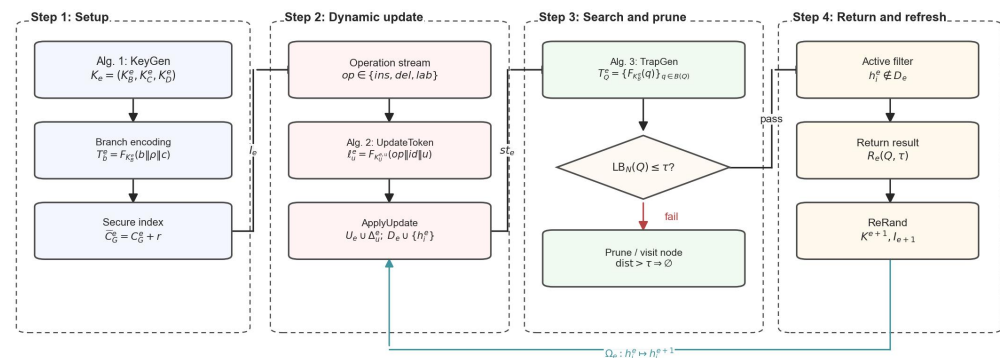


Figure 2. Algorithmic workflow of DFB-PPGSQ. The figure summarizes setup, dynamic update processing, encrypted graph similarity search and pruning, result return, and epoch refresh.

5.2. Branch Extraction, Token Generation, and Setup

For a graph $G = (V, E, L_V, L_E)$, the branch centered at $v \in V$ is represented as

$$b_v = (L_V(v), \{(L_E(v, u), L_V(u)) : (v, u) \in E\}), \quad (15)$$

where $\{\{\cdot\}\}$ denotes a multiset. The graph branch multiset is

$$B(G) = \{\{b_v : v \in V\}\}. \quad (16)$$

For each branch $b \in B(G_i)$, the client generates an epoch-specific feature token and a per-record occurrence handle:

$$\theta_b^e = F_{K_F^e}(\text{canon}(b)), \quad \eta_{b,i,c_b}^e = F_{K_O^e}(\theta_b^e \parallel h_i^e \parallel c_b \parallel r_i), \quad (17)$$

where F is a secure pseudo-random function, K_F^e is the feature token key, K_O^e is the occurrence-handle key, h_i^e is the active graph handle, r_i is a graph-local nonce, and c_b is a local occurrence counter. Queries match θ_b^e values. The nonce appears only in η_{b,i,c_b}^e , so it randomizes stored occurrences without preventing query matching.

The client also prepares a protected branch count vector. A branch position is derived by

$$j_e(b) = H_{K_H^e}(b) \bmod m, \quad \mathbf{x}_G^e[j] = \sum_{b \in B(G)} \mathbf{1}[j_e(b) = j], \quad (18)$$

and the protected count is

$$\tilde{\mathbf{x}}_G^e[j] = \mathbf{x}_G^e[j] + M_{K_C^e}(\text{id}_G, j) \pmod{p}. \quad (19)$$

For a tree node v containing graph handles S_v , the aggregate plaintext count vector is

$$\mathbf{x}_v^e[j] = \max_{G_i \in S_v} \mathbf{x}_{G_i}^e[j], \quad (20)$$

and the stored value is masked in the same way. The maximum aggregate makes the node a safe filter: if a query cannot match the aggregate, it cannot match any child graph under the same lower-bound rule.

Algorithm 1 shows that setup is not merely an encryption step; it also initializes the epoch keys, graph nonces, masked branch counts, and protected node aggregates required for later pruning. The server sees protected handles and vector lengths, while the client keeps the semantic map from plaintext branches to epoch-local feature tokens.

Algorithm 1 Setup and Protected Branch–Tree Construction

Input: master key K , initial graph database $\mathcal{G}^0 = \{G_1, \dots, G_N\}$, vector length m , modulus p

Output: encrypted database EDB_0 , branch-tree index I_0 , client state st_0

- 1: Derive epoch keys $(K_F^0, K_O^0, K_H^0, K_C^0, K_U^0, K_D^0) \leftarrow F_K(\text{setup} \parallel 0)$
 - 2: Initialize $\text{EDB}_0 \leftarrow \emptyset, S_0 \leftarrow \emptyset, U_0 \leftarrow \emptyset, D_0 \leftarrow \emptyset$
 - 3: **for** each graph $G_i \in \mathcal{G}^0$ **do**
 - 4: sample graph nonce $r_i \leftarrow \{0, 1\}^\lambda$ and handle $h_i^0 \leftarrow F_{K_D^0}(\text{id}_i \parallel 0)$
 - 5: compute branch multiset $B(G_i)$ and count vector $\mathbf{x}_{G_i}^0$
 - 6: **for** each occurrence $(b, c_b) \in B(G_i)$ **do**
 - 7: generate feature token $\theta_b^0 \leftarrow F_{K_F^0}(\text{canon}(b))$
 - 8: generate occurrence handle $\eta_{b,i,c_b}^0 \leftarrow F_{K_O^0}(\theta_b^0 \parallel h_i^0 \parallel c_b \parallel r_i)$
 - 9: **end for**
 - 10: mask counts $\tilde{\mathbf{x}}_{G_i}^0[j] \leftarrow \mathbf{x}_{G_i}^0[j] + M_{K_C^0}(\text{id}_i, j) \bmod p$
 - 11: append $\text{Enc}_K(G_i)$ to EDB_0 and $(h_i^0, \{(\theta_b^0, \eta_{b,i,c_b}^0)\}, \tilde{\mathbf{x}}_{G_i}^0)$ to S_0
 - 12: **end for**
 - 13: build $I_0 \leftarrow \text{BuildTree}(S_0)$ with node aggregates $\tilde{\mathbf{x}}_v^0$
 - 14: set $\text{st}_0 \leftarrow (K, 0, \{r_i\}_{i=1}^N, \text{ctr} = 0)$
 - 15: **return** $(\text{EDB}_0, I_0, \text{st}_0)$
-

5.3. Dynamic Branch–Tree Index

The branch–tree index groups graphs by encrypted branch summary ranges. Each internal node stores a protected aggregate summary of its children, and each leaf stores encrypted graph handles and compact branch count summaries. BUILDTree is a client-side protected tree construction routine. Given active summaries, the client first applies a keyed pseudo-random ordering or the refresh permutation, packs leaves with a fixed parity parameter a , computes each internal node aggregate as the coordinate-wise maximum of its child plaintext count vectors before masking, and uploads only protected node summaries, child pointers, and size metadata allowed by the leakage function. The cloud stores and traverses this protected tree but does not need plaintext labels to construct node aggregates.

For a query branch count vector $\mathbf{x}_Q^e$ and a node aggregate $\mathbf{x}_v^e$, define the residual branch deficit and the residual vertex count deficit as

$$\Delta_B(Q, v) = \sum_{j=1}^m \max\{0, \mathbf{x}_Q^e[j] - \mathbf{x}_v^e[j]\}, \quad \Delta_V(Q, v) = \max\{0, |V_Q| - V_v^{\max}\}. \quad (21)$$

The node lower bound follows the same max-based form as the graph-level lower bound:

$$\text{LBED}_e(Q, v) = \max\left\{\left\lceil \frac{\Delta_B(Q, v)}{2} \right\rceil, \left\lceil \frac{\Delta_V(Q, v)}{2} \right\rceil\right\}. \quad (22)$$

If encrypted vertex label aggregates are enabled, $\Delta_V(Q, v)$ can instead use the same positive-deficit sum over vertex label counters; the prototype uses the conservative vertex-count version. A node is safely pruned when

$$\text{LBED}_e(Q, v) > \tau. \quad (23)$$

For any child graph G_i under v , the aggregate construction gives $\mathbf{x}_{G_i}^e[j] \leq \mathbf{x}_v^e[j]$, so

$$\text{LBED}_e(Q, v) \leq \text{LBED}_e(Q, G_i). \quad (24)$$

Thus, the pruning rule cannot discard a graph for which the exact branch lower bound is at most τ .

5.4. Forward-Private Insertion

When inserting a graph G_{new} , the client extracts $B(G_{\text{new}})$, derives feature tokens θ_b^e , generates a fresh graph nonce r_{new} , and creates one-time update labels with a monotonically increasing counter u :

$$U_{b,u}^e = F_{K_U^e}(\theta_b^e \parallel u). \quad (25)$$

The counter is never re-used. The update record is

$$\Delta_{\text{ins}} = (\text{ins}, h_{\text{new}}^e, \text{Enc}_K(G_{\text{new}}), \{U_{b,u}^e\}_{b \in B(G_{\text{new}})}, \tilde{\mathbf{x}}_{\text{new}}^e). \quad (26)$$

The cloud appends this record to U_e . Since $U_{b,u}^e$ is derived from an update key and a future counter, an old query token cannot be used to test whether a future inserted graph contains a branch matching that old query. A later query authorizes only the current update counter interval needed to test buffered records.

5.5. Backward-Private Deletion

When deleting graph G_i , the client issues an epochal deletion handle

$$h_i^e = F_{K_D^e}(\text{id}_i \parallel e). \quad (27)$$

The deletion token is

$$\Delta_{\text{del}} = (\text{del}, h_i^e), \quad D_e \leftarrow D_e \cup \{h_i^e\}. \quad (28)$$

During the same epoch, deletion leaks only the tombstone handle and the fact that a candidate may be suppressed by D_e . At re-randomization, deleted records are not migrated into I_{e+1} ; after that point, they no longer participate in future search traces.

5.6. Label Update

A label update modifies vertex labels or edge labels in a graph. It changes only those branches for which the center or one-hop neighborhood intersects the edited element. If δ is an edit operation on G_i , define the affected branch set as

$$\text{Aff}(G_i, \delta) = B(G_i) \triangle B(G_i \oplus \delta), \quad (29)$$

where $\triangle$ is multiset symmetric difference. The update decomposes into removals of old branches and insertions of new branches:

$$\text{Aff}^- = \text{Aff} \cap B(G_i), \quad \text{Aff}^+ = \text{Aff} \cap B(G_i \oplus \delta). \quad (30)$$

For each $b \in \text{Aff}^+$, the client creates a fresh one-time update label $U_{b,u}^e$, while for each $b \in \text{Aff}^-$ the client creates a removal tag tied to the active handle. The protected delta vector is

$$\Delta \tilde{\mathbf{x}}_i^e[j] = (\mathbf{x}_{G_i \oplus \delta}^e[j] - \mathbf{x}_{G_i}^e[j]) + M_{K_e}(\text{id}_i, j, u) \pmod{p}. \quad (31)$$

Algorithm 2 is the unified dynamic update interface. Insertions and positive label deltas use fresh update labels, while deletions use epochal tombstones; label changes are represented as local branch deltas. This keeps ordinary updates small and postpones tree rebuilding to epoch maintenance.

Algorithm 2 Dynamic Graph Update Processing

Input: operation op , epoch e , client state st_e , outsourced state $(\text{EDB}_e, I_e, U_e, D_e)$

Output: updated outsourced state and client state

```

1: if  $\text{op} = \text{Insert}(G)$  then
2:   sample graph nonce  $r_G$  and compute  $B(G)$ ,  $\tilde{\mathbf{x}}_G^e$ , and active handle  $h_G^e$ 
3:   generate feature tokens  $\{\theta_b^e\}_{b \in B(G)}$  and one-time update labels  $\{U_{b,u}^e\}_{b \in B(G)}$  with fresh
   counter values
4:    $\Delta \leftarrow (\text{ins}, h_G^e, \text{Enc}_K(G), \{U_{b,u}^e\}, \tilde{\mathbf{x}}_G^e)$ 
5:   append  $\Delta$  to  $U_e$  and  $\text{Enc}_K(G)$  to  $\text{EDB}_e$ 
6: else if  $\text{op} = \text{Delete}(\text{id})$  then
7:   compute  $h_{\text{id}}^e \leftarrow F_{K_D}(\text{id} \parallel e)$ 
8:   append  $h_{\text{id}}^e$  to  $D_e$ 
9: else if  $\text{op} = \text{UpdateLabel}(\text{id}, \delta)$  then
10:  compute  $\text{Aff}^-$  and  $\text{Aff}^+$  for  $G_{\text{id}} \oplus \delta$ 
11:  generate removal tags for  $\text{Aff}^-$  and one-time update labels for  $\text{Aff}^+$ 
12:  compute masked delta vector  $\Delta \tilde{\mathbf{x}}_{\text{id}}^e$ 
13:  append  $(\text{upd}, h_{\text{id}}^e, \text{Aff}^-, \text{Aff}^+, \Delta \tilde{\mathbf{x}}_{\text{id}}^e)$  to  $U_e$ 
14: end if
15: update local counters in  $\text{st}_e$ 
16: return  $(\text{EDB}_e, I_e, U_e, D_e, \text{st}_e)$ 

```

5.7. Query Processing

Given query graph Q and threshold τ , the client extracts $B(Q)$ and generates a query token

$$\mathcal{T}_Q^e = \left(\{\theta_q^e\}_{q \in B(Q)}, \{\omega_{q,u}^e\}_{q \in B(Q), u \in \mathcal{U}_e}, \tilde{\mathbf{x}}_Q^e, \tau, \text{auth}_e \right), \quad (32)$$

where $\omega_{q,u}^e = F_{K_U}^e(\theta_q^e \parallel u)$ authorizes comparisons with buffered update labels for the current update counter interval $\mathcal{U}_e$. Thus, θ_q^e searches the compact branch tree I_e , while $\omega_{q,u}^e$ searches one-time labels in U_e . The cloud computes two candidate sets:

$$\text{Cand}_I = \{h_i^e \in I_e : \text{LBED}_e(Q, G_i) \leq \tau\}, \quad (33)$$

$$\text{Cand}_U = \{h_i^e \in U_e : \text{LBED}_e(Q, G_i) \leq \tau\}. \quad (34)$$

The active candidate set is

$$\text{Cand}_e(Q, \tau) = (\text{Cand}_I \cup \text{Cand}_U) \setminus D_e. \quad (35)$$

The cloud never sees Q or plaintext graph labels. It observes the declared traversal pattern, update buffer checks, and result size. If the application requires less traversal leakage, the same query interface can be combined with padded traversal or oblivious access at additional cost.

The node-level primitive operations of nodes in Algorithm 3 are as shown in Algorithm 4. It no longer introduces a separate $0.5t$ vertex-token term; instead, it computes the same max-based residual form used in Section 4. The early termination rule is safe because Δ_B is a sum of non-negative coordinate deficits; once $\lceil \Delta_B/2 \rceil > \tau$, later coordinates cannot reduce the lower bound below the pruning threshold. This makes the relationship between the pruning algorithm and $\text{LBED}(G, Q) = \max\{\lceil \Delta_B/2 \rceil, \lceil \Delta_V/2 \rceil\}$ explicit.

Algorithm 3 Encrypted Graph Similarity Search

Input: query graph Q , threshold τ , epoch e , state (I_e, U_e, D_e)
Output: encrypted candidate set $R_e(Q, \tau)$

- 1: Client computes $B(Q)$, $\tilde{x}_Q^e$, and query token $\mathcal{T}_Q^e$
- 2: Server initializes $\text{Cand}_I \leftarrow \emptyset$ and queue $\mathcal{Q} \leftarrow \{\text{root}(I_e)\}$
- 3: **while** $\mathcal{Q} \neq \emptyset$ **do**
- 4: pop node v from $\mathcal{Q}$
- 5: compute $\text{dist}_v \leftarrow \text{LBED}_e(Q, v)$ using authorized protected summaries
- 6: **if** $\text{dist}_v \leq \tau$ and v is internal **then**
- 7: push all children of v into $\mathcal{Q}$
- 8: **else if** $\text{dist}_v \leq \tau$ and v is a leaf **then**
- 9: test graph-level summaries in v and append passing handles to Cand_I
- 10: **end if**
- 11: **end while**
- 12: compute Cand_U by testing buffered insertions and label-update deltas in U_e
- 13: filter tombstones: $\text{Cand}_e \leftarrow (\text{Cand}_I \cup \text{Cand}_U) \setminus D_e$
- 14: return encrypted graph records $R_e(Q, \tau) \leftarrow \{\text{EDB}_e[h] : h \in \text{Cand}_e\}$

Algorithm 4 Node Lower-Bound Test

Input: query count vector $\mathbf{x}_Q^e$, node aggregate $\mathbf{x}_v^e$, query vertex count $|V_Q|$, node maximum vertex count $V_v^{\max}$, threshold τ
Output: lower-bound score dist_v for node pruning

- 1: Initialize $\Delta_B \leftarrow 0$
- 2: **for** $j = 1$ to m **do**
- 3: $\delta_j \leftarrow \max\{0, \mathbf{x}_Q^e[j] - \mathbf{x}_v^e[j]\}$
- 4: $\Delta_B \leftarrow \Delta_B + \delta_j$
- 5: **if** $\lceil \Delta_B/2 \rceil > \tau$ **then**
- 6: **return** $\lceil \Delta_B/2 \rceil$
- 7: **end if**
- 8: **end for**
- 9: $\Delta_V \leftarrow \max\{0, |V_Q| - V_v^{\max}\}$
- 10: $\text{dist}_v \leftarrow \max\{\lceil \Delta_B/2 \rceil, \lceil \Delta_V/2 \rceil\}$
- 11: **return** dist_v

5.8. Periodic Re-Randomization

At the end of epoch e , the client starts ReRand. It derives new epoch keys

$$(K_F^{e+1}, K_O^{e+1}, K_H^{e+1}, K_C^{e+1}, K_U^{e+1}, K_D^{e+1}) = F_K(\text{epoch} \parallel e + 1). \quad (36)$$

Let the active migration set be

$$\mathcal{M}_e = \{h_i^e : h_i^e \in \text{Leaves}(I_e) \cup \text{Handles}(U_e), h_i^e \notin D_e\}. \quad (37)$$

For every active graph, as shown in Algorithm 5, the client computes new feature tokens, occurrence handles, active handles, and masked counters under epoch $e + 1$ keys. The refresh package is

$$\Omega_e = \pi_e \left(\{ (h_i^{e+1}, \{ (\theta_b^{e+1}, \eta_{b,i,c_b}^{e+1}) \}_{b \in B(G_i)}, \tilde{x}_i^{e+1}) : h_i^e \in \mathcal{M}_e \} \right), \quad (38)$$

where π_e is sampled by the client. The cloud installs this package as a replacement state, meaning that it observes the number of active records and bucketed sizes but not a pairwise old-to-new handle map.

Algorithm 5 Epoch Re-Randomization and State Refresh

Input: epoch e , index I_e , update buffer U_e , tombstones D_e , client state st_e

Output: refreshed index I_{e+1} and cleared buffers

- 1: Client derives $(K_F^{e+1}, K_O^{e+1}, K_H^{e+1}, K_C^{e+1}, K_U^{e+1}, K_D^{e+1})$
 - 2: Client determines active logical ids represented by $\mathcal{M}_e$ and excludes D_e
 - 3: Initialize local refresh list $\mathcal{S}_{e+1} \leftarrow \emptyset$
 - 4: **for** each active graph id id_i in random order π_e **do**
 - 5: compute $h_i^{e+1} \leftarrow F_{K_D^{e+1}}(\text{id}_i \parallel e + 1)$
 - 6: refresh feature tokens $\theta_b^{e+1} \leftarrow F_{K_F^{e+1}}(\text{canon}(b))$
 - 7: refresh occurrence handles $\eta_{b,i,c_b}^{e+1} \leftarrow F_{K_O^{e+1}}(\theta_b^{e+1} \parallel h_i^{e+1} \parallel c_b \parallel r_i)$
 - 8: recompute or remask $\tilde{x}_i^{e+1}$
 - 9: append $(h_i^{e+1}, \{ (\theta_b^{e+1}, \eta_{b,i,c_b}^{e+1}) \}, \tilde{x}_i^{e+1})$ to $\mathcal{S}_{e+1}$
 - 10: **end for**
 - 11: Client computes protected node aggregates and sends a shuffled tree package $\Omega_e = \text{BuildTree}(\mathcal{S}_{e+1})$
 - 12: Server installs $I_{e+1} \leftarrow \Omega_e$ without receiving an old-to-new handle map
 - 13: set $U_{e+1} \leftarrow \emptyset$, $D_{e+1} \leftarrow \emptyset$, and securely discard stale epoch material
 - 14: update client state st_{e+1}
 - 15: **return** $(I_{e+1}, U_{e+1}, D_{e+1}, \text{st}_{e+1})$
-

Re-randomization can be full or incremental. Full re-randomization rebuilds the whole active index from a shuffled client refresh package. Incremental re-randomization uploads a fixed budget of refreshed records per time window, with a fresh permutation inside each batch. In both cases, deleted records are excluded from $\mathcal{M}_e$, which is the mechanism that gives the construction its epochal backward-privacy effect. The incremental mode leaks batch sizes and batch timing; the full mode leaks the epoch refresh size but avoids pairwise migration linkage.

6. Security Analysis

6.1. Threat Model and Leakage Profile

We model the cloud as an honest-but-curious probabilistic polynomial-time adversary. It follows Setup, ApplyUpdate, Query, and ReRand but tries to infer graph contents, query branches, update contents, and temporal relations among operations. The client is trusted and keeps secret the master key K , epoch keys, graph nonces, logical ID map, and local counters. The encrypted graph database is protected by an IND-CPA secure symmetric encryption scheme, and all token-generation functions are modeled as pseudo-random functions.

The security goal is leakage-based structured encryption security. Let

$$\mathcal{O} = (\text{setup}, \text{op}_1, \dots, \text{op}_T) \quad (39)$$

be an adaptive operation sequence selected by the adversary, where each op_t is an insertion, deletion, label update, query, or epoch refresh. The real server view is

$$\text{View}_{\mathcal{A}}^{\Pi}(\lambda, \mathcal{O}) = \left(\text{EDB}_e, I_e, U_e, D_e, \{\Delta_t\}_{t \leq T}, \{\mathcal{T}_{Q_t}^{e_t}\}_{t \leq T}, \{\text{path}_{e_t}(Q_t), R_{e_t}(Q_t, \tau_t)\}_{t \leq T} \right). \quad (40)$$

The simulator is allowed to see only a leakage function

$$\mathcal{L} = (\mathcal{L}_{\text{setup}}, \mathcal{L}_{\text{upd}}, \mathcal{L}_{\text{qry}}, \mathcal{L}_{\text{rr}}). \quad (41)$$

Definition 1 (Leakage functions). For an initial database $\mathcal{G}^0$, update op_t , query (Q_t, τ_t) , and epoch refresh from e to $e + 1$, DFB-PPGSQ exposes the following leakage:

$$\mathcal{L}_{\text{setup}}(\mathcal{G}^0) = \left(N, \{|G_i|\}_{i=1}^N, \{|B(G_i)|\}_{i=1}^N, m, p, \text{shape}(I_0), \{|S_v|\}_{v \in I_0} \right), \quad (42)$$

$$\mathcal{L}_{\text{upd}}(\text{op}_t, e) = (t, e, \text{type}(\text{op}_t), \text{bucket}(|\Delta_t|), \text{bucket}(|\text{Aff}(\text{op}_t)|), \mathbf{1}[\text{op}_t = \text{del}]), \quad (43)$$

$$\mathcal{L}_{\text{qry}}(Q_t, \tau_t, e) = (t, e, \tau_t, \text{path}_e(Q_t), \text{updscan}_e(Q_t), |\text{Cand}_e(Q_t, \tau_t)|, |R_e(Q_t, \tau_t)|), \quad (44)$$

and

$$\mathcal{L}_{\text{rr}}(e) = (e, |\mathcal{M}_e|, |D_e|, |U_e|, \text{shape}(I_{e+1}), \text{bucketmultiset}(\{|\text{EDB}[h]| : h \in \mathcal{M}_e\}), \text{bucket}(T_{\text{rr}})). \quad (45)$$

Here, N is the number of outsourced graphs, m and p are public counter parameters, S_v is the handle set under node v , Aff is the affected branch-delta set for a label update, $\text{path}_e(Q_t)$ is the tree traversal trace, and $\text{updscan}_e(Q_t)$ is the set of update buffer records tested by the query token.

The leakage definition is intentionally explicit. DFB-PPGSQ does not claim oblivious access, result-size hiding, or fully hidden update timing; instead, it prevents the cloud from learning plaintext graph labels, plaintext branch equality across protected epochs, and token-level relations beyond the declared size, timing, traversal, and migration metadata. This leakage-based theorem proves that the real view reveals no more than $\mathcal{L}$ under the stated cryptographic assumptions; it does not prove that every observable in $\mathcal{L}$ is harmless against arbitrary auxiliary information, statistical, or machine learning leakage abuse attacks, which must be evaluated separately for a concrete deployment profile.

6.2. Real–Ideal Security Definition

Definition 2 (Adaptive leakage security). Let $\text{Real}_{\mathcal{A}}^{\Pi}(\lambda)$ be the experiment in which adversary $\mathcal{A}$ adaptively interacts with the real DFB-PPGSQ protocol and receives the real server view. Let $\text{Ideal}_{\mathcal{A}, \mathcal{S}}^{\mathcal{L}}(\lambda)$ be the experiment in which $\mathcal{A}$ interacts with a simulator $\mathcal{S}$ that receives only $\mathcal{L}(\mathcal{O})$ for the same operation sequence. DFB-PPGSQ is adaptively secure with respect to $\mathcal{L}$ if, for every PPT adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that

$$\text{Adv}_{\Pi, \mathcal{L}}^{\text{se}}(\mathcal{A}, \mathcal{S}, \lambda) = |\Pr[\mathcal{A}(\text{View}_{\text{real}}) = 1] - \Pr[\mathcal{A}(\text{View}_{\text{ideal}}) = 1]| \leq \text{negl}(\lambda). \quad (46)$$

The simulator uses $\mathcal{L}_{\text{setup}}$ to sample ciphertext strings, protected handles, vector lengths, and a tree with the leaked shape. It uses $\mathcal{L}_{\text{upd}}$ to sample update records with the correct operation class and size bucket. It uses $\mathcal{L}_{\text{qry}}$ to reproduce the same pattern of visited nodes, update buffer scan size, candidate size, and returned ciphertext count. It uses $\mathcal{L}_{\text{rr}}$ to sample shuffled epoch refresh messages and the new tree shape without using a pairwise old-to-new handle or branch map.

6.3. Token and Ciphertext Indistinguishability

Lemma 1 (Pseudo-random token simulation). *Let F be a secure pseudo-random function. For any polynomial number of inputs*

$$x_r \in \{\text{canon}(b), \theta_b^e \parallel h_i^e \parallel c_b \parallel r_i, \theta_b^e \parallel u, \text{id}_i \parallel e, \text{epoch} \parallel e\}, \quad (47)$$

the token family $\{F_K(x_r)\}_r$ is computationally indistinguishable from random strings of the same length, preserving only equality that is intentionally leaked by same-epoch feature tokens or authorized update probes. More precisely, for every PPT distinguisher $\mathcal{D}$,

$$|\Pr[\mathcal{D}(\{F_K(x_r)\}_r) = 1] - \Pr[\mathcal{D}(\{U_r\}_r) = 1]| \leq \text{Adv}_F^{\text{prf}}(\mathcal{B}, \lambda), \quad (48)$$

where $\{U_r\}_r$ are independent uniform strings and $\mathcal{B}$ is a standard PRF adversary constructed from $\mathcal{D}$.

Proof. The proof is the standard PRF hybrid. Replace the oracle used to answer each token request with a truly random function. Feature token equality for the same branch inside one epoch is intentionally retained because it is needed for matching; occurrence handles, update labels, deletion handles, and epoch identifiers use non-reused namespaces. If $\mathcal{D}$ distinguishes real tokens from random labels under these retained equality constraints, then $\mathcal{D}$ directly yields a PRF distinguisher $\mathcal{B}$ with the same distinguishing gap. Thus, the difference is bounded by $\text{Adv}_F^{\text{prf}}(\mathcal{B}, \lambda)$. $\square$

Lemma 2 (Encrypted record simulation). *If the graph encryption scheme is IND-CPA secure, then encrypted graph records in EDB_e can be simulated from record-size leakage. For any PPT adversary $\mathcal{A}$,*

$$\left| \Pr[\mathcal{A}(\{\text{Enc}_K(G_i)\}_i) = 1] - \Pr[\mathcal{A}(\{\text{Enc}_K(0^{|G_i|})\}_i) = 1] \right| \leq \text{Adv}_{\text{Enc}}^{\text{ind-cpa}}(\mathcal{B}, \lambda). \quad (49)$$

Proof. The simulator encrypts zero strings of the same size buckets as the real graph encodings. Any adversary that distinguishes these simulated ciphertexts from real graph ciphertexts breaks IND-CPA security through the usual many-message hybrid. $\square$

6.4. Main Security Theorem

Theorem 1 (Structured encryption security of DFB-PPGSQ). *Assume that F is a secure pseudo-random function, M_{K_C} is a pseudo-random mask family over $\mathbb{Z}_p$, graph encryption is IND-CPA secure, and nonces/counters are non-reused within their namespaces. Then, DFB-PPGSQ is adaptively secure with respect to the leakage function $\mathcal{L}$ in Definition 1:*

$$\text{Adv}_{\Pi, \mathcal{L}}^{\text{se}}(\mathcal{A}, S, \lambda) \leq \text{Adv}_{\text{Enc}}^{\text{ind-cpa}} + q_F \text{Adv}_F^{\text{prf}} + q_M \text{Adv}_M^{\text{prf}} + \text{negl}(\lambda), \quad (50)$$

where q_F and q_M are polynomially bounded numbers of token and mask queries.

Proof. We use a sequence of hybrids. In H_0 , the adversary receives the real server view. In H_1 , every encrypted graph record $\text{Enc}_K(G_i)$ is replaced by an encryption of a zero string with the same leaked size. By Lemma 2, H_0 and H_1 are indistinguishable under IND-CPA security.

In H_2 , all feature tokens, occurrence handles, update labels, deletion handles, query labels, and epoch keys are replaced by random strings that preserve equality only when equality is allowed by $\mathcal{L}$ in the same epoch or by a query-authorized update probe. Lemma 1 bounds the distinguishing advantage between H_1 and H_2 by the PRF advantage.

In H_3 , protected count values

$$\tilde{\mathbf{x}}_G^e[j] = \mathbf{x}_G^e[j] + M_{K_C^e}(\text{id}_G, j) \pmod{p} \quad (51)$$

and node aggregates are replaced by random elements of $\mathbb{Z}_p$ that are consistent with the leaked vector length, modulus, and comparison outcomes. Since $M_{K_C^e}$ is pseudo-random, the adversary sees only masked values and the declared pruning outcomes. Thus, H_2 and H_3 are indistinguishable up to $\text{Adv}_M^{\text{prf}}$.

In H_4 , the simulator generates the entire server view using $\mathcal{L}$. It samples ciphertexts with leaked sizes, random handles and tokens with leaked equality patterns, a tree with shape $\text{shape}(I_e)$, query traversal paths $\text{path}_e(Q)$, update scans $\text{updscan}_e(Q)$, result counts, and shuffled epoch refresh sizes. By construction, H_3 and H_4 have the same distribution; therefore, the real and ideal experiments are computationally indistinguishable except for the stated cryptographic advantages. $\square$

6.5. Forward Privacy

Theorem 2 (Forward privacy). *Let t be a time before an insertion or positive label-update delta at time $t' > t$. Under the PRF assumption and non-reused update counters, any token view observed before t' is independent of the plaintext branch set inserted at t' except for $\mathcal{L}_{\text{upd}}$. Formally, for every branch predicate P and PPT adversary $\mathcal{A}$,*

$$\left| \Pr[\mathcal{A}(\text{View}_{<t'}, \Delta_{t'}) = P(B_{\text{new}})] - \Pr[\mathcal{A}(\text{View}_{<t'}, \hat{\Delta}_{t'}) = P(B_{\text{new}})] \right| \leq \text{Adv}_F^{\text{prf}} + \text{negl}(\lambda), \quad (52)$$

where $\hat{\Delta}_{t'}$ is a simulated update record with the same leakage but random update labels.

Proof. For each inserted branch $b \in B_{\text{new}}$, the server receives

$$U_{b,u}^e = F_{K_U^e}(\theta_b^e \parallel u), \quad (53)$$

where u has not appeared before. Tokens issued before t' did not contain the future counter probe $\omega_{b,u}^e = F_{K_U^e}(\theta_b^e \parallel u)$. Hence, there is no efficiently testable relation between an old search token and $U_{b,u}^e$ unless a future query intentionally authorizes that update counter interval. Replacing $U_{b,u}^e$ with a random string changes the adversary's view only by the PRF advantage. The remaining visible information is update time, type, epoch, and size bucket, which is exactly $\mathcal{L}_{\text{upd}}$. $\square$

6.6. Backward Privacy

Theorem 3 (Epochal backward privacy). *Let the graph handle h_i^e be deleted during epoch e , and let the next refresh produce epoch $e + 1$. If deleted handles are excluded from $\mathcal{M}_e$, then future search views in epochs $e' > e$ are independent of G_i beyond deletion and migration leakage:*

$$h_i^e \in D_e \implies h_i^{e+1} \notin \text{Leaves}(I_{e+1}) \cup \text{Handles}(U_{e+1}), \quad (54)$$

and for every future query (Q, τ) ,

$$\Pr[h_i^{e+1} \in \text{Cand}_{e+1}(Q, \tau)] = 0. \quad (55)$$

Proof. During epoch e , the deletion operation adds

$$\Delta_{\text{del}} = (\text{del}, h_i^e), \quad D_e \leftarrow D_e \cup \{h_i^e\}. \quad (56)$$

Same-epoch queries may reveal that a candidate is suppressed by a tombstone; this is included in the deletion leakage. At ReRand, the migration set is

$$\mathcal{M}_e = (\text{Leaves}(I_e) \cup \text{Handles}(U_e)) \setminus D_e. \quad (57)$$

Thus, $h_i^e \notin \mathcal{M}_e$, and no new active handle h_i^{e+1} , occurrence handle η_{b,i,c_b}^{e+1} , or protected count vector $\tilde{x}_i^{e+1}$ is generated for the deleted graph. Future queries operate only on I_{e+1} and U_{e+1} . Therefore, the deleted graph cannot be returned or tested as an active record after refresh. Any attempt to link future query tokens to stale deleted-graph tokens reduces to distinguishing independently generated epoch- $e + 1$ PRF outputs from random strings plus the explicitly leaked refresh-size metadata. $\square$

6.7. Query and Result Privacy

Proposition 1 (Query-view simulation). *For a query (Q, τ) in epoch e , the server view*

$$\text{View}_{\text{qry}}(Q, \tau, e) = (\mathcal{T}_Q^e, \text{path}_e(Q), \text{updscan}_e(Q), \text{Cand}_e(Q, \tau), R_e(Q, \tau)) \quad (58)$$

is simulatable from $\mathcal{L}_{\text{qry}}(Q, \tau, e)$ and cryptographic sizes.

Proof. The simulator samples pseudo-random query labels for $\mathcal{T}_Q^e$, creates a traversal trace with the leaked path length and node identifiers, samples update buffer checks according to $\text{updscan}_e(Q)$, and returns the leaked number of ciphertext records. The plaintext branch labels in Q are used only inside PRF inputs and masked count vectors. By Lemmas 1 and 2, replacing these objects with simulated labels and ciphertexts is computationally indistinguishable. The non-hidden parts are exactly traversal pattern, update buffer checks, candidate count, and result size. $\square$

6.8. Re-Randomization Security

Proposition 2 (Cross-epoch unlinkability). *For any active branch occurrence (b, i, c) refreshed from epoch e to $e + 1$, the occurrence handle pair*

$$(\eta_{b,i,c}^e, \eta_{b,i,c}^{e+1}) = (F_{K_O^e}(\theta_b^e \parallel h_i^e \parallel c \parallel r_i), F_{K_O^{e+1}}(\theta_b^{e+1} \parallel h_i^{e+1} \parallel c \parallel r_i)) \quad (59)$$

is computationally unlinkable across epochs except through $\mathcal{L}_{rr}$ and workload metadata. The cloud does not receive the pair (h_i^e, h_i^{e+1}) during full shuffle refresh. For any PPT linking adversary $\mathcal{A}$,

$$\text{Adv}_{\text{link}}(\mathcal{A}) \leq 2\text{Adv}_F^{\text{prf}} + \text{Pr}[\text{MetaColl}] + \text{negl}(\lambda), \quad (60)$$

where MetaColl denotes a uniqueness event in leaked metadata such as record size buckets, refresh batch timing, or repeated result size patterns.

Proof. Epoch keys are derived as independent PRF outputs:

$$K_O^e = F_K(\text{epoch} \parallel e \parallel O), \quad K_O^{e+1} = F_K(\text{epoch} \parallel e + 1 \parallel O). \quad (61)$$

Replacing epoch- e occurrence handles with random labels and then replacing epoch- $(e + 1)$ occurrence handles with independent random labels changes the adversary's view only by the PRF advantage. Independent random labels do not encode branch or record equality across epochs. Because the refresh package is shuffled, a successful link must use non-token information such as bucketed record size, batch timing, or repeated query result metadata. These correlations are captured by $\text{Pr}[\text{MetaColl}]$ and the declared re-randomization leakage. $\square$

6.9. Comparison with Static PPGSQ-Style Indexing

In a static PPGSQ-style index, a branch occurrence keeps a stable encrypted identifier. Abstractly, if the same branch occurrence appears in two time windows, the server may observe

$$\eta_{b,i,c}^e = \eta_{b,i,c}^{e+1} \quad \text{or} \quad \text{path}_e(Q) \approx \text{path}_{e+1}(Q), \quad (62)$$

which enables longitudinal profiling. Inserted branches can be tested against old query identifiers if update labels reuse the same branch namespace, and deleted branches may remain observable through stale index nodes until a full rebuild is performed.

DFB-PPGSQ changes these equalities into epoch-local relations:

$$\Pr[\eta_{b,i,c}^e = \eta_{b,i,c}^{e+1}] \leq 2^{-\lambda}, \quad \text{Act}_{e+1} = (\text{Act}_e \cup \text{Ins}_e \cup \text{Upd}_e) \setminus D_e. \quad (63)$$

Thus, active records are re-tokenized, update records use one-time labels, deleted records are removed from the next searchable state, and the cloud does not receive a pairwise old-to-new refresh map. The price is visible update timing, buffer scans, bucketed refresh sizes, and epoch-maintenance leakage, while the gain is that temporal branch and handle equality are no longer stable server-side identifiers.

7. Experimental Evaluation

This section reports a reproducible prototype evaluation of DFB-PPGSQ. The goal is to test whether the proposed dynamic mechanisms preserve the main branch-filtering advantage of PPGSQ-style search while avoiding full index rebuild after ordinary updates. It implements protocol-relevant operations in Python: branch extraction, HMAC-SHA256 token generation, feature token matching, one-time update labels, branch count lower-bound filtering, feature-only dynamic SSE lookup, update buffering, tombstone filtering, and shuffle-based epoch re-randomization.

The evaluation is organized around three contribution-oriented questions: whether dynamic updates avoid immediate full-index rebuilds, whether the protected branch tree preserves the pruning advantage of static PPGSQ-style search, and whether epoch re-randomization reduces cross-epoch linkability while exposing the expected maintenance tradeoff.

7.1. Prototype Implementation and Workloads

The evaluation was executed on Windows 11 with Python 3.14.2 on an Intel64 Family 6 Model 189 processor. Every plotted point reports the mean over five independent random seeds, and the error bars show the standard deviation across these seeds. The generated workload models a dynamic sensor topology graph database. Each default mesh graph contains 8–22 vertices. Vertices receive assigned sensor or device labels such as temperature, pressure, gateway, camera, and alarm. Edges encode communication, control, backup, or event relations. A ring backbone keeps each graph connected, and additional sparse edges emulate redundant sensor links or event dependencies. To strengthen the sensor-oriented evaluation, we also add industrial CPS and campus IoT stress profiles with larger graphs and richer device vocabularies, motivated by the dynamic dependency and security requirements observed in sensor network studies [1,2], as shown in Table 3.

Protocol scope vs. engineering completeness. The current Python prototype is intentionally scoped to the core protocol logic that directly affects the security and filtering performance claims: branch extraction, HMAC-SHA256 based PRF/token generation, masked count vectors, update buffers, deletion tombstones, tree node aggregate computation, and epoch refresh. However, several components required for a deployable system are deliberately omitted because they are standard engineering layers that do not affect

the protocol-level security or filtering latency measurements reported in this paper. These omissions include: (i) authenticated encryption for graph records (the symbolic $\text{Enc}_K(G_i)$ is used instead of a concrete scheme such as AES-GCM); (ii) persistent client-side storage of epoch keys, logical ID maps, graph nonces, and update counters with rollback protection; (iii) crash recovery mechanisms for partially applied epoch refreshes; and (iv) an exact graph edit distance (GED) refinement engine executed on the client after decryption of returned candidates. Treating these as orthogonal allows us to isolate the contributions of the dynamic structured encryption design from implementation-dependent engineering choices and to focus the evaluation on the protocol-level tradeoffs.

Table 3. Prototype workload and reproducibility settings.

Component	Setting	Range/Value	Purpose
Graph database size	generated sensor topology graphs	250, 500, 1000, 2000, 4000 graphs	query scalability
Graph structure	connected sparse graph	8–22 vertices; ring backbone plus random extra edges	sensor network topology model
Stress profiles	mesh, industrial CPS, campus IoT	1500–4000 graphs; mean 15–56 vertices	robustness to heterogeneous sensor networks
Dynamic operations	insert, delete, label update, epoch refresh	five random seeds: 1701–1705	update and maintenance cost
Query threshold	graph edit distance threshold proxy	$\tau \in \{1, 2, 3, 4, 5, 6\}$	pruning sensitivity
Epoch length	refresh interval	50, 100, 250, 500, 1000 updates	privacy–efficiency tradeoff

The dynamic workload is denoted by

$$\mathcal{W} = \{\text{op}_1, \text{op}_2, \dots, \text{op}_T\}, \quad (64)$$

where $\text{op}_i \in \{\text{Insert}, \text{Delete}, \text{UpdateLabel}, \text{Search}, \text{ReRand}\}$. A workload profile is represented as

$$\pi = (p_{\text{ins}}, p_{\text{del}}, p_{\text{lab}}, p_{\text{qry}}, E), \quad (65)$$

where E is the epoch length. Insertions generate new sensor topology graphs, deletions insert epoch-specific tombstone handles, and label updates perturb local vertex or edge labels to regenerate only the affected branches. Queries are generated by perturbing existing graphs by one to three local label changes.

7.2. Baselines

The comparison includes both graph similarity baselines and dynamic encrypted search baselines. Table 4 summarizes their roles. Static PPGSQ1 performs full branch lower-bound scanning. Static PPGSQ2 uses the same size and branch count filtering principle as a branch–tree baseline, but does not include dynamic privacy. DSSE-branch represents a feature-only dynamic searchable encryption service that retrieves graphs sharing authorized sensor labels but does not maintain a graph similarity branch tree. Dynamic rebuild rebuilds the whole protected index after each update. Finally, the no-re-randomization baseline keeps stable branch identifiers, and is used only for linkability testing.

Table 4. Experimental baselines.

Method	Mechanism Profile			Purpose
	Query	Update	Privacy	
Static PPGSQ1-style	linear branch scan	rebuild only	content privacy	query-cost lower baseline
Static PPGSQ2-style	branch-tree pruning	rebuild only	content privacy	strongest static pruning baseline
Rebuild-after-update	branch-tree pruning	full rebuild	freshness	maintenance-cost upper baseline
DSSE-branch	feature-only lookup	dynamic update	forward privacy	cost of losing branch-tree pruning
No re-randomization	dynamic branch-tree	buffered update	partial privacy	value of epoch refresh
DFB-PPGSQ	dynamic branch-tree	buffer and compaction	forward/backward privacy	proposed construction

7.3. Metrics

For a query set $\mathcal{Q}$, average query latency is

$$\bar{T}_q = \frac{1}{|\mathcal{Q}|} \sum_{Q \in \mathcal{Q}} T_q(Q). \quad (66)$$

For an update stream $\mathcal{U}$, average update latency is

$$\bar{T}_u = \frac{1}{|\mathcal{U}|} \sum_{\text{op} \in \mathcal{U}} T_u(\text{op}). \quad (67)$$

The pruning ratio for query Q is

$$\text{PR}(Q) = 1 - \frac{|\text{Cand}(Q)|}{|\mathcal{G}|}. \quad (68)$$

The false positive candidate ratio after local refinement is

$$\text{FPR}(Q) = \frac{|\text{Cand}(Q) \setminus R^*(Q)|}{|\text{Cand}(Q)|}, \quad (69)$$

where $R^*(Q)$ is the exact result set under the graph edit distance threshold. Linkability attack success is measured as

$$\text{ASR} = \frac{\text{\#correctly linked epoch-crossing tokens}}{\text{\#tested token pairs}}. \quad (70)$$

Storage and communication overhead are not optimized in this Python prototype, and are not used for the main comparison. The reported metrics are query latency, update latency, epoch refresh maintenance cost, candidate pruning ratio, and linkability attack success. The stress profile experiment additionally reports a metadata-assisted linkability attack that combines branch label length, local degree, and coarse branch string length buckets. Latency consists of wall clock measurements in milliseconds.

7.4. Query Scalability

The first experiment measures query latency as the graph database grows from 250 to 4000 graphs. Figure 3 shows that DFB-PPGSQ remains close to the static branch-tree baseline while outperforming full scanning and feature-only DSSE at larger scales. At 4000 graphs, DFB-PPGSQ required 46.60 ms on average, compared with 44.72 ms for Static PPGSQ2, 62.22 ms for Static PPGSQ1, and 64.95 ms for DSSE-branch. Thus, the dynamic mechanisms add about 4.2% query overhead relative to the static branch-tree baseline at the largest tested scale while avoiding a return to linear scanning of the full database.

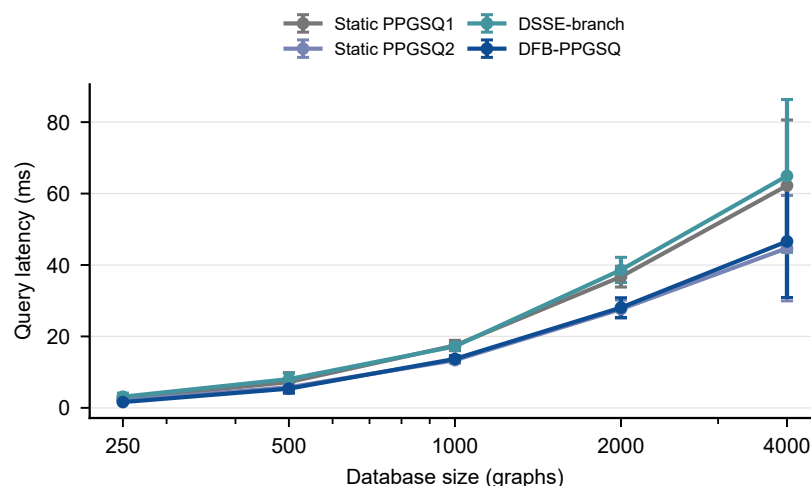


Figure 3. Query latency under increasing graph database scale. The legend has been moved outside the plotting area; points show mean values over five random seeds and error bars show standard deviations.

7.5. Dynamic Update Cost

The second experiment separates ordinary updates from epoch maintenance. Figure 4 shows that immediate rebuild is expensive for insertions, deletions, and label updates because the whole protected index is reconstructed after every operation. In contrast, DFB-PPGSQ appends insertions and label updates to the update buffer and stores deletions as tombstones. In the measured prototype, insertion and label update cost 0.091 ms and 0.092 ms on average, respectively, while delete cost 0.007 ms. Epoch refresh remains expensive because it rebuilds and re-randomizes the active index. Its mean cost was 192.73 ms, which is expected because DFB-PPGSQ moves rebuild cost from every update to scheduled epoch boundaries.

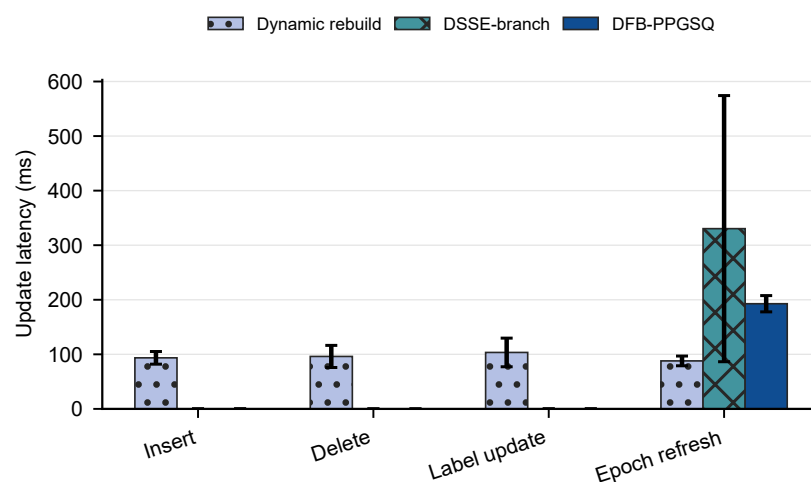


Figure 4. Dynamic update latency for insertion, deletion, label update, and epoch refresh. The legend has been separated from the bars; bars show mean values over five random seeds and error bars show standard deviations.

7.6. Re-Randomization Tradeoff

The third experiment varies the epoch length. A shorter epoch reduces the time window in which tombstone and update buffer traces remain linkable, but increases the amortized maintenance cost. Figure 5 shows this tradeoff. The measured maintenance cost falls from 4.03 ms per update at an epoch length of 50 updates to 0.177 ms per update at

1000 updates. The normalized linkability window grows in the opposite direction, since a longer epoch leaves stable within-epoch metadata visible for a longer period.

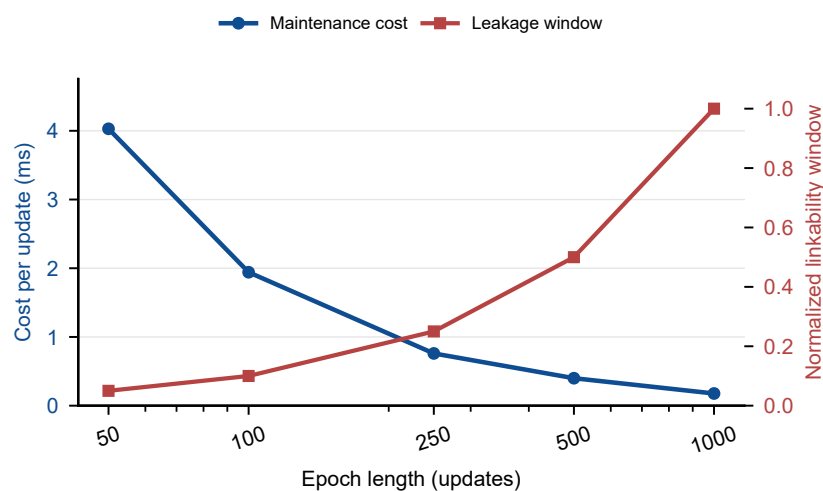


Figure 5. Epoch-length tradeoff between amortized re-randomization maintenance cost and normalized linkability window. The dual-axis legend has been placed above the plot to avoid crossing plotted lines or labels.

7.7. Pruning Ability

The fourth experiment evaluates whether dynamic privacy weakens branch-based pruning. Figure 6 reports the fraction of graphs removed before refinement. DFB-PPGSQ closely tracks the static branch-tree baseline because it preserves the same lower-bound filtering structure and adds only update buffer and tombstone checks. At $\tau = 1$, Static PPGSQ2 pruned 68.9% of candidates, while DFB-PPGSQ pruned 68.2%. At $\tau = 6$, the pruning ratios were 2.0% and 2.0%, respectively. The feature-only DSSE baseline pruned almost no graphs in this generated sensor workload because most graphs share at least one coarse sensor label; this illustrates why dynamic keyword-style search is not a substitute for graph similarity filtering.

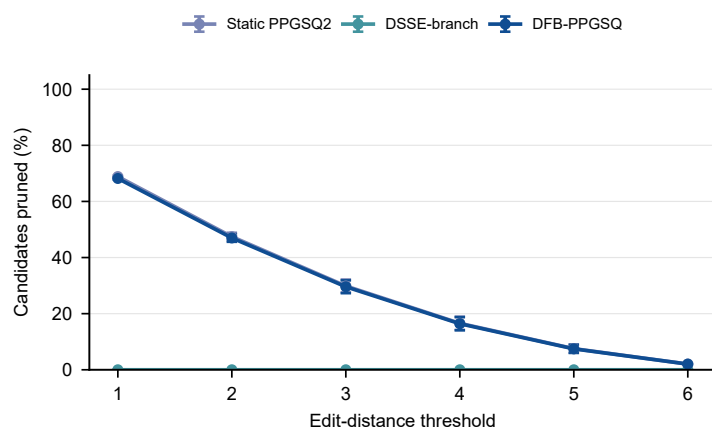


Figure 6. Candidate pruning ratio under different edit distance thresholds. The legend has been placed outside the plotting area; points show mean values over five random seeds and error bars show standard deviations.

7.8. Sensor Network Stress Workloads

The fifth experiment tests whether the same implementation remains stable under more heterogeneous sensor network profiles. Figure 7 reports three workloads. The default sensor mesh profile uses 4000 graphs with 14.98 vertices per graph on average. The industrial CPS and campus IoT profiles use 1500 larger graphs with 40.05 and 56.16 vertices

per graph on average, respectively, and include richer device and communication labels. DFB-PPGSQ required 52.92 ms on the 4000-graph mesh profile, 23.86 ms on the industrial CPS profile, and 23.80 ms on the campus IoT profile. The two profiles with larger graphs also produced stronger pruning, which is because their richer labels and higher local degree make branch summaries more selective. The, respective pruning ratios were 63.2% and 73.2%. Under a stronger metadata-assisted cross-epoch attacker using label length, degree, and coarse branch-string buckets, the measured attack success rates for the three profiles were 3.3%, 5.6%, and 4.8%, respectively. These results do not replace evaluation on public IoT traces, but give a stronger stress test than a single small synthetic topology.

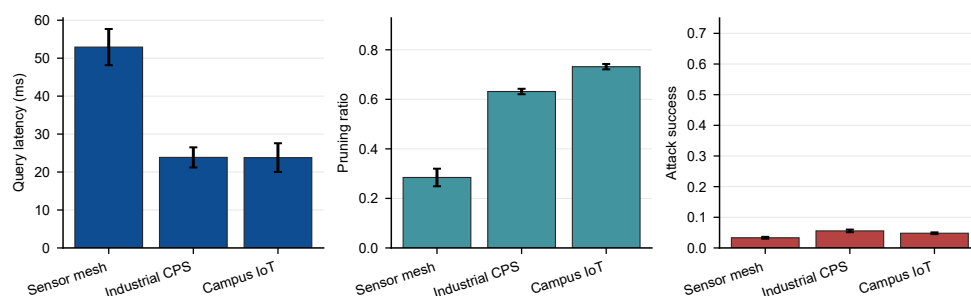


Figure 7. Sensor network stress workloads. The three panels report DFB-PPGSQ query latency, candidate pruning ratio, and metadata-assisted cross-epoch linkability attack success over mesh, industrial CPS, and campus IoT graph profiles, with adjusted panel spacing for readability. Bars show mean values over five random seeds and error bars show standard deviations.

7.9. Resilience to Leakage

The sixth experiment evaluates temporal linkability under a simple attacker. For the no-re-randomization baseline, the attacker links occurrence handles across epochs by deterministic equality. For DFB-PPGSQ, equality no longer works after epoch refresh, and the attacker falls back to coarse bucket matching based on branch label length and local degree. Figure 8 shows that stable identifiers make occurrence–handle linkage trivial in the baseline, whereas DFB-PPGSQ reduces the measured bucket-matching success rate to approximately 1.4–1.5%. While this experiment does not prove resistance to all leakage abuse attacks, it demonstrates that refreshing the epoch key removes the strongest equality signal under the declared leakage model, leaving only coarse metadata leakage.

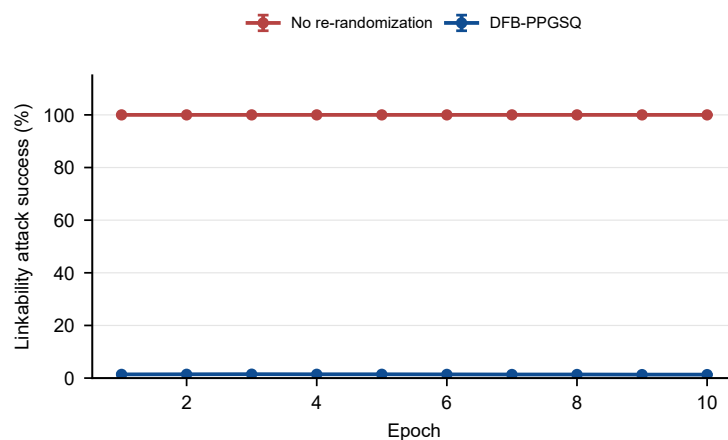


Figure 8. Linkability attack success rate across epochs with and without periodic re-randomization. The legend and vertical range have been adjusted so that the no-re-randomization curve is not obscured; points show mean values over five random seeds and error bars show standard deviations.

7.10. Prototype Scope and Deployment Limitations

The prototype is intended to evaluate the protocol-relevant filtering path, not to be a complete production encrypted graph database. It implements branch extraction, HMAC-SHA256 based PRF/token simulation, masked count vectors, update buffers, tombstones, and epoch refresh. A deployable system should instantiate $\text{Enc}_K(G_i)$ with authenticated encryption, ensure persistence of the client's epoch state and counters with rollback protection, provide crash recovery for partially applied refreshes, and use constant-time cryptographic library implementations. These engineering requirements are outside the current prototype, and are not included in the measured latency results.

The reported query latency measures server-side encrypted filtering before local exact refinement. In an end-to-end deployment, the client must decrypt returned candidates and run an exact or approximate graph edit distance refinement engine. The local refinement cost depends on the candidate count, query size, graph size, and chosen GED implementation. Thus, the pruning ratio experiment reports the main cost driver for refinement, and should not be read as a complete end-to-end latency claim.

The storage and communication overheads are also explicit. Let $\bar{B}$ be the average number of branch occurrences per graph, m the count vector length, λ the token length, and $|U_e|, |D_e|$ the current buffer and tombstone sizes. The protected server-side storage has the following order:

$$O\left(\sum_i |\text{Enc}(G_i)| + N\bar{B}\lambda + Nm \log p + |U_e|(\bar{B}\lambda + m \log p) + |D_e|\lambda\right). \quad (71)$$

A query token contains current-epoch feature tokens, protected query counts, and update buffer probes, giving communication on the order of

$$O(|B(Q)|\lambda + m \log p + |B(Q)| |U_e|\lambda), \quad (72)$$

while a full refresh uploads a shuffled active-state package of size

$$O(|\mathcal{M}_e|(\bar{B}\lambda + m \log p)). \quad (73)$$

These expressions explain why epoch length, update buffer growth, and vector length must be tuned for a target sensor cloud deployment.

The leakage resilience experiments test equality linkage removal and a metadata-assisted bucket attacker; however, they do not cover stronger statistical or machine learning attackers that combine traversal paths, repeated result sizes, public IoT graph distributions, update timing, and refresh sizes. The present construction also does not hide volume, result size, update time, or tree traversal. Deployments that require stronger leakage profiles should add padding, batched updates, dummy records, fixed-rate refresh, oblivious traversal, or ORAM-like access. The cost of these additions is not measured in the current prototype.

Finally, the threat model used in this paper is honest-but-curious. It does not verify whether the cloud returned all and only correct candidates; a malicious or compromised server could omit results, replay stale nodes, or return malformed ciphertexts unless the system is augmented with authenticated data structures, verifiable search proofs, or consistency audits. Supporting malicious-server integrity, multi-owner authorization, edge device energy profiling, public IoT traces, and explicit edge-insertion/edge-deletion implementations are important directions for future work.

8. Conclusions

This paper proposes DFB-PPGSQ, a dynamic forward-private and epochal backward-private graph similarity matching query scheme over encrypted graph databases. The scheme targets a realistic sensor cloud setting where graphs are not static and need to be inserted, deleted, relabeled, and periodically maintained.

DFB-PPGSQ combines branch-based lower-bound filtering with dynamic structured encryption. It introduces epoch-local feature tokens, per-record occurrence handles, update buffers, deletion tombstones, and shuffle-based periodic re-randomization. These mechanisms preserve efficient graph similarity filtering while reducing temporal leakage from dynamic updates. The security analysis shows how the scheme achieves forward privacy for future insertions and label updates and epochal backward privacy for deleted graphs after compaction under an explicit leakage model. The prototype evaluation supports the design goal under which ordinary updates avoid immediate full rebuild, query latency remains close to the static branch-tree baseline, stress workloads remain practical on heterogeneous sensor topologies, and epoch refresh removes stable cross-epoch token equality.

Future work will extend the prototype to public IoT and cyber-physical graph traces, optimize constant-time cryptographic implementations, measure client-side energy and exact refinement costs, and investigate verifiable multi-user and stronger leakage-resistant encrypted graph services.

Author Contributions: H.H. was responsible for generating the paper's ideas and writing the manuscript. Y.M. and Z.Z. conducted the experimental data collection and analysis. X.G. revised and reviewed the paper. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the National Natural Science Foundation of China (No. 62302152), the Henan Provincial Key Scientific and Technological Project (242102211074), the Zhengzhou Municipal R&D Special Fund-Supported Research Project (22ZZRDZX30), and the Open Project of Key Laboratory of Grain Information Processing and Control, Ministry of Education (KFJN2022017).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The prototype script, generated workload settings, measured CSV files, and figure-generation outputs are included with this manuscript package under `scripts/`, `source_data/`, and `figures/`. No human-subject data or third-party restricted datasets are used in the reported prototype evaluation.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

DFB-PPGSQ	Dynamic Forward/Backward-Private Privacy-Preserving Graph Similarity Query
DSSE	Dynamic Searchable Symmetric Encryption
GED	Graph Edit Distance
HMAC	Hash-Based Message Authentication Code
IoT	Internet of Things
PPGSQ	Privacy-Preserving Graph Similarity Query
SSE	Searchable Symmetric Encryption

References

1. Li, S.; Song, H.; Iqbal, M. Privacy and Security for Resource-Constrained IoT Devices and Networks: Research Challenges and Opportunities. *Sensors* **2019**, *19*, 1935. <https://doi.org/10.3390/s19081935>.
2. Wang, C.; Xing, S.; Gao, R.; Yan, L.; Xiong, N.; Wang, R. Disentangled Dynamic Deviation Transformer Networks for Multivariate Time Series Anomaly Detection. *Sensors* **2023**, *23*, 1104. <https://doi.org/10.3390/s23031104>.
3. Zheng, Y.; Zhu, H.; Lu, R.; Guan, Y.; Zhang, S.; Wang, F.; Shao, J.; Li, H. PGSim: Efficient and Privacy-Preserving Graph Similarity Query Over Encrypted Data in Cloud. *IEEE Trans. Inf. Forensics Secur.* **2023**, *18*, 2030–2045. <https://doi.org/10.1109/TIFS.2023.3262147>.
4. Wang, S.; Zheng, Y.; Jia, X.; Huang, H.; Wang, C. PrigSim: Towards Privacy-Preserving Graph Similarity Search as a Cloud Service. *IEEE Trans. Knowl. Data Eng.* **2023**, *35*, 10478–10496. <https://doi.org/10.1109/TKDE.2023.3266449>.
5. Ge, X.; Yu, J.; Hao, R. Privacy-Preserving Graph Similarity Matching Query Over Encrypted Graph Database. *IEEE Trans. Knowl. Data Eng.* **2026**, *38*, 1932–1946. <https://doi.org/10.1109/TKDE.2026.3656414>.
6. Sanfeliu, A.; Fu, K.S. A Distance Measure Between Attributed Relational Graphs for Pattern Recognition. *IEEE Trans. Syst. Man Cybern.* **1983**, *SMC-13*, 353–362. <https://doi.org/10.1109/TSMC.1983.6313167>.
7. Gao, X.; Xiao, B.; Tao, D.; Li, X. A Survey of Graph Edit Distance. *Pattern Anal. Appl.* **2010**, *13*, 113–129. <https://doi.org/10.1007/s10044-008-0141-y>.
8. Riesen, K.; Bunke, H. Approximate Graph Edit Distance Computation by Means of Bipartite Graph Matching. *Image Vis. Comput.* **2009**, *27*, 950–959. <https://doi.org/10.1016/j.imavis.2008.04.004>.
9. Yan, X.; Yu, P.S.; Han, J. Graph Indexing: A Frequent Structure-Based Approach. In Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, Paris, France, 13–18 June 2004; pp. 335–346. <https://doi.org/10.1145/1007568.1007607>.
10. Zheng, W.; Zou, L.; Lian, X.; Wang, D.; Zhao, D. Efficient Graph Similarity Search Over Large Graph Databases. *IEEE Trans. Knowl. Data Eng.* **2015**, *27*, 964–978. <https://doi.org/10.1109/TKDE.2014.2349924>.
11. Zhao, X.; Xiao, C.; Lin, X.; Wang, W. Efficient Graph Similarity Joins with Edit Distance Constraints. In Proceedings of the 2012 IEEE 28th International Conference on Data Engineering, Arlington, VA, USA, 1–5 April 2012; pp. 834–845. <https://doi.org/10.1109/ICDE.2012.91>.
12. Song, D.X.; Wagner, D.; Perrig, A. Practical Techniques for Searches on Encrypted Data. In Proceedings of the 2000 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 14–17 May 2000; pp. 44–55. <https://doi.org/10.1109/SECPRI.2000.848445>.
13. Boneh, D.; Di Crescenzo, G.; Ostrovsky, R.; Persiano, G. Public Key Encryption with Keyword Search. In *Proceedings of the Advances in Cryptology—EUROCRYPT 2004*; Springer: Berlin/Heidelberg, Germany, 2004; pp. 506–522. https://doi.org/10.1007/978-3-540-24676-3_30.
14. Chang, Y.C.; Mitzenmacher, M. Privacy Preserving Keyword Searches on Remote Encrypted Data. In *Proceedings of the Applied Cryptography and Network Security*; Springer: Berlin/Heidelberg, Germany, 2005; pp. 442–455. https://doi.org/10.1007/11496137_30.
15. Curtmola, R.; Garay, J.; Kamara, S.; Ostrovsky, R. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. In Proceedings of the 13th ACM Conference on Computer and Communications Security, Alexandria, VI, USA, 30 October–3 November 2006; pp. 79–88. <https://doi.org/10.1145/1180405.1180417>.
16. Cash, D.; Jarecki, S.; Jutla, C.; Krawczyk, H.; Rosu, M.C.; Steiner, M. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *Proceedings of the Advances in Cryptology—CRYPTO 2013*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 353–373. https://doi.org/10.1007/978-3-642-40041-4_20.
17. Kamara, S.; Papamanthou, C.; Roeder, T. Dynamic Searchable Symmetric Encryption. In Proceedings of the 2012 ACM Conference on Computer and Communications Security, Raleigh, NC, USA, 16–18 October 2012; pp. 965–976. <https://doi.org/10.1145/2382196.2382298>.
18. Kamara, S.; Papamanthou, C. Parallel and Dynamic Searchable Symmetric Encryption. In *Proceedings of the Financial Cryptography and Data Security*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 258–274. https://doi.org/10.1007/978-3-642-39884-1_22.
19. Cash, D.; Jaeger, J.; Jarecki, S.; Jutla, C.; Krawczyk, H.; Rosu, M.C.; Steiner, M. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In Proceedings of the 2014 Network and Distributed System Security Symposium, San Diego, CA, USA, 23–26 February 2014. <https://doi.org/10.14722/ndss.2014.23264>.
20. Stefanov, E.; Papamanthou, C.; Shi, E. Practical Dynamic Searchable Encryption with Small Leakage. In Proceedings of the 2014 Network and Distributed System Security Symposium, San Diego, CA, USA, 23–26 February 2014. <https://doi.org/10.14722/ndss.2014.23298>.
21. Bost, R. Σ₀φ₀: Forward Secure Searchable Encryption. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 1143–1154. <https://doi.org/10.1145/2976749.2978303>.

22. Bost, R.; Minaud, B.; Ohrimenko, O. Forward and Backward Private Searchable Encryption from Constrained Cryptographic Primitives. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October–3 November 2017; pp. 1465–1482. <https://doi.org/10.1145/3133956.3133980>.
23. Cao, N.; Yang, Z.; Wang, C.; Ren, K.; Lou, W. Privacy-Preserving Query over Encrypted Graph-Structured Data in Cloud Computing. In Proceedings of the 2011 IEEE 31st International Conference on Distributed Computing Systems, Minneapolis, MN, USA, 20–24 June 2011; pp. 393–402. <https://doi.org/10.1109/ICDCS.2011.84>.
24. Meng, X.; Kamara, S.; Nissim, K.; Kollios, G. GRECS: Graph Encryption for Approximate Shortest Distance Queries. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015; pp. 504–517.
25. Du, M.; Wu, S.; Wang, Q.; Chen, D.; Jiang, P.; Mohaisen, A. GraphShield: Dynamic Large Graphs for Secure Queries with Forward Privacy. *IEEE Trans. Knowl. Data Eng.* **2020**, *34*, 3295–3308. <https://doi.org/10.1109/TKDE.2020.3024883>.
26. Falzon, F.; Ghosh, E.; Paterson, K.G.; Tamassia, R. PathGES: An Efficient and Secure Graph Encryption Scheme for Shortest Path Queries. In Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security, Salt Lake City, 14–18 October 2024; pp. 4047–4061. <https://doi.org/10.1145/3658644.3670305>.
27. Yang, S. Privacy-Preserving Multi-User Graph Intersection Scheme for Wireless Communications in Cloud-Assisted Internet of Things. *Sensors* **2025**, *25*, 1892. <https://doi.org/10.3390/s25061892>.
28. Cash, D.; Grubbs, P.; Perry, J.; Ristenpart, T. Leakage-Abuse Attacks against Searchable Encryption. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015; pp. 668–679. <https://doi.org/10.1145/2810103.2813700>.
29. Naveed, M.; Kamara, S.; Wright, C.V. Inference Attacks on Property-Preserving Encrypted Databases. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015; pp. 644–655. <https://doi.org/10.1145/2810103.2813651>.
30. Grubbs, P.; Lacharite, M.S.; Minaud, B.; Paterson, K.G. Learning to Reconstruct: Statistical Learning Theory and Encrypted Database Attacks. In Proceedings of the 2019 IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 19–23 May 2019; pp. 1067–1083. <https://doi.org/10.1109/SP.2019.00030>.
31. Oya, S.; Kerschbaum, F. Hiding the Access Pattern is Not Enough: Exploiting Search Pattern Leakage in Searchable Encryption. In Proceedings of the 30th USENIX Security Symposium, Online, 11–13 August 2021; pp. 127–142.
32. Lewi, K.; Wu, D.J. Order-Revealing Encryption. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 1167–1178. <https://doi.org/10.1145/2976749.2978376>.
33. Chase, M.; Kamara, S. Structured Encryption and Controlled Disclosure. In *Proceedings of the Advances in Cryptology—ASIACRYPT 2010*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 577–594. https://doi.org/10.1007/978-3-642-17373-8_33.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.