Journal of Information Security and Applications 99 (2026) 104439

Contents lists available at ScienceDirect

Journal of Information Security and Applications

journal homepage: www.elsevier.com/locate/jisa

Practical and secure history-independent indexing for queryable-encrypted databases

Mattia Trabucco *, Mauro Andreolini, Luca Ferretti

Department of Physics, Informatics, Mathematics, University of Modena and Reggio Emilia, Italy

ARTICLE INFO

Keywords:
Encrypted database
Queryable encryption
History-independent data structure
History independence
Order revealing encryption
Skip list

ABSTRACT

Queryable encryption denotes a class of techniques which enable efficient query processing on encrypted databases, but may be affected by severe leakage if associated with improper indexes for achieving sublinear times in single-round query protocols. In this paper, we present an indexing data structure based on skip lists which does not introduce any additional leakage than the order of encrypted records: our index is history-independent, does not leak duplicates, is optimized for external-memory and range queries, and operates with a stateless client, making it well-suited for deployment in real-world databases. Previous works use no indexes or multi-round protocols, possibly with stateful clients, to achieve best security, but affect performance and alter the setting of existing databases, thus limiting deployability. Otherwise, they adopt standard indexes already available within the database at the cost of affecting security guarantees, or design in-memory data structures which do not suit database contexts. We demonstrate the practicality of our index by developing a prototype extension for PostgreSQL and for Order Revealing Encryption, which achieves performance that is comparable to the standard balanced tree implementation for up to 1M records, and acceptable overhead for encrypted data when scaling to 10M records.

1. Introduction

Standard encryption solutions for databases protect *data at rest* [1], but require database servers to decrypt data at runtime to execute queries. First motivated by the advent of database outsourcing and then by the popularity of public Cloud computing services, encryption techniques to protect *data in use* have been designed to protect data privacy against potentially *curious* service providers or database administrators [2]. Nowadays, encryption of *data in use* is also relevant for improving security of on-premise solutions in case of data breaches caused by cyber attacks, that subvert database systems or software applications running on top of them [3]. However, encryption alone may not be sufficient to protect information regarding the history of insertions or deletions, which is highly sensitive information for many types of databases, such as those related to law enforcement, journalists or voting systems, where history may allow to identify records associated with certain individuals (e.g, revealing the information about how and when the data was collected may reveal sources that a law enforcement agency wants to stay private). Indeed, transactions history could be disclosed by the topology of the indexing data structure or even by its bit representation. The goal of history independence is to let the memory representation of a data structure depend only on the content of the data structure and not on the sequence of operations that led to that content [4], and the notion of history independence derives from the concept of unique representation [5] and uniformly represented randomized structures [6,7]. History-independent data structures naturally support information-theoretically-secure deletion, unlike standard secure delete methods (where the file system overwrites deleted data with zeros) that may reveal how much and where data was deleted. Thus, they also help avoiding subtle and hard-to-quantify side channels which sometimes affect information confidentiality, even when dealing with encrypted data.

Queryable encryption [8–11] (also called *property-preserving encryption*) denotes a popular class of techniques for achieving practical performance while exposing "limited" information leakage, including the result of the evaluation and potentially additional information that is specific for each scheme. However, as the authors first observed in [12], even the most secure queryable encryption schemes offering semantic security of stored encrypted data could be affected by severe leakage if associated with improper indexes.

In this paper, we extend our preliminary results [12] and propose an indexing data structure that can be used to index encrypted records

* Corresponding author.
E-mail addresses: mattia.trabucco@unimore.it (M. Trabucco), mauro.andreolini@unimore.it (M. Andreolini), luca.ferretti@unimore.it (L. Ferretti).

https://doi.org/10.1016/j.jisa.2026.104439

Available online 17 March 2026

with minimal leakage when deployed with stateless clients and single round query-protocols, that is, based on the same system model that characterizes existing databases. We implement it as an extension of the popular open-source PostgreSQL database for Order Revealing Encryption (ORE), and we propose an extensive performance evaluation for very large databases including up to 10M records. The proposed index does not leak duplicates, is history-independent, and is optimized for external-memory and range queries, making it well-suited for real-world databases. Our design is a variant of the skip list designed by Bender et al. [13], which guarantees efficient access to external memory and history-independence but leaks duplicate values as its design is not specialized for encrypted data. Intuitively, our design avoids leaking duplicates by letting each index node store only one record identifier, and guarantees efficient lookup performance by structuring the list at each level as a doubly-linked list with both forward and backward pointers. Moreover, we try to fill the gap between the theoretical work by Bender et al. and our practical implementation by giving some details that the original work does not provide.

We are interested in *snapshot adversaries* both for data and for the index, thus we focus on encryption schemes which guarantee *semantic security* of stored data and on *weakly history-independent data structures*, which guarantee history independence against adversaries which only observe the indexing data structure once. The adversary that obtains a snapshot of the database, including the index, may infer information about the transaction history, potentially also revealing information about the distribution of plaintext records if such distribution is not independent of the transaction history. Different types of techniques, such as Oblivious RAM [14–16], assume weaker security models including online and/or persistent adversaries, but modify the typical database management system models by requiring stateful clients and multi-round query protocols, making them impractical in many real-world scenarios (see Section 3.3).

Typical standard indexing data structures which have been used for encrypted data such as AVL trees [17], BTrees [18,18] and B+trees [19] store all pointers to duplicate database values within the same index node to improve query performance and to reduce the overall index size, thus leaking information about duplicate values as if adopting a deterministic encryption scheme and possibly re-enabling powerful inference attacks [20]. The same indexes may leak information regarding the history of the transactions executed on the database, which could act as side-channel information even when encrypting records. While periodic re-balancing by the database server is a trivial but effective strategy to mitigate leakage of history information when dealing with plaintext information [12,13,21], the database server cannot re-balance an index built over semantically secure encrypted data without the assistance of a trusted party which must download and decrypt the whole database, re-build the encrypted index, and upload it again on the server. Thus, in some sense, designing history-independent indexing data structures may be even more important for encrypted databases than for plaintext data. Known history-independent variants of B-trees like B-treaps [22] and of skip lists [13] leak information about duplicates due to the same reasons, and cannot be adopted as-is for encrypted databases. While adapting the B-treap [22] seems more arduous (if possible), and left as future work, we propose modifications to [13] and design the first data structure for external memory which both guarantees (weakly) history independence and does not affect the semantic security of encrypted data.

While the proposed data structure can be used to index data encrypted with other queryable encryption schemes, for our implementation we focus on supporting the ORE scheme by Lewi and Wu (LW-ORE) [8]. ORE is specialized on comparison of numerical data (e.g., “greater than” operators), but also supports other derived operations, such as sorting and substring prefix/suffix matching, making it particularly well-suited for a database scenario. We show that the performance of our indexing data structure is practical and often comparable to that of the balanced tree implementation included within the DBMS in case of data encrypted with LW-ORE.

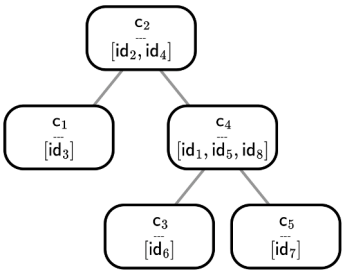


Fig. 1. Example of AVL tree for indexing queryable-encrypted records.

The rest of the paper is organized as follows. In Section 2, we discuss background knowledge and notation. In Section 3, we overview the system and threat models. In Section 4, we describe the proposed indexing data structure and discuss its security guarantees, and in Section 5 we describe its implementation for ORE as a PostgreSQL extension. In Section 6, we present a formal security analysis of minimal leakage for the proposed data structure. In Section 7, we present the results of our experimental evaluation. In Section 8 we discuss related work, and in Section 9 we conclude the paper.

2. Base knowledge and notation

2.1. Indexing duplicate values with standard data structures

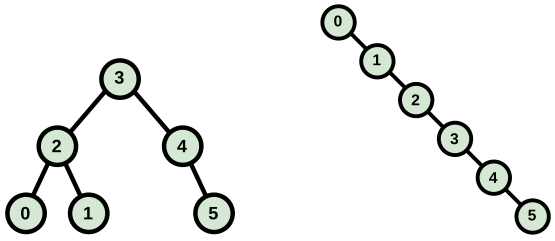
The typical design choice of any indexing data structure is to store all pointers to duplicate database values within the same node to improve query performance and to reduce the index size. The result is that applying standard indexes to queryable-encrypted records leaks *duplicate values* within the database, and thus the security guarantees of semantically secure schemes fall back to those of a deterministic encryption scheme, possibly re-enabling related inference attacks [20].

To better explain this class of vulnerability, we consider the example shown in Fig. 1 based on an AVL tree, which represents the implementation of [17]. The *values* are semantically secure ciphertexts $\{c_1, \ldots, c_5\}$, and the pointers to records are identifiers $\{id_1, \ldots, id_8\}$. It is clear that adversaries that access the AVL tree know that records identified with id_2 and id_4 (id_1, id_5 and id_8) are associated with the same *value*, even if they are not able to compute the encrypted comparison function or know the corresponding plaintext value. Similar analyses apply for B+trees considered for encrypted data in [19], the B-tree default index in PostgreSQL,[1] and history-independent B-treaps [22].

2.2. History independence

We clarify potential information leakage of a *history dependent* data structure by considering the example in Fig. 2. We consider two Binary Search Trees (BSTs), both generated by providing the same set of values $\{0, \ldots, 5\}$ in different orders. The example uses plaintext values for better clarity, but the same leakage applies to encrypted values. The first BST (Fig. 2a) is built from the insertion sequence $SEQ_A = \{3, 2, 4, 0, 1, 5\}$. The second BST (Fig. 2b) is built from insertion sequence $SEQ_B = \{0, 1, 2, 3, 4, 5\}$. The structure of the second BST leaks that data have been inserted within the database in ascending order. The result is that, even if two databases store the same values, the topologies of their indexes may differ if the insertion order of the values is not the same, and thus, leak information about the transactions history.

[1] See PostgreSQL documentation, Chap. 67.4.3, B-Tree Indexes, Deduplication, https://www.postgresql.org/docs/16/btree-implementation.html



(a) Insertion history $SEQ_A = \{3, 2, 4, 0, 1, 5\}$ (b) Insertion history $SEQ_B = \{0, 1, 2, 3, 4, 5\}$

Fig. 2. Example of two plaintext BSTs with different insertion histories.

Re-balancing the BST may help hiding information leakage related to the data structure topology but, when dealing with plaintext records, frequent re-balancing may critically affect performance and information may still leak from the internal bit representation [23]. Even more importantly, when dealing with semantically secure encrypted records, as we are considering, the DBMS is not able to autonomously operate re-balancing operations because it can not directly perform encrypted comparisons, thus needing the help of a trusted party which possesses decryption keys, such as a client. *Thus, re-balancing strategies are absolutely impractical in the context of semantically secure encrypted records*.

2.3. Skip lists

A skip list is a hierarchical data structure composed of multiple layered linked lists [6]. The lowest level (*leaf level*) is the *complete* list of all the sorted values stored within the data structure. Each higher level consists of a *sparse* list which includes a subset of the values of the list at the level below, and acts as a sort of "fast track" skipping over intermediate nodes for a quicker access. In particular, at insertion time, each value has probability p of being *promoted* to the higher layer, from leaf level until promotion fail (or until a certain maximum level is reached, depending on implementations). The promotion probability p is typically set to $\frac{1}{2}$ or $\frac{1}{4}$ and determines the size of sparse lists and thus the storage overhead of the skip list. A skip list inherently offers a weak history-independent topology because the structure of each sparse list is completely independent of the content actually stored and on the transaction history, without requiring any re-balancing operation. These properties make skip lists a good candidate as an indexing data structure for encrypted databases.

2.4. Lewi-Wu ORE

The Order Revealing Encryption scheme by Lewi and Wu [8] (LW-ORE) is a symmetric encryption scheme for supporting comparison between encrypted data. The peculiar trait of LW-ORE, previously also adopted in [9], is to include two types of encryption functions denoted as *right encryption* and *left encryption*, each generating a different type of ciphertext from the same plaintext: semantically secure *right ciphertexts* which are stored in the encrypted database, and deterministic *left ciphertexts* which are sent when querying the database. LW-ORE uses an evaluation function that compares *left* and *right ciphertexts* and output the result (e.g., 0 or 1 if the plaintext used to generate the *left ciphertext* is equal or greater than that used to generate the *right ciphertext*). Formally, LW-ORE includes four algorithms: *setup* (ORE.Setup), *left encrypt* ($\mathsf{ORE.Encrypt_L}$), *right encrypt* ($\mathsf{ORE.Encrypt_R}$), and *compare* (ORE.Compare):

- $\mathsf{sk} \xleftarrow{\$} \mathsf{ORE.Setup}(1^\lambda)$ is a probabilistic algorithm which takes security parameter 1^λ and outputs secret key sk;
- $\mathsf{c}^\mathsf{L} \leftarrow \mathsf{ORE.Encrypt_L}(\mathsf{sk}, \mathsf{p})$ is a deterministic algorithm which takes key sk and plaintext p and outputs *left ciphertext* c^L;
- $\mathsf{c}^\mathsf{R} \xleftarrow{\$} \mathsf{ORE.Encrypt_R}(\mathsf{sk}, \mathsf{p})$ is a probabilistic algorithm which takes key sk and plaintext p and outputs *right ciphertext* c^R;
- $\{-1, 0, 1\} \leftarrow \mathsf{ORE.Compare}(\mathsf{c}_1^\mathsf{L}, \mathsf{c}_2^\mathsf{R})$ is a deterministic algorithm which takes *left* and *right ciphertexts* c_1^L and c_2^R, and outputs -1, 0, 1 if p_1 is less than, equal, or greater than p_2, where $\mathsf{c}_1^\mathsf{L} \leftarrow \mathsf{ORE.Encrypt_L}(\mathsf{sk}, \mathsf{p}_1)$ and $\mathsf{c}_2^\mathsf{R} \xleftarrow{\$} \mathsf{ORE.Encrypt_R}(\mathsf{sk}, \mathsf{p}_2)$, for any $\mathsf{sk} \xleftarrow{\$} \mathsf{ORE.Setup}(1^\lambda)$.

LW-ORE does not expose a native decryption routine, but decryption can be implemented as a binary search over the plaintext domain via encryption and compare (see [8, Remark 2.1]). In the context of encrypted databases, ORE ciphertexts can be associated with ciphertexts computed with any standard symmetric scheme with semantic security to enable more efficient decryption, and in this paper we also comply with this approach (see Section 3.1).

2.5. LW-ORE range query protocol for encrypted databases

LW-ORE scheme allows designing a *stateless* and *single round* Range Query protocol (RQ) [8, Section 5.2] among a *client* and a database *server* that includes four operations: database *setup* (RQ.Setup), *range* query (RQ.Range), *insert* (RQ.Insert), and *delete* (RQ.Delete). Without loss of generality, in this paper we omit *delete* to avoid too much verbosity.

- $\mathsf{st} \xleftarrow{\$} \mathsf{RQ.Setup}(1^\lambda, P)$: the client initializes ORE secret key $\mathsf{sk} \xleftarrow{\$} \mathsf{ORE.Setup}(1^\lambda)$, sorts the database P as $\hat{P} = \langle \mathsf{p}_1, \dots, \mathsf{p}_N \rangle$ such that $\mathsf{p}_n \le \mathsf{p}_{n+1}\ \forall n \in [N-1]$, and encrypts the sorted database $\hat{P}$ as $\hat{C} = \langle \mathsf{c}_1^\mathsf{R}, \dots, \mathsf{c}_N^\mathsf{R} \rangle$ such that $\mathsf{c}_n^\mathsf{R} \xleftarrow{\$} \mathsf{ORE.Encrypt_R}(\mathsf{sk}, \mathsf{p}_n)\ \forall n \in [N]$. The client sends $\hat{C}$ to the server, which sets its state information $\mathsf{st} = \hat{C}$.
- $\langle \mathsf{p}_i, \dots, \mathsf{p}_j \rangle \leftarrow \mathsf{RQ.Range}(\mathsf{sk}, \mathsf{q} = (x, y), \mathsf{st} = \hat{C})$: the client computes the tuple $\langle \mathsf{c}_x^\mathsf{L}, \mathsf{c}_y^\mathsf{L} \rangle$ such that $\mathsf{c}_x^\mathsf{L} \leftarrow \mathsf{ORE.Encrypt_L}(\mathsf{sk}, x)$ and $\mathsf{c}_y^\mathsf{L} \leftarrow \mathsf{ORE.Encrypt_L}(\mathsf{sk}, y)$, and sends it to the server. The server uses c_x^L (c_y^L) to find c_i^R (c_j^R) within $\hat{C}$ such that i (j) is the smallest (greatest) index of $\hat{C} = \langle \mathsf{c}_n^\mathsf{R} \rangle_{n \in [N]}$ such that $\mathsf{ORE.Compare}(\mathsf{c}_x^\mathsf{L}, \mathsf{c}_i^\mathsf{R}) = -1$ ($\mathsf{ORE.Compare}(\mathsf{c}_y^\mathsf{L}, \mathsf{c}_j^\mathsf{R}) = 1$). As suggested in the original protocol [8], the server can use binary search to compute a number of ORE.Compare operations that is logarithmic in the size of the database N. The server then returns the slice of values within $\hat{C}$ between $\langle \mathsf{c}_i^\mathsf{R}, \dots, \mathsf{c}_j^\mathsf{R} \rangle$, which the client can decrypt to $\langle \mathsf{p}_i, \dots, \mathsf{p}_j \rangle$.
- $\mathsf{st}' \xleftarrow{\$} \mathsf{RQ.Insert}(\mathsf{sk}, \mathsf{q} = x, \mathsf{st} = \hat{C})$: the client computes the tuple $\langle \mathsf{c}_x^\mathsf{L}, \mathsf{c}_x^\mathsf{R} \rangle$ such that $\mathsf{c}_x^\mathsf{L} \leftarrow \mathsf{ORE.Encrypt_L}(\mathsf{sk}, x)$ and $\mathsf{c}_x^\mathsf{R} \xleftarrow{\$} \mathsf{ORE.Encrypt_R}(\mathsf{sk}, x)$, and sends it to the server. The server uses c_x^L to find an insertion position i within $\hat{C}$ such that $\mathsf{ORE.Compare}(\mathsf{c}_x^\mathsf{L}, \mathsf{c}_i^\mathsf{R}) = \{-1 \vee 0\}$ and $\mathsf{ORE.Compare}(\mathsf{c}_x^\mathsf{L}, \mathsf{c}_{i+1}^\mathsf{R}) = \{0 \vee 1\}$ (or $i = N$). As for RQ.Range, binary search can be used to compute a logarithmic number of ORE.Compare operations. Then, the server inserts c_x^R at the position i within the updated database $\hat{C}'$. The server updates its state to st' as the new database $\hat{C}'$. Clearly, there may be multiple acceptable values of i if the database includes duplicate values.

3. System design overview

We overview the system architecture (Section 3.1), information flow (Section 3.2), and threat model (Section 3.3).

3.1. Architecture

Fig. 3 shows the architecture of the system, which includes a *client* and a DataBase Management System (*DBMS*).

The client is a software component, such as a standalone application or a transparent proxy as typical in literature (see Section 8), which can perform any CRUD operation (Create, Read, Update, Delete) on the DBMS, interacting through single-round protocols. We assume that the client knows all cryptographic keys needed to encrypt query tokens and

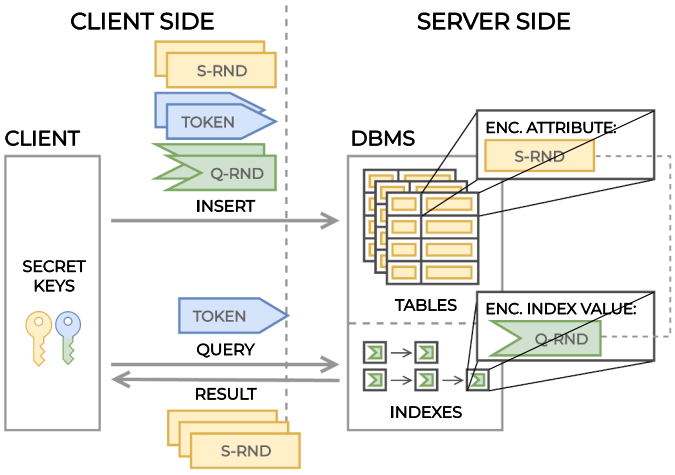


Fig. 3. System architecture showing how different types of cryptographic data are exchanged between client and DBMS, and stored within tables and indexes.

decrypt results. Key distribution is orthogonal to our design. The client is stateless in the sense that it does not maintain additional information needed for the protocols to be executed (e.g., portions of indexes, auxiliary metadata), and all due information is maintained by the DBMS.

The DBMS stores encrypted records within tables, maintains indexes, and supports execution of the encrypted comparison operator and possibly standard functionalities such as triggers. Depending on the queryable encryption scheme, encrypted records may include different types of ciphertexts:

- **S-RND**: data encrypted with *standard* semantically secure encryption scheme (e.g., standard IND-CPA, IND-CCA or IND-CCA2 encryption mode based on AES, as we show in detail in Section 5), which are stored within tables, and used as inputs in insert operations, and as outputs in queries;
- **Q-RND**: data encrypted with semantically secure *queryable* encryption scheme, which are stored within indexes at server side, and used as inputs in insert or update operations[2];
- **TOKEN**: encrypted query tokens generated by the queryable encryption scheme, which are used at runtime during query processing to access indexes and allow the encrypted comparison with the Q-RND ciphertexts already stored in the database indexes. One or more TOKENs are sent in any CRUD operation. The database does not need to persistently store TOKENs, and *should not* when TOKENs leak additional information than Q-RND.

The different types of ciphertexts are not opaque to the DBMS, in the sense that the DBMS needs to selectively handle the different types for supporting the encrypted query protocol (see Section 3.2) and avoid leaking information which may decrease security of the queryable encryption scheme.

We note that all three types of ciphertexts are associated with the same plaintext, and must not be confused with different types of queryable encryption schemes that are used to support different types of query operations [25,26].

We also remark that, depending on the queryable encryption scheme adopted, the three ciphertexts presented above may coincide. For example, if the queryable encryption scheme is based on fully homomorphic encryption, S-RND = Q-RND = TOKEN, therefore the same ciphertext can be stored in the tables, indexes, and used as input in the query protocol.

3.2. Information flow

We describe the overall information flow for the execution of insert and query operations, highlighting the role of the different types of ciphertexts as also shown in Fig. 3. We omit update and delete operations as they can be considered minor variants. All operations are executed through single-round protocols: no additional communication is needed between the client and the DBMS to complete the described operations.

To insert a record, the client encrypts each plaintext attribute with all three types of encryption (i.e., generates S-RND, Q-RND and TOKEN ciphertexts for each attribute of the record), and sends them to the DBMS. In the example shown in Fig. 3, the record has two attributes, and the client sends two S-RND, two Q-RND and two TOKEN ciphertexts to the DBMS. For each attribute, in order to find the correct position to insert the new attribute, the DBMS takes the TOKEN ciphertext of the attribute and runs the encrypted comparison function with all the Q-RND ciphertexts already included in indexes, similarly to the normal behavior for inserting new plaintext data. However, the DBMS stores in the index only the Q-RND ciphertext of the attribute. After updating the indexes, the DBMS stores only the S-RND ciphertext of the attribute in the table. Thus, the DBMS never permanently stores the TOKEN ciphertext, which may be deterministic and could be used by an attacker to break the semantic security guarantees of the queryable encryption scheme. We remind that, since we are considering single-round query protocols, the Q-RND ciphertext stored in a database index is associated with a pointer (id) to the corresponding record in the table used to retrieve the record when a query is executed (see Section 2.1).

To query the database, the client creates a select statement by encrypting all the due filtering clauses through TOKEN generation, and sends the *encrypted query* to the DBMS. First, the DBMS uses TOKEN ciphertexts to traverse the indexes and find the encrypted records that match the query, as for insertion. Then, the DBMS retrieves the corresponding S-RND ciphertexts, which are stored in the table, and sends them back to the client. Finally, the decryption of the resulting records is performed by the client using the decryption function of the S-RND scheme. The DBMS cannot decrypt the data or the query result because it does not have access to the secret keys.

We also note that, if it is impossible to directly compare two Q-RND ciphertexts (e.g., LW-ORE, as explained in Section 2.4) and the database does not store TOKENs, the DBMS loses the ability to construct the index at a later point in time.

3.3. Threat model

In this paper, we focus on passive offline attackers which may obtain a snapshot of the database (tables and indexes included) [27]. We do not consider online attackers who compromises the DBMS server and passively observes all its operations, including the ability to persistently observe the queries issued to the database and how they access the encrypted data [28]. Moreover, we assume that the attacker follows the protocol as intended, and does not actively interfere with the system (e.g., modifying the data, the queries, or the results). Thus, the attacker cannot insert, delete, or modify the records stored in the DBMS. Security solutions in this context relate to data integrity guarantees and involve cryptographic tools which are orthogonal to our proposal [29]. Finally, we comply with typical threat models for database encryption in use [27] and assume that the client is secure and trusted.

The practical trade-offs of offline security. Someone may wonder whether snapshot adversaries are strong enough, especially considering that stronger attacker models are realistic and may obtain useful information [30,31], and that stronger techniques exist to defend against online persistent adversaries, such as Oblivious RAM (ORAM) [14–16] which hide access patterns on outsourced databases (see Section 8).

[2] Note that some queryable encryption scheme may not guarantee semantic security, possibly due to deterministic behavior [24]. However, since we are designing an indexing data structure which does not affect semantic security, such a design makes sense only when deployed in combination with semantically secure queryable encryption.

However, techniques to defend against online persistent adversaries, such as ORAM, require multi-round protocols, stateful clients, and redesigning the database management system, thus their integration in many real-world settings may not be possible or they may achieve impractical performance. We consider the same system model and communication paradigms of existing databases, based on stateless clients and single-round query protocols. To the best of our knowledge, no queryable encryption scheme has been designed in this setting with equal or less information leakage than our proposal. By achieving acceptable performance overhead in unmodified database settings, our proposal represents a strong security measure against data breaches and a mitigation against more complex adversaries. We leave evaluating the integration of the proposed data structure with techniques such as ORAM as future work.

4. Indexing data structure

We give details on history-independent skip lists for external memory, which is the starting point of our design and still leak duplicates, in Section 4.1, then we describe the design of the proposed indexing data structure in Section 4.2, and finally we discuss the security guarantees of our design and how it achieves minimal information leakage in Section 4.3.

4.1. History-independent external-memory skip list

As shown in Fig. 4a, the History-Independent External-Memory (HI-EM) skip list proposed by Bender et al. [13] optimizes skip lists storage into *disk pages*. To this aim, it organizes *disk pages* into *slots*, and given a skip list-like data structure (see Section 2.3), both the *complete* list at leaf level and *sparse* lists at higher levels are implemented as lists of slots, and for supporting efficient insertion of new values, both of them may include *empty slots*. HI-EM also defines *arrays* as sequences of slots, where the first slot includes a promoted value, and the other ones include a non-promoted value or are empty slots. Note that empty slots can only exist at the end of the array (that is, within the same array, no empty slot is followed by a value). The values stored within each array appear in sorted order. Each array is stored into one or multiple disk pages, and organized differently depending on the level in which it resides:

- arrays at leaf level are denoted as *leaf arrays*. HI-EM denotes a data structure called *leaf node* as a set of contiguous leaf arrays, where the first value stored in the first leaf array has been promoted at least twice (that is, it exists at least at level 1 and 2), and all the other leaf arrays only includes values promoted at most at level 1. Since the end of each leaf array may include empty slots, the leaf node may be considered as a list including sparse empty slots (that is, not only at the end of the leaf node, but also in intermediate positions).
- arrays at non-leaf levels are denoted as *internal arrays*. As opposed to *leaf arrays*, internal arrays are not packed together into *nodes*, but may still be stored in one or multiple disk pages. Thus, empty slots within internal arrays are only present in the last page in which the array is stored.

The promotion probability for the HI-EM skip list is $p = 1/B^{\gamma}$, such that $\frac{1}{2} < \gamma \leq 1 - \log(\log(B))/\log(B)$, where B is the total number of slots that can be stored in a disk page, and γ is a parameter chosen at configuration time. Tuning the parameter γ allows for a trade-off between the cost of a range query and the worst-case cost of an insertion operation. As the value of γ approaches $1 - \log(\log(B))/\log(B)$, the cost of range queries decreases, and the worst-case cost of insertions increases (and vice versa as γ approaches $\frac{1}{2}$). This is due to the impact on the expected number of promoted values: as the promotion probability p decreases, the leaf arrays store more values contiguously, and thus range queries become more efficient because fewer disk pages need to be accessed. Also, as p decreases, the search for the correct position for inserting a new value becomes slower, since fewer promoted values can be used to quickly reach the leaf level.

At the leaf level, each slot including a value stores the record identifier of the corresponding indexed record in the database table, while at non-leaf levels it stores a pointer to the same value at the lower level.

The size of each leaf array is chosen history-independently [32], and each disk page is allocated history-independently [33]. As a result, the list is *weakly history-independent* [13, Section 6.3], which makes it a good fit for *snapshot passive adversaries* considered in our threat model (Section 3.3).

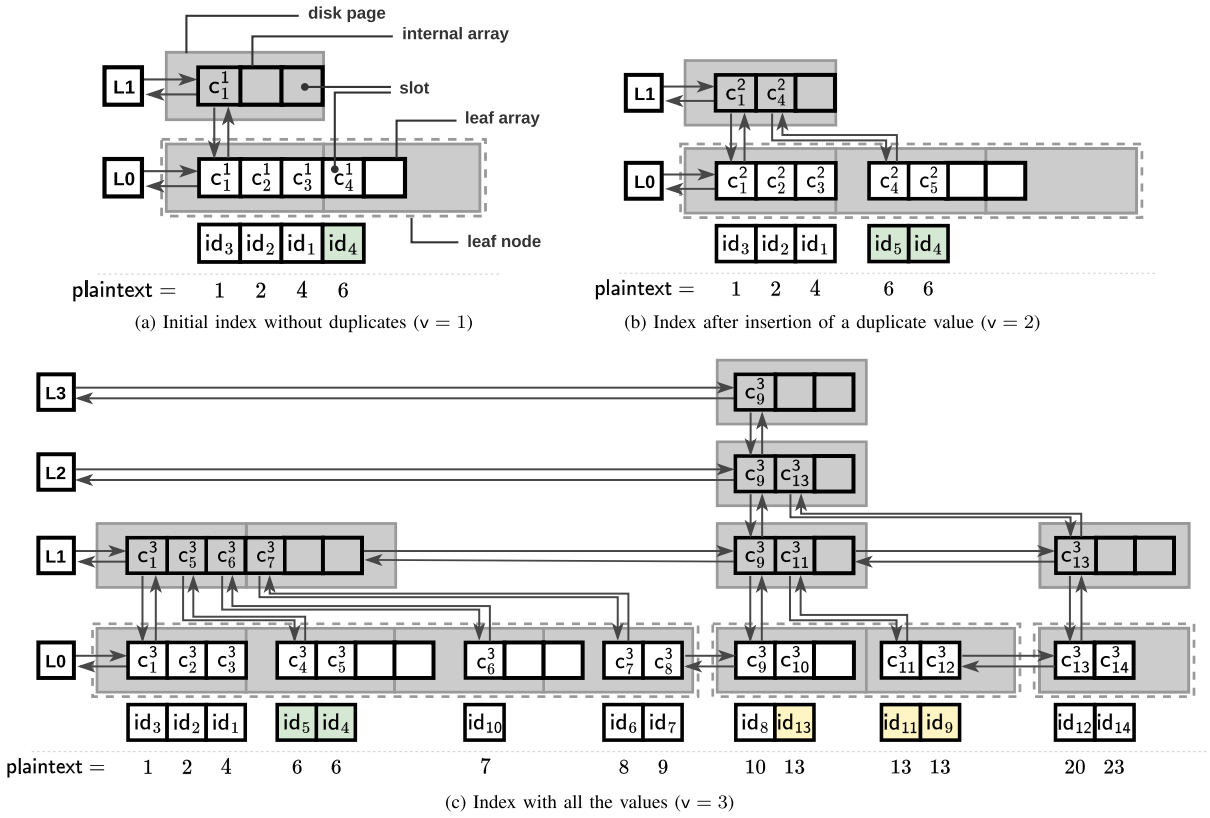
4.2. Doubly-linked history-independent external-memory skip list

The HI-EM skip list leaks information about duplicates, as all the record identifiers associated with the same indexed attribute value are stored within a single slot at leaf level. This is a common design choice for standard indexing data structures (see Section 2.1). Indeed, Bender et al. [13] did not deal with encrypted data and thus information leakage due to duplicates is out of their design aims. To minimize the information leakage of the HI-EM skip list while still improving query performance of encrypted databases, we design an indexing data structure that we call *Doubly-linked History-independent External-memory skip list* (DHE skip list) which differs for two modifications to the original design by Bender et al.:

- to avoid leaking duplicates, we let each slot at leaf level store only one record identifier. However, note that by performing this modification, traversing the index may lead to the first slot of a leaf array that is not the *first* of all the slots including the searched value (remind that, if there exist duplicate values within the skip list, they are stored in virtually contiguous leaf-level slots). Retrieving all values both in exact-match and range queries would require reverse navigation within non-leaf-level slots and long scans at leaf-level slots, thus leading to inefficient index lookup. A similar disadvantage was already observed in [12] for in-memory skip lists.
- to guarantee efficient lookup performance, we design *lists* at both leaf and non-leaf levels, which originally are *linked lists* with forward pointers only, as *doubly-linked lists* provided with both forward and backward pointers. Thus, even if traversing the index does not lead to the first slot in a sequence of slots associated with the same value, we can efficiently scan the leaf slots backwards. Moreover, we also add a backward pointer between slots at different levels of the skip list (that is, from a value to its corresponding promoted value in a slot at level above, if it exists) to efficiently maintain consistency during array resizing.

For a better comprehension, we describe how the proposed DHE skip list evolves throughout insertion operations by referring to the example represented in Fig. 4: Fig. 4a shows the initial state of the data structure including four values, Fig. 4b shows the same data structure after the insertion of a duplicate value, and Fig. 4c after the insertion of many other values. Within the figure, we denote as $\mathsf{c}_n^{\mathsf{v}}$ a semantically secure ciphertext included within the data structure at "version" v, and id_t the record identifier inserted via the t^{th} insertion. We also denote as $\mathsf{L0}$ the list at *leaf level* and as $\mathsf{L1}, \ldots, \mathsf{L3}$ the lists at *non-leaf levels*. All the *arrays* are stored in *disk pages* of size $B = 3$. Disk pages that belong to the same *internal array* or the same *leaf node* are doubly-linked and drawn as attached to each other in the figure (to avoid overcrowding the figure, we do not draw arrows in this case).

At the beginning ($\mathsf{v} = 1$, Fig. 4a), where we consider a dataset without duplicate values (plaintext values are $\langle 1, 2, 4, 6 \rangle$), $\mathsf{L0}$ is stored in a single *leaf node*, that spans across two disk pages, and $\mathsf{L1}$ is stored in a single *internal array* (stored in a single disk page). The leaf node is composed of a single *leaf array*. The leaf array starts with a value promoted at level 1 (c_1^1), stores all the values of $\mathsf{L0}$ in four slots, and contains one



(a) Initial index without duplicates (v = 1)

(b) Index after insertion of a duplicate value (v = 2)

(c) Index with all the values (v = 3)

Fig. 4. Sequence of insertions within our *DHE skip list* used for indexing a queryable-encrypted database ($B = 3$ and $p = \frac{1}{2}$).

extra empty slot. The data structure can be built by using the same insertion algorithm of HI-EM skip list, with the exception that all pages are doubly linked.

When a duplicate value is inserted into the data structure (see Fig. 4b, where ciphertext c_4^2 corresponding to plaintext 6 is associated with record identifier id_5), the insertion algorithm uses the lists at non-leaf levels to access the list at leaf level, scanning right until a slot associated with a value smaller than that associated with the value to be inserted is found (as the original skip list insertion algorithm). In Fig. 4a, access c_1^1 at L1, descend to c_1^1 at L0, scan until c_4^1, insert the new value at position $n = 4$ (shifting the existing value associated with id_4 to position $n = 5$), flip a coin, and assume that the new value is promoted to L1. The insertion algorithm then splits the existing leaf array into two and creates a second leaf array that starts with the new value. This insertion (v = 2, Fig. 4b) does not require the creation of a new leaf node because the new value is not promoted to a level ≥ 2 (see Section 4.1). Since the ciphertexts stored in the DHE skip list are semantically secure, even if the underlying plaintext associated with id_5 and id_4 is the same (i.e., value 6 is duplicate), the two corresponding ciphertexts c_4^2 and c_5^2 are different.

Following the same procedure, in Fig. 4c (v = 3) we also insert values $\langle 8, 9, 10, 13, 7, 13, 20, 13, 23\rangle$ in this order, and assume that $\langle 8, 7, 13\rangle$, $\langle 20\rangle$, and $\langle 10\rangle$ are promoted to L1, L2 and L3, respectively.

Now we consider searching value plaintext = 13 within the index at version v = 3. The search algorithm starts at L3, descends to c_9^3 first at L2 and then at L1, because the result of the encrypted comparison with c_{13}^3 tells the algorithm not to continue the search at L2 (20 is greater than 13). At L1, comparison with c_{11}^3 returns *"equal to"*, thus the algorithm descends to L0 at position $n = 11$. Finally, the algorithm scans L0 both backward and forward until different values are found (i.e., output of the encrypted comparison $\neq$ *"equal to"*). Backward scan finds c_{10}^3 and stops at c_9^3, and forward scan finds c_{12}^3 and stops at c_{13}^3.

Without adopting a *doubly-linked list* at the leaf level, the search algorithm would not be able to operate the backward scan. An alternative may involve descending at previous values at some upper level, however such an approach would be more inefficient and lead to longer forward scan operations.

When a leaf array needs resizing, some values must be shifted within the same or a different disk page. Note that a new page may be allocated to accommodate the increased size of the entire leaf node. To maintain a consistent state, the relocation of a skip list value requires the index to update all the pointers that reference that value both in the disk page that stores the same value at the level below, and in the disk page that stores the same value at the level above. This operation is done efficiently thanks to backward pointers from a value in a slot to its corresponding promoted value in a slot at level above.

4.3. Security guarantees of the proposed DHE skip list

DHE skip list prevents leakage related to the disclosure of duplicate plaintext values, and related to statistical information on the plaintext distribution and the history of transactions that led to the current state of the index. Moreover, it is history-independent because it complies with the same principles used in the original HI-EM skip list, discussed in Section 4.1 (also see [13, Section 6.3] for the exact principles). Both leaf and non-leaf nodes of the DHE skip list nodes include IND-CPA secure ciphertexts, which do not leak any information about plaintext records. Thus, the only information which could leak is due to the topology of the data structure. At leaf level, the DHE skip list is a simple ordered list, thus a passive offline attacker that obtains a copy of the index can only learn the relative order of stored records (i.e., can determine which encrypted values are greater than others, but does not learn the corresponding plaintext values, nor if two encrypted values subsequently stored in the index are equal or not). Such a leakage cannot

be prevented if clients are stateless, the query protocol involves only a single round of communication, and sublinear query times are needed (see Section 3.3). Moreover, all non-leaf nodes are part of superimposed sparse lists generated with probabilistic algorithms which are independent of the indexed values, thus the attacker learns nothing from them. Inference attacks that are only based on sorting order [28] (also known as leakage-abuse attacks) require substantial background knowledge regarding the distribution of the underlying plaintext values, and do not perform well when the dataset distribution is close to uniform. The most simple approach to also remove such leakage and achieve complete semantic security against offline attackers would require to encrypt record identifiers stored in the index, and to adopt a two-round query protocol which lets the client decrypt such identifiers at its side. This type of approach is not new in the literature, see e.g., Poddar et al. [34], and is orthogonal to our design (see Section 8). Note that even in that case, the design of our DHE skip list still contributes to security because using standard techniques may leak information anyway.

As a result, the proposed DHE skip list achieves the minimum leakage among known practical indexing data structures in the literature for stateless clients and single-round query protocols with sublinear times.

5. Implementation for LW-ORE and PostgreSQL

We implement the proposed DHE skip list for Lewi-Wu Order Revealing Encryption (LW-ORE, see Section 2.4) as a PostgreSQL (version 16) extension. We selected the state-of-the-art ORE scheme by Lewi and Wu [8], which is specialized on efficient range queries for numerical data, but also supports other derived operations, such as sorting and substring prefix/suffix matching, making it particularly well-suited for database applications. Moreover, we selected PostgreSQL because it is a very popular open-source DBMS and offers standard APIs to extend its data types, operators, and indexes. Our implementation includes three main components:

- the implementation of the DHE index is a PostgreSQL C extension, which is not specific for LW-ORE, but can be used for any data types which defines a proper comparison operator. This detail allows us to compare our proposal with the pre-existing B-Tree implementation within PostgreSQL even with plaintext numeric data (see Section 7). We detail the page layout of the DHE index in PostgreSQL in Section 5.1;
- the implementation of custom data types and related operators for supporting LW-ORE (including S-RND encryption scheme for efficient decryption, as described in Section 3.1). To this aim, we modify an open-source extension for PostgreSQL developed by The Enquo Project [35] called PG_Enquo, developed with the Rust programming language. Details are given in Section 5.2;
- the implementation of a PostgreSQL-compatible C custom function which is invoked by a trigger at data insertion time. This component is used to selectively delete LW-ORE ciphertexts after usage. Details are given in Section 5.3.

As a minor note, we also implement a Rust client to perform the experimental evaluation which uses our modified version of the PG_Enquo cryptographic libraries (see Section 7).

5.1. Page layout of the DHE index

PostgreSQL organizes its data in (disk) pages of a fixed size, by default 8KB in version 16,[3] and every table and index is stored as an array of pages. Each page is an abstraction layered on top of a disk block (simple unformatted unit of I/O), and has a predefined layout consisting of a page header, including metadata such as the page size and checksum, and an optional *special section*, which is only used for data allocated by indexes. As an example, the B-Tree index, implemented by default in PostgreSQL, uses the special section to store pointers to left and right sibling pages and other information that is relevant to manage the index. The remaining free space within each page can be used to store actual data, and its organization is decided accordingly. As an example, table pages store record tuples and organize the free space according to data types, and index pages store index entries and record identifiers (formally called tuple identifiers). PostgreSQL extensions can use the same storage and buffer managers also used by native indexes to access and manage the whole of pages.

We use PostgreSQL pages to store *disk pages* as defined in Section 4 (which we denote here as *skip pages* to avoid ambiguities), as well as an additional page, that we denote as *metadata page*, which stores the index parameters, including the current maximum level of the skip list, the pointers to the first pages for each level, and the size of the data type that is being indexed. The metadata page is allocated at index creation for each DHE index. Both the metadata and skip pages inherit the same page header of a standard PostgreSQL page, and use the unallocated free space to store their data, but only the skip pages use the PostgreSQL page's final special section. The special section of each skip page includes information needed by the index to link pages and access the stored slots, namely the number of indexed values on that page, a forward pointer to the next page in the array (if any), a backward pointer to the previous page in the array (if any), and an identifier used to distinguish if the next page is part of the same array or not (both if the array is a *leaf node* or an *internal array*).

A copy of a record attribute that needs to be indexed is maintained within a skip page of the DHE skip list, and is organized as a skip list slot (as described in Section 4) by adding a pointer to the same value in a page that is part of a list at the level below (if any), a pointer to the same value in a page that is part of a list at the level above (if promoted), and some flags used to link the slot with its corresponding *leaf array* (these flags are used only if the slot is in a *leaf node* page).

5.2. Custom data types and operators

We inherit the custom data type for LW-ORE encryption already defined by PG_Enquo, which implements the types of encrypted data described in Section 3.1 as follows:

- S-RND encryption is implemented by using AES-GCM-SIV and by generating a random nonce for each encrypted value;
- TOKEN and Q-RND encryption use LW-ORE *left* and *right* encryption functions based on AES as PRF (respectively);

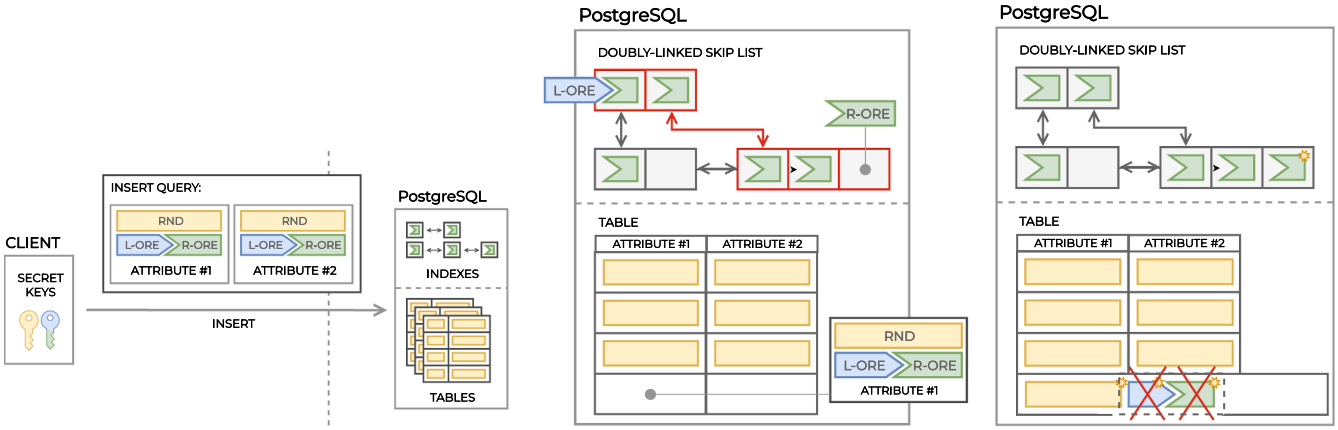
In the rest of the section, we denote as RND the S-RND ciphertext, as L-ORE (LW-ORE left ciphertext) the TOKEN ciphertext, and as R-ORE (LW-ORE right ciphertext) the Q-RND ciphertext.

We observe that PG_Enquo offers two operation modes:

- the default and more secure mode only generates RND and R-ORE ciphertexts at insertion time to prevent attacks against deterministic L-ORE ciphertexts. L-ORE ciphertexts are only sent as query tokens for retrieving encrypted data. This mode is not able to use PostgreSQL indexes, and all queries require linear scans over all LW-ORE records;
- the "reduced security mode", which generates RND, R-ORE and L-ORE at insertion time, stores all of them persistently within the database, and allows the use of B-Tree or Hash indexes for efficiency. As also described by the authors of the project [36], this mode opens to frequency attacks as already described in Section 2.1.

To take advantage of the main benefits of LW-ORE and comply with our proposed architecture, we modify the extension to securely store and index encrypted records without the drawbacks of the "reduced security mode".

[3] See PostgreSQL documentation, Chap. 65, Database Physical Storage, Database Page Layout (https://www.postgresql.org/docs/16/storage-page-layout.html)



(a) The client sends the encrypted record to PostgreSQL

(b) The LW-ORE left ciphertext is used to traverse the *doubly-linked skip list* index. The LW-ORE right ciphertext is stored inside the index. The whole encrypted attribute is stored inside the database table

(c) The LW-ORE left and right ciphertexts are removed from all encrypted attributes just inserted in the table

Fig. 5. Sequence of operations for inserting a new encrypted record with two attributes.

We also applied an additional modification related to storage overhead of LW-ORE encrypted data. Indeed, note that one of the main challenges when dealing with queryable encryption is ciphertext expansion [37]. Remind that, other than costs due to additional storage, such overhead possibly affect performance as a whole, including throughput and response times. This is especially true when considering databases with a large number of records and complex indexing strategies. The ciphertexts generated by the LW-ORE scheme are larger than the corresponding plaintext values, however how encrypted data are encoded may also further increase storage overhead. In this regard, recall from [8, Section 3.1] that right encryption $\mathsf{ORE.Encrypt}_R$ computes the comparison between the value to encrypt and all the possible plaintext values that can be represented in the data domain, where each comparison CMP outputs either -1, 0 or 1. PG_Enquo encodes each comparison result by using 2 bits ($0b00$, $0b01$ and $0b10$ to encode -1, 0 and 1, respectively), thus increasing by $\frac{4}{3}$ the size of the ciphertext.

We decide to use a more efficient encoding technique which packs 5 comparisons in a single byte through a base conversion (a five digits base-3 number to a base-2 number, and vice versa). This allows reducing the size of R-ORE ciphertexts by almost 20%, with regard to those stored by the original PG_Enquo.

5.3. Information flow

We refer to Fig. 5 to better explain the information flow of our implementation. The figure shows the sequence of operations made by the client and the DBMS to insert a record with two encrypted attributes. In Fig. 5a, the client generates RND, L-ORE and R-ORE ciphertexts and sends them to the DBMS via unmodified PostgreSQL APIs. In Fig. 5b, PostgreSQL forwards the whole encrypted record to the DHE skip list index, which uses the L-ORE ciphertext to perform ORE.Compare with all the R-ORE ciphertexts already indexed and find the correct insert position. The crucial part of our implementation is that the L-ORE ciphertext is not stored in the index: before saving encrypted record in the index page, our implementation deletes both the RND and L-ORE ciphertexts from each encrypted attribute, so that the new index slot only contains R-ORE. The whole encrypted record is then also stored in the table. Finally, in Fig. 5c we show that before concluding the insert operation, PostgreSQL needs to delete both L-ORE and R-ORE ciphertexts from the database table. Our implementation uses a trigger function applied "AFTER INSERT" to remove L-ORE and R-ORE from the newly inserted record. We did not find a more elegant way to let PostgreSQL forward only temporary values to an index without storing them in tables, or to selectively forward different portions of the same data to indexes and tables. An alternative may be to modify internal routines of PostgreSQL. However, other than adding a substantial amount of effort, this strategy would also have drawbacks on the long-term maintainability of the implementation. Nonetheless, we consider such direction as an interesting future work.

We omit details of queries for data retrieval because they do not have challenges related to selective ciphertext deletion, and thus it is possible to just refer to the description proposed in Section 3.2.

5.4. Security considerations

All the records stored in PostgreSQL tables and indexes are encrypted with semantically secure encryption schemes. Then, if a passive offline attacker obtains a copy of the database, it cannot infer which encrypted records are equal. This prevents inference attacks that are possible when using encryption at rest or deterministic encryption schemes. Data values (and attributes) are protected also during the query procedure. The PostgreSQL server only learns the statement of the query, accessed index entries, and encrypted result values. Due to the deterministic property of LW-ORE left ciphertexts, it also learns the repeated queries on the same attribute. Such information is also available to a passive online attacker that has full access to the DBMS instance.

Real passive offline attackers may also obtain diagnostic tables and logs in the event of, e.g., "smash-and-grab" attacks that can steal a (full-state) snapshot of a virtual machine or full-system compromises that involve rooting the DBMS and gaining full access to persistent and volatile OS and DBMS state [38]. To reduce the chance that a passive offline attacker gets some of the information an online attacker would access we disable query logs and design our implementation to discard the L-ORE ciphertext as soon as possible, which is one of the peculiarities of our design.

5.5. Limitations of our implementation

The current implementation of the DHE skip list for LW-ORE in PostgreSQL serves as a proof-of-concept prototype designed to validate the feasibility and practicality of the proposed design in a real-world setting. The prototype implementation has not yet been optimized for performance and does not support concurrent write operations. We acknowledge these limitations and leave further engineering efforts as future work.

6. Formal security analysis of minimal leakage

We model offline security of indexing data structures through a variant of the simulation-based notion [39] of *indistinguishability with leakage* introduced by [10] and also used by LW-ORE [8]. Intuitively, the original notion models encryption schemes leaking information about some property existing over plaintexts. We model an indexing scheme operating through knowledge of an *input leakage* which produces a data structure leaking information characterized by an *output leakage* when accessed offline. The model is general enough for being applied to many types of leakage functions, but for concreteness we consider that the input leakage is ORE minimal leakage, that is, *strict global ordering* of plaintexts, namely the leakage function outputs whether plaintexts are smaller (<), equal (=) or greater (>) with regard to each other. Instead, the output leakage is *non-strict global order*, namely whether any two plaintexts are either smaller or equal (≤), or greater or equal (≥) with respect to each other. Given this novel security model, we show that proving that the proposed data structure is secure under statistical assumptions is quite easy thanks to probabilistic operations, which are independent of the contents of the indexed values, and the absence of re-balancing operations, which makes it easier to comply to a subtle constraint of the formal model. We propose details in the following.

Definition 1 (Indexing scheme based on leakage)**.** An indexing scheme based on leakage is a set of algorithms $\langle \mathsf{Setup}, \mathsf{Add}, \mathsf{Comp}\rangle$, where $\mathsf{idx} \leftarrow \mathsf{Setup}(1^\lambda)$ initializes the indexing data structure idx, $\mathsf{idx}' \leftarrow \$\mathsf{Add}(\mathsf{idx}, \mathcal{L}_{\mathsf{IN}}(\cdot))$ updates index idx into idx' by using leakage information given by input leakage function $\mathcal{L}_{\mathsf{IN}}$ computed over all indexed values, and $\mathcal{L}_{\mathsf{OUT}}(i, j, \ldots) \leftarrow \mathsf{Comp}(\mathsf{idx}, i, j, \ldots)$ returns the output of some type of comparison between the ith,jth,…, values inserted in the index with Add.

Remark 1 (Reducing indexing to indexing based on leakage)**.** A typical indexing scheme, which accepts data as input and not leakage, can be easily reduced to indexing based on leakage as of Definition 1 by first executing some comparison function and then using the output as the *input leakage*.

Definition 2 (Offline security of indexing based on leakage)**.** let $\Upsilon := \langle \mathsf{Setup}, \mathsf{Add}, \mathsf{Comp}\rangle$ be an indexing scheme as for Definition 1, let $\mathcal{A} = \langle \mathcal{A}_0, \ldots, \mathcal{A}_q\rangle$ be an adversary for some $q = \mathsf{poly}(n)$, let $\mathcal{S} = \langle \mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_q\rangle$ be a simulator, and let $\mathcal{L}_{\mathsf{IN}}(\cdot)$ and $\mathcal{L}_{\mathsf{OUT}}(\cdot)$ be *input* and *output leakage functions*. We define experiments $\mathsf{REAL}^{\Upsilon}_{\mathcal{A},\mathcal{L}_{\mathsf{IN}}}(\lambda)$ and $\mathsf{SIM}^{\Upsilon}_{\mathcal{A},\mathcal{S},\mathcal{L}_{\mathsf{OUT}}}(\lambda)$ as:

$\mathsf{REAL}^{\Upsilon}_{\mathcal{A},\mathcal{L}_{\mathsf{IN}}}(\lambda)$

$\mathsf{idx}_0 \leftarrow \Upsilon.\mathsf{Setup}(1^\lambda)$
$\langle \mathsf{st}_{\mathcal{A}}\rangle \leftarrow\$\ \mathcal{A}_0(1^\lambda)$
for $i \in 1, \ldots, q$:
  $\langle \mathsf{m}_i, \mathsf{st}_{\mathcal{A}}\rangle \leftarrow\$\ \mathcal{A}_i(\mathsf{st}_{\mathcal{A}}, \mathsf{idx}_{i-1})$
  $\mathsf{idx}_i \leftarrow\$\ \Upsilon.\mathsf{Add}(\mathsf{idx}_{i-1}, \mathcal{L}_{\mathsf{IN}}(\mathsf{m}_1, \ldots, \mathsf{m}_i))$
return $\langle \mathsf{idx}_0, \ldots, \mathsf{idx}_q\rangle, \mathsf{st}_{\mathcal{A}}$

$\mathsf{SIM}^{\Upsilon}_{\mathcal{A},\mathcal{S},\mathcal{L}_{\mathsf{OUT}}}(\lambda)$

$\langle \mathsf{idx}_0, \mathsf{st}_{\mathcal{S}}\rangle \leftarrow\$\ \mathcal{S}_0(1^\lambda)$
$\langle \mathsf{st}_{\mathcal{A}}\rangle \leftarrow\$\ \mathcal{A}_0(1^\lambda)$
for $i \in 1, \ldots, q$:
  $\langle \mathsf{m}_i, \mathsf{st}_{\mathcal{A}}\rangle \leftarrow\$\ \mathcal{A}_i(\mathsf{st}_{\mathcal{A}}, \mathsf{idx}_{i-1})$
  $\langle \mathsf{idx}_i, \mathsf{st}_{\mathcal{S}}\rangle \leftarrow\$\ \mathcal{S}_i(\mathsf{st}_{\mathcal{S}}, \mathsf{idx}_{i-1}, \mathcal{L}_{\mathsf{OUT}}(\mathsf{m}_1, \ldots, \mathsf{m}_i))$
return $\langle \mathsf{idx}_0, \ldots, \mathsf{idx}_q\rangle, \mathsf{st}_{\mathcal{A}}$

Indexing scheme Υ is secure with output leakage $\mathcal{L}_{\mathsf{OUT}}$ if the outputs of $\mathsf{REAL}^{\Upsilon}_{\mathcal{A},\mathcal{L}_{\mathsf{IN}}}(\lambda)$ and $\mathsf{SIM}^{\Upsilon}_{\mathcal{A},\mathcal{S},\mathcal{L}_{\mathsf{OUT}}}(\lambda)$ are statistically or computationally indistinguishable.

Definition 3 (Minimal leakage for ORE [8,10])**.** the strongest notion of simulation-based security for ORE is security with respect to the leakage function that only reveals the comparison (CMP) between plaintexts:

$$\mathcal{L}_{\mathsf{CMP}}(\mathsf{m}_1, \ldots, \mathsf{m}_t) = \{\langle i, j, \mathsf{CMP}(\mathsf{m}_i, \mathsf{m}_j)\rangle \mid 1 \le i < j \le t\},$$

where:

$$\mathsf{CMP}(\mathsf{m}_0, \mathsf{m}_1) = \begin{cases} 1 & \text{if } \mathsf{m}_0 > \mathsf{m}_1 \\ 0 & \text{if } \mathsf{m}_0 = \mathsf{m}_1 \\ -1 & \text{if } \mathsf{m}_0 < \mathsf{m}_1 \end{cases}$$

Definition 4 (Minimal leakage for indexing based on sorting)**.** the strongest notion of simulation-based security as of Definition 2 is security with respect to the leakage function that only reveals the Non-Strict Inequality (NSI) between plaintexts:

$$\mathcal{L}_{\mathsf{NSI}}(\mathsf{m}_1, \ldots, \mathsf{m}_t) = \{\langle i, j, \mathsf{NSI}(\mathsf{m}_i, \mathsf{m}_j)\rangle \mid 1 \le i < j \le t\},$$

where:

$$\mathsf{NSI}(\mathsf{m}_0, \mathsf{m}_1) = \begin{cases} 1 & \text{if } \mathsf{m}_0 \ge \mathsf{m}_1 \\ 0 & \text{if } \mathsf{m}_0 \le \mathsf{m}_1 \end{cases}$$

Remark 2 (Constraining non-strict inequality comparisons)**.** We highlight a subtle difference in constraints included in Remarks 3 and 4 which may otherwise go unnoticed. In both definitions, constraining the execution of the comparison function only to inputs $\mathsf{m}_i, \mathsf{m}_j : i < j$ is meant to prevent execution of the function by using the same inputs in different order. However, for strict ordering (Definition 3) this is a technicality meant to prevent inclusion of redundant information, because knowledge of $\mathsf{CMP}(\mathsf{m}_i, \mathsf{m}_j)$ also implies knowledge of $\mathsf{CMP}(\mathsf{m}_j, \mathsf{m}_i)$. Instead, for non-strict inequality (Definition 4), this is paramount to avoid leaking additional information than what is meant by the model. Indeed, it is obvious that leaking both $\mathsf{NSI}(\mathsf{m}_i, \mathsf{m}_j)$ and $\mathsf{NSI}(\mathsf{m}_j, \mathsf{m}_i)$ is equivalent to leak $\mathsf{CMP}(\mathsf{m}_i, \mathsf{m}_j)$.

Theorem 1. *the DHE skip list (Section 4.2) is a statistically secure indexing schemes Υ as of Definition 2 with input leakage $\mathcal{L}_{\mathsf{IN}} = \mathcal{L}_{\mathsf{CMP}}$ (Definition 3) and output leakage $\mathcal{L}_{\mathsf{OUT}} = \mathcal{L}_{\mathsf{NSI}}$ (Definition 4).*

Proof sketch of Theorem 1. DHE skip lists are organized in three major components: sparse lists at non-leaf levels including promoted values, the list at leaf level including sorted values, and memory allocation strategies for better organize data included in both previous types of lists in external memory. Since sparse lists and allocated memory are picked randomly independently of the inserted values, it is trivial to construct a simulator which is able to build sparse lists with the same probability distribution of the actual algorithm (indeed, they can implement the very same algorithm). For the leaf level, we show a simulator algorithm in Algorithm 1. First, note that the algorithm always access leakage information $L[\langle i, j\rangle]$ complying with Definition 4, because j is always greater than i. At the first execution ($j = 1$), index idx_0 is empty, thus the returned index idx' only includes 1. At the jth insertions ($j \ge 2$), the index idx_{j-1} includes indexes of elements from 0 to $j - 1$ sorted in ascending order with regard to their referenced values, based on the leakage received at the insertion operation. Note that indexes which reference equal values are sorted in reverse order with regard to their insertion index j, because they are inserted as soon as their referenced value is greater or equal (and thus, the index which precedes them in the list reference a value which is strictly smaller). As a result, statistical indistinguishability holds if the real indexing algorithm produces the same list deterministically, given the same insertion history, as already described in Section 4.2 and Fig. 4. □

Remark 3 (Indexing with reduced leakage)**.** An alternative way to prove security of an indexing scheme may involve ensuring that the insertion algorithm never uses greater information leakage than that allowed by $\mathcal{L}_{\mathsf{OUT}}$, implying $\mathcal{L}_{\mathsf{IN}} = \mathcal{L}_{\mathsf{OUT}}$ in Definition 2, thus forcing the algorithm to behave like a simulator. However, it may be difficult to assess that an implemented algorithm never breaks Definition 2 (e.g., for efficiency, an algorithm may operate an equality comparison or operate

Algorithm 1 Insertion algorithm of simulator $\mathcal{S}_j (j \geq 1)$.

$\mathcal{S}_j(\mathsf{st}_{\mathcal{S}}, \mathsf{idx}_{j-1} = \langle i \rangle_{i \in [j-1]}, L = \mathcal{L}_{\mathsf{NSI}}(\mathsf{m}_1, \ldots, \mathsf{m}_{j-1}))$

```
1 : pos := 0
2 : for i in idx_{j-1} then
3 :   if L[⟨i, j⟩] = 1 then   / check if x_i ≥ x_j
4 :     idx' ← idx.insert(pos, j)   / insert j at position pos
5 :     return ⟨st_S, idx'⟩
6 :   pos ← pos + 1
7 : idx' ← idx.insert(pos, j)   / insert j as the last element
8 : return ⟨st_S, idx'⟩
```

a comparison which involves $j > i$), thus showing that the algorithm behaves like the simulator constructed above may be preferable and more reliable.

Remark 4. (indistinguishability and history independence) Interestingly, the list in Algorithm 1 is generated deterministically given a certain insertion order. At first glance, this may seem to violate history-independence guarantees. In particular, if an adversary is able to distinguish which values are equal to each other, then it could infer their insertion order with probability one. However, an adversary that does not know the plaintexts stored in the database a priori is not able to identify these equalities and thus can not leverage this information to break history-independence.

7. Performance evaluation

We present results of our performance evaluation with the PostgreSQL prototype implementation described in Section 5.

Testbed. We used two identical machines for the experiments acting as client and database server. Both machines are equipped with an Intel Xeon E5-2650 @ 2.80GHz (AES-NI) CPU, 32GB of ECC DIMM DDR3 RAM @ 1600MHz (8 × 4GB Samsung M393B5270DH0), and a 480GB Kingston SA400S3 SATA SSD, and run Debian 12. The server machine uses PostgreSQL 16, and the client machine uses a custom client application written in Rust with benchmark code. The two machines are connected via a 1Gbps Ethernet link to the same network switch. We configured an 8-bit block size for LW-ORE with AES-256 as PRF and use AES-GCM-SIV with AES-256 for RND (with random nonce). The PostgreSQL DBMS instance is configured with the default settings. In total, the code of the C extension implementing the DHE skip list consists of 2500 lines of code, and the Rust client consists of 200 lines of code.

The evaluation focuses on the performance of the DHE skip list when used to index LW-ORE, but we also show some results for plaintext data types to provide a reference for the data structure as a stand alone index and to show that our implementation does not focus exclusively on ORE encrypted data. When the DHE skip list is used to index plaintext data types, both our C extension and our client Rust implementation do not perform any encryption operations, and PostgreSQL uses standard comparison operators for the data type being indexed. The metrics analyzed in our experiments are the storage overhead for an encrypted attribute, both in the database table and in the index, and the end-to-end (E2E) latency experienced by the client when performing insertions and queries on the database. Since we are considering a stateless, single-round query protocol, the network overhead is only due to the size of L-ORE ciphertexts sent within query clauses and of ciphertext expansion of encrypted data during data retrieval and insertion. In our testbed, the size of each L-ORE ciphertext is 77 B. We describe ciphertext expansion in detail within *storage overhead* discussion below.

Note that, while our current implementation of the DHE skip list supports both LW-ORE and standard PostgreSQL plaintext data types, it requires the deployment of two dedicated extensions specifically configured and compiled for each family of data types. This is due to the need of selectively deleting portions of data only when dealing with LW-ORE encryption.

Table 1
Size of a single encrypted attribute (64-bit integer).

	Skip List	BTree
Attribute size within table	60 Bytes	643 Bytes
Attribute size within index	506 Bytes	~665 Bytes

Table 2
Size of the entire indexing data structure (DB size: 100k encrypted records).

	Size of the index
Skip List	~127MBytes
BTree	110MBytes

Storage overhead. We first analyze the size of an encrypted attribute when stored in a database table and in the index. Table 1 shows the size of a single encrypted attribute (64-bit integer) when PostgreSQL adopts our DHE skip list (*Skip List* column) or the default B-Tree index (*BTree* column). When using our DHE skip list, PostgreSQL only stores the R-ORE ciphertext in the index, taking about 506 B, and the RND ciphertext (AES-GCM-SIV ciphertext and tag plus nonce) in the table, taking about 60 B. Note that the ideal size in the table for this type of encryption would be 36 B (8 B ciphertext, 12 B nonce, and 16 B tag), however the encrypted data in our implementation also embeds metadata regarding data type and other information for correctly decoding after decryption, and all these data are encoded using CBOR. When using the standard B-Tree to index encrypted records, all components of an encrypted attribute (RND, L-ORE, R-ORE) are stored both in the index and in the table, taking about 643 B and 665 B, respectively. Also, the space overhead of a single indexed attribute may vary due to alignment padding in the PostgreSQL B-Tree implementation (see Section 5.1). As a result, for a single attribute our implementation shows smaller storage overhead for both tables and indexes.

Now we analyze the size of the entire indexing data structure for a database including $100k$ encrypted records (we omit table size because it just increases linearly with the number of records). Table 2 shows the size of the entire index when PostgreSQL adopts our DHE skip list (*Skip List* row) or the default B-Tree index (*BTree* row). The sizes for the DHE skip list and for the standard B-Tree are about 127 MB[4] and 110 MB, respectively, thus making the DHE skip list larger than the B-Tree, in opposition to the result obtained for a single record. The main reason is that the DHE skip list maintains additional free gaps in the list at leaf-level, and stores multiple copies of the same index value when promoted to higher levels (i.e., one for each page to which it is promoted).

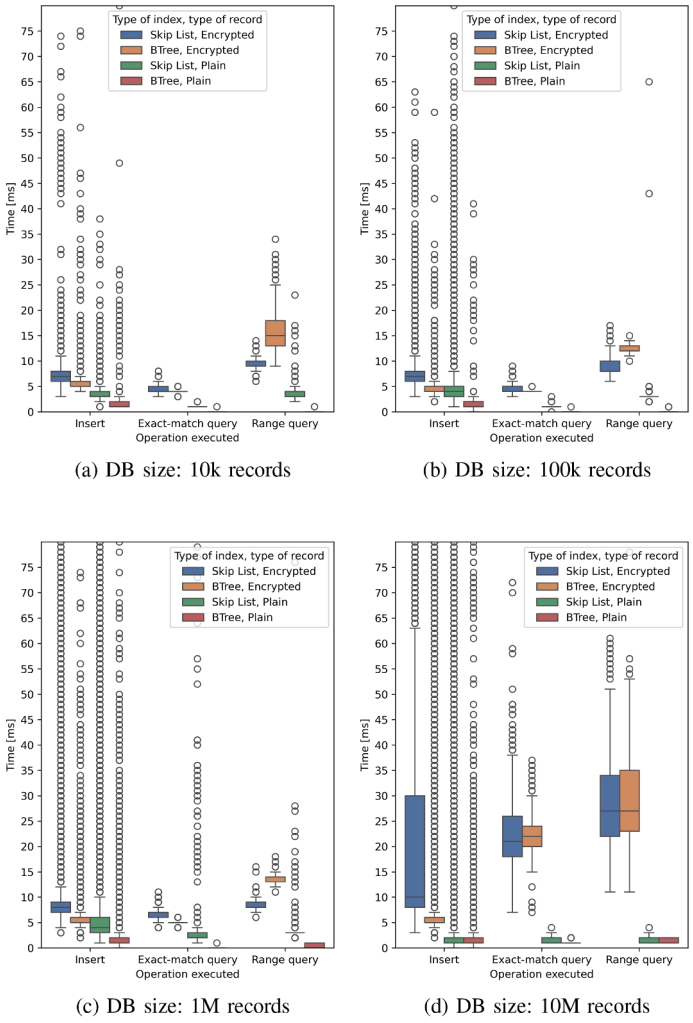
End-to-end latencies. We design a set of micro-benchmarks which measure latency experienced by the client to perform insertion operations, single-match queries, and range queries. Measured latencies include the total time required by the client to construct the query (encryption included, when present) and send it over the network, and by PostgreSQL to process the requested operation and send back the response. Fig. 6 shows the results of the first set of micro-benchmarks. We compare the performance of the DHE skip list with the default B-Tree index in PostgreSQL, and we run our tests with both encrypted and plaintext 64-bit integer data types. We measure latencies for a single attribute. In Fig. 6a, we show the box plot representing $10k$ measures related to insertion operations into the database (that is, the database starts empty, and after the execution it stores $10k$ records). The figure also shows box plots for exact-match and range queries on the database with $10k$ records. Range queries are designed to return about 50 records,

[4] At each execution, the space overhead of the DHE skip list may change due to the probabilistic algorithms which govern the number of empty slots left at leaf level and the number of promoted values. However, it does not change greatly, and the proposed value is representative of the expected median value.

but the exact amount of records may slightly vary due to the probabilistic nature of the benchmark. In Fig. 6b–d we repeat the same tests on databases with $100k$, $1M$ and $10M$ records, respectively. The results show that the DHE skip list achieves better performance than the B-Tree for the range query operation when the two data structures are used to index encrypted records. The difference in performance between the two indexes is almost the same for all the four database sizes. On the other hand, the B-Tree index remains faster than the DHE skip list when the indexed records are plaintext. We assume that, since we developed the DHE skip list to be specialized for storing encrypted records, this difference in performance is due to the optimizations that our implementation makes to handle records that are bigger in size with respect to the plaintext equivalent. When inserting an encrypted record or performing an exact-match query, for all the four database sizes, the DHE skip list is slower than the B-Tree index. We also observe that the performance of the DHE skip list seems to scale well for an increasing number of records, although it can be observed that the number of outliers increases. The DHE skip list has significantly larger variance than the B-Tree, and shows worse performance for insertion operations in case of very large databases comprising 10M records. Both results are probably due to the complexity of managing history independence, which involves frequent page splits (and merges) and array resizes. We leave further investigations and potential improvements to the implementation as future work. To better show the performance of the DHE skip list over the entire database creation, we show additional line plots in Fig. 7. We use the same data points of the previous tests (Fig. 6), but we plot them incrementally to show how the latency evolves when the database size increases. Fig. 7a and c show the end-to-end latency measured when inserting encrypted records in a database that uses the DHE skip list, from empty to $100k$ records and from empty to $10M$ records, respectively. The plots also show the 95th and 99th percentile computed over a sliding window of the previous 5000 insertions. Fig. 7b and d show the same plots for the PostgreSQL B-Tree index. In Fig. 7b the 95th percentile is represented by a dashed line due to its overlap with the 99th percentile line. Results show that, despite some outliers, probably due to promotions and page splits (and merges), the latency required to insert a record in the DHE skip list remains almost stable as the database size increases, for both database sizes considered. It is possible to observe a similar trend for the B-Tree index, and, although our implementation is not as optimized as the B-Tree implementation in PostgreSQL, the DHE skip list achieves acceptable performance up to $10M$ records.

We now consider end-to-end latency of range queries. We note that the query time may highly depend on the number of records that are returned and, especially for the DHE skip list, on the number of duplicate values stored within the index. Thus, we design an experiment accordingly. In Fig. 8a, we measure the end-to-end latency required to perform a range query on a database with $100k$ records while varying the number of records returned by the range query. We analyze the performance of the DHE skip list and the PostgreSQL B-Tree index when the result of a range query returns 100, 1000, and 10000 records. As for the previous tests, we measure latencies for operations related to both encrypted and plaintext 64-bit integer data types. In the figure, we use the notation “Skip List” to refer to the DHE skip list and “B-Tree” to refer to the PostgreSQL B-Tree. We add suffix “Enc” to specify that the data type is encrypted, and “Plain” to specify that the data type is plaintext. Results show that the latency experienced when executing a range query on the DHE skip list slightly increases as the number of records returned by the query increases. Also, we obtained a similar outcome as in the previous tests: the DHE skip list is faster than the B-Tree index when the indexed records are encrypted, and it achieves results that are comparable with the B-Tree index when the indexed records are plaintext.

Another microbenchmark is shown in Fig. 8b. We measure the end-to-end latency required to insert $10k$ additional encrypted records into a database already populated with $1M$ records indexed with the DHE skip list. Results are consistent with the previous tests, and the index



(a) DB size: 10k records (b) DB size: 100k records

(c) DB size: 1M records (d) DB size: 10M records

Fig. 6. E2E latency for insertion, exact-match query, and range query for both encrypted and plaintext 64-bit integers. Comparison between the DHE skip list and the PostgreSQL B-Tree used as indexes.

achieves almost the same performance when compared with an empty database.

An additional set of experiments is shown in Fig. 9. We analyze the impact of the promotion probability p on the performance of the DHE skip list when indexing encrypted 64-bit integer data types. We vary the value of $p = 1/B^{\gamma}$ by changing the parameter γ (see Section 4), and we measure the end-to-end latency required to perform insertion operations and range queries on a database with $100k$ records. The results show that it is difficult to identify a clear trend by only plotting the two extreme values of γ from the allowed range $0.5 < \gamma \leq 1 - \log(\log(B))/\log(B) = 0.63$, thus we also plot the performance for $\gamma = 0.96$. As expected, increasing the promotion probability p (i.e., decreasing γ) slightly improves the performance of insertion operations, while it worsens the performance of range queries. A higher promotion probability makes the search operations required by the insertion algorithm faster, but increases the number of pages at leaf level (due to more frequent splits and merges), thus increasing the latency of range queries (due to an increased number of pages accessed). When lowering the promotion probability p, the opposite effect is observed: insertion operations become slower, while range queries become faster.

The last set of experiments is shown in Fig. 10. We compare the performance of our DHE skip list with a modified version of the original HI-EM skip list by Bender et al. [13] to evaluate the effectiveness of backward pointers in index search. The *modified HI-EM skip list* is adapted

from that of Bender et al. [13] to support duplicate values without leakage, but only uses only forward pointers between slots and pages (see Section 4), such that the search algorithm stops the search at each level until a value greater than or equal to the searched value is found, instead of taking advantage of a direct path to leaf level when an exact match is found. Intuitively, this may cause longer linear searches at the leaf level.[5] Fig. 10 shows the end-to-end latency required to perform exact match and range queries (returning ~ 50 records) on a database with $10M$ encrypted 64-bit integer records indexed with both the DHE skip list and the modified HI-EM skip list. Results show that the DHE skip list performs slightly better than the modified HI-EM skip list, for both types of queries, thus confirming the effectiveness of our doubly-linked list design choice.

Final remarks. Adopting the DHE skip list over a standard data structure, such as the B-Tree in PostgreSQL, is a fundamental security requirement rather than a mere design preference in the context of queryable encryption with stateless clients and single-round query protocols (see Section 1 and Section 2). By leveraging its history-independence property and explicitly preventing the disclosure of duplicate values, the DHE skip list offers minimal leakage for semantically secure encrypted records and acceptable performance overhead.

8. Related work

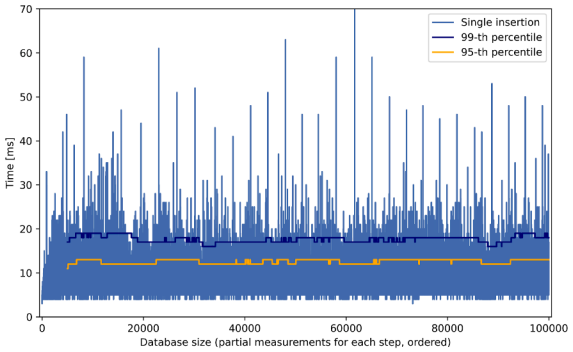
Encrypted Databases. Significant implementations proposed by researchers [25,26,34,40,41] and companies [42–45] combine a mixture of specialized techniques for supporting the encryption of data *in use* for popular workloads in specific database environments. The research efforts in this area encompass a wide range of approaches [46], such as new cryptographic primitives and schemes [8,9,11,47–50], and key management strategies to support cryptographic access control [51–53]. Our proposal is orthogonal to these research lines, as the proposed DHE index may be also useful to many types of existing and future cryptographic primitives, including techniques based on data distribution strategies [54,55], and can be integrated with key management strategies for fine-grained access control.

An alternative type of approach relies on techniques for aggregating and indexing encrypted data for efficient confidential retrieval [56–59]. These solutions offer a different type of security guarantees, not (only) based on cryptographic assumptions, but also on some types of heuristics. Moreover, a different class of proposals is based on trusted enclave technologies for fully fledged computation, like Intel SGX [41,60,61], hardware-based co-processors [62], and other non-standard trusted computing device [63]. This latter class of solutions theoretically achieves better performance and flexibility, but in practice showed to suffer from side-channel attacks [31,64].

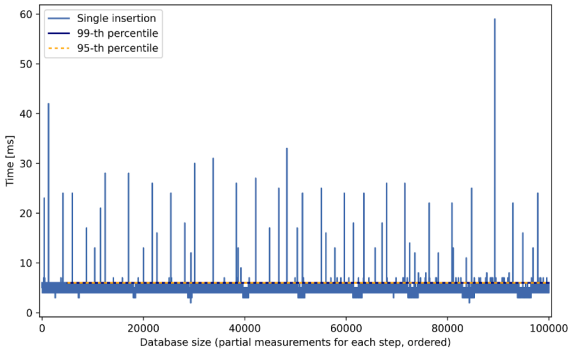
Parallel research efforts focus on analyzing the security of encrypted databases, and include novel attack techniques such as inference of statistical information [20,28,65], correlation among multiple data [38,66–68], and analyses of query patterns [30,69], access pattern [70,71] and communication volume [70,72].

Queryable encryption. In this paper, we do not modify existing encryption schemes or propose newer variants, but we only focus on indexing techniques. The state-of-the-art Order Revealing Encryption scheme by Lewi and Wu [8] (LW-ORE) was selected due to its security and efficiency trade-offs. However, we note that the proposed indexing data structure is not limited to left-right ORE schemes and can be used with other encryption schemes or even plaintext data.

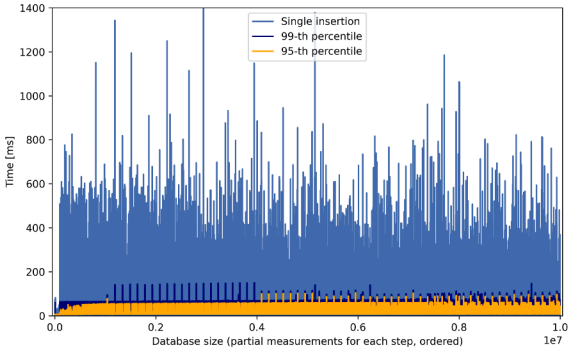
Our client is stateless and the query protocol is single-round, which are two important properties for practical and efficient deployments in real-world applications. The frequency-hiding OPE scheme by



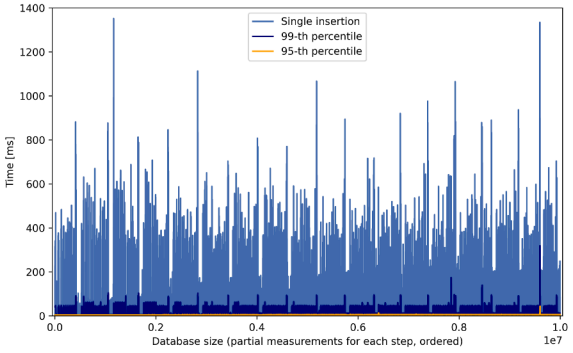
(a) DHE skip list index, DB size: 100k records



(b) PostgreSQL B-Tree index, DB size: 100k records



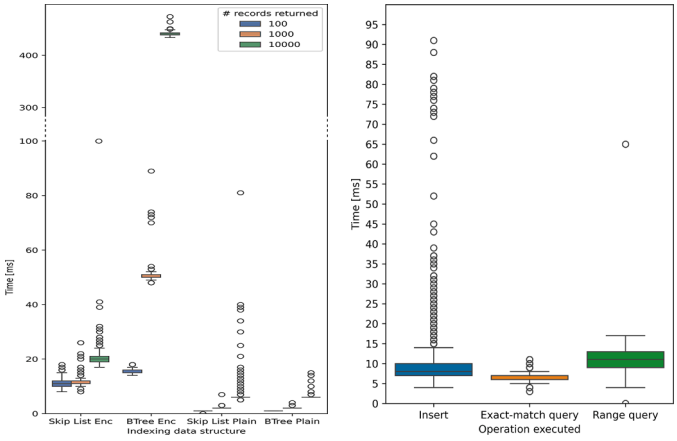
(c) DHE skip list index, DB size: 10M records



(d) PostgreSQL B-Tree index, DB size: 10M records

Fig. 7. E2E latency for insertion operations plotted incrementally (encrypted 64-bit integers).

[5] Note that our implementation of the *modified HI-EM* is based on the same code base of our DHE skip list, thus backward pointers still exist within data structures. However, we highlight that completely removing backward pointers may not be viable option, because other than the peculiar context of managing encrypted data, we found them also necessary for efficiently managing page splits, page merges, and array resizes of traversed nodes during insertions. Nonetheless, also considering that there is no other available implementation of such a data structure, we expect results to provide some insights on whether backward pointers can be useful for faster queries, and leave further analyses on optimized implementations as future work.



(a) E2E latency of range queries returning a different number of records, both encrypted and plaintext, indexed with the proposed DHE skip list or the PostgreSQL B-Tree. To improve readability of the figure, the y-axis is truncated and resized over 100 ms, as there are no values to show between 100 and 400 ms. DB size: 100k records

(b) E2E latency for insertion, exact-match query, and range query when adding 10k extra encrypted records to a database of size 1M records indexed with the proposed DHE skip list

Fig. 8. Additional E2E latencies with 64-bit integer data type.

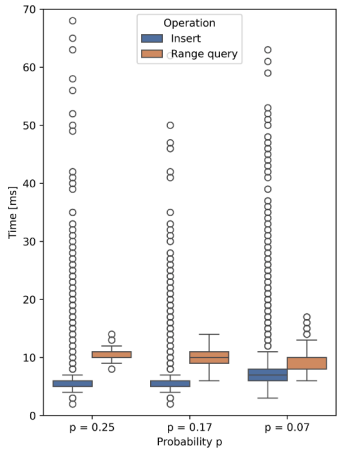


Fig. 9. E2E latency for insertion and range query operations by varying the promotion probability p (encrypted 64-bit integers, DHE skip list index, DB size: 100k records).

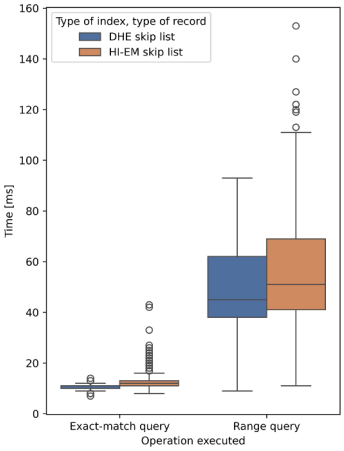


Fig. 10. Comparing the E2E latency between the DHE skip list and the modified HI-EM skip list (encrypted 64-bit integers, DB size: 10M records).

Kerschbaum [49] requires a client-side state, and the ORE scheme by Boneh et al. [9] produces larger ciphertexts and it's slower to compute. Arx [34] database system relies on Yao's garbled circuits to encrypt and compare the data inside the index, and requires the client (implemented as a proxy) to reconstruct part of the index after every use. Also, Arx requires additional rounds between the stateful client and the server that may increase the query overhead, and uses a history-independent treap data structure to implement the index for range queries, but requires the client to maintain a counter for each distinct value in the database.

Relevant literature associated with open source implementations of LW-ORE in databases adopts standard indexing techniques [17,19,35] that are vulnerable to inference attacks. CipherStash [43] also implements LW-ORE, but does not describe the indexing technique at server side. Among proposals which consider tight integration with production-ready databases, an interesting solution is represented by MongoDB Queryable Encryption [45], which supports searchable encryption [58]. However, the security analysis in [73] shows how the MongoDB Queryable Encryption implementation is vulnerable to inference attacks, and highlight the challenges in designing and integrating secure encryption schemes in real-world database systems.

History Independence. History independence is critical for applications where the data structure is shared by multiple users and where the sequence of operations is a sensitive information that should not be disclosed [23,74–77]. While the design of history-independent data structures was originally oriented towards applications in RAM, subsequent designs have been developed also for external memory [13,21,22,78,79]. In the context of encrypted databases, history-independent data structures have been implemented for indexing purposes, enhancing the security of the entire database system when deployed in real-world scenarios [34,80,81]. However, to the best of our knowledge, we are the first to address both the need for hiding duplicate values and history-independence. Moreover, evaluation of our implementation may also be of interest for the original design proposed by Bender et al. [13], for which there are no known implementations openly available.

Stronger security guarantees: Oblivious RAM (ORAM). ORAM [14–16] is a popular approach to hide access patterns to data stored on untrusted servers. Although state-of-the-art practical ORAM constructions [82,83] typically offer stronger security guarantees, they still need to maintain some form of mapping between logical addresses and randomized physical storage locations, which may be difficult to apply to real-world databases. First, ORAM is stateful, thus the network and computation overhead for synchronizing the client and server states is significant. Second, ORAM requires a multi-round protocol for query the database (e.g. to shuffle and/or re-encrypt data blocks after every access), making it impractical for large-scale databases or latency-sensitive applications. Third, integrating ORAM with existing DBMS architectures is challenging, as it often requires redesigning storage and query processing layers. In contrast, our approach minimizes leakage while maintaining practical performance and compatibility with standard database systems, without the heavy overhead and complexity of ORAM.

9. Conclusions

We design and implement a new indexing data structure, the Doubly-linked History-independent External-memory (DHE) skip list, which can be used to secure queryable-encrypted databases. Previous works either do not adopt indexes, adopt standard indexes already available within the database at the cost of affecting security guarantees of the encryption scheme, or in-memory data structures which do not suit database contexts. Our index guarantees minimal leakage by not leaking duplicates or information about the history of transactions. Our solution can be used to increase confidentiality of databases in the event of a data breach, both when the database is deployed on-premise or in the Cloud. We demonstrate the practicality of the index by developing a prototype

extension for PostgreSQL that uses Order Revealing Encryption (ORE), a state-of-the-art queryable encryption scheme for exact-match and range queries. Our implementation achieves performance that is comparable to the standard B-Tree implementation of PostgreSQL in most configurations, only showing a generalized increase in the number of outliers, and worse performance on insertion operations when dealing with very large databases storing ten million records. Also, thanks to our efficient encoding technique, we are able to minimize the space overhead of the ORE encrypted records. Thus, our DHE skip list can be a valid alternative not only for securely indexing ORE-encrypted records, but possibly in other scenarios.

CRediT authorship contribution statement

Mattia Trabucco: Writing – review & editing, Writing – original draft, Visualization, Software, Resources, Methodology, Investigation, Data curation, Conceptualization; **Mauro Andreolini:** Visualization, Validation, Supervision, Software; **Luca Ferretti:** Writing – review & editing, Writing – original draft, Validation, Supervision, Project administration, Conceptualization.

Data availability

Source code of our Doubly-linked History-independent External-memory skip list implementation as a PostgreSQL extension is available at https://secloud.ing.unimore.it/shared/trab/postgresql_jisa26.zip

Declaration of competing interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] Microsoft. Transparent data encryption. https://learn.microsoft.com/en-us/sql/relational-databases/security/encryption/transparent-data-encryption; Vis. May 2025.
[2] Hacigümüş H, Iyer B, Mehrotra S. Providing database as a service. In: Proceedings of the 18th IEEE Int'l conference data engineering. 2002.
[3] Saleem H, Naveed M. SoK: anatomy of data breaches. In: Proceedings of the 20th privacy enhancing technologies symposium 2020.
[4] Micciancio D. Oblivious data structures: applications to cryptography. In: Proceedings of the 29th ACM symposium theory of computing. 1997.
[5] Snyder L. On uniquely represented data strauctures. In: 18th symposium foundations of computer science. 1977.
[6] Pugh W. Skip lists: a probabilistic alternative to balanced trees. Commun ACM 1990; 33, 668–76.
[7] Seidel R, Aragon CR. Randomized search trees. Algorithmica 1996; 16, 464–97.
[8] Lewi K, Wu DJ. Order-revealing encryption: new constructions, applications, and lower bounds. In: Proceedings of the 23rd ACM CCS. 2016.
[9] Boneh D, Lewi K, Raykova M, Sahai A, Zhandry M, Zimmerman J. Semantically secure order-revealing encryption: multi-input functional encryption without obfuscation. In: Proceedings of the IACR advances in cryptology (EUROCRYPT). 2015.
[10] Chenette N, Lewi K, Weis SA, Wu DJ. Practical order-revealing encryption with limited leakage. In: Proceedings of the IACR fast software encryption. 2016.
[11] Kamara S, Moataz T. SQL on structurally-encrypted databases. In: Proceedings of the ASIACRYPT. 2018.
[12] Ferretti L, Trabucco M, Andreolini M, Marchetti M. How (not) to index order revealing encrypted databases. In: Proceedings of the ITASEC 2023: the italian conf. on cybersecurity. 2023.
[13] Bender MA, Berry JW, Johnson R, Kroeger TM, McCauley S, Phillips CA, et al. Anti-persistence on persistent storage: history-independent sparse tables and dictionaries. In: Proceedings of the 35th ACM symposium principles of database systems. 2016.
[14] Garg S, Mohassel P, Papamanthou C. TWORAM: efficient oblivious ram in two rounds with applications to searchable encryption. In: CRYPTO 2016. 2016.
[15] Faber S, Jarecki S, Kentros S, Wei B. Three-party ORAM for secure computation. In: Proceedings of the ASIACRYPT. 2015.
[16] Chen H, Chillotti I, Ren L. Onion ring ORAM: efficient constant bandwidth oblivious RAM from (leveled) TFHE. In: Proceedings of the ACM CCS. 2019.
[17] Alves P, Aranha D. A framework for searching encrypted databases. J Internet Serv Appl 2018; 9, 1–18.
[18] PostgreSQL. Chapter 67.4.3. B-tree indexes. deduplication. https://postgresql.org/docs/16/btree-implementation.html; Vis. May 2025.
[19] Bogatov D, Kollios G, Reyzin L. A comparative evaluation of order-revealing encryption schemes and secure range-query protocols. In: Proceedings of the conference VLDB. 2019.
[20] Naveed M, Kamara S, Wright CV. Inference attacks on property-preserving encrypted databases. In: Proceedings of the 22nd ACM CCS. 2015.
[21] Bender MA, Farach-Colton M, Goodrich MT, Komlós H. History-independent dynamic partitioning: operation-order privacy in ordered data structures. Proc ACM Manage Data 2024; 2, 1–27.
[22] Golovin D. B-treaps: a uniquely represented alternative to b-trees. In: Int'l conference automata, languages, and programming. 2009.
[23] Moran T, Naor M, Segev G. Deterministic history-independent strategies for storing information on write-once memories. In: Int'l conference automata, languages, and programming. 2007.
[24] Agrawal R, Kiernan J, Srikant R, Xu Y. Order preserving encryption for numeric data. In: Proceedings of the ACM Int'l conference management of data. 2004.
[25] Popa RA, Redfield C MS, Zeldovich N, Balakrishnan H. CryptDB: protecting confidentiality with encrypted query processing. In: Proc. 23rd ACM symposium operating systems principles. 2011.
[26] Ferretti L, Colajanni M, Marchetti M. Distributed, concurrent, and independent access to encrypted cloud databases. IEEE Trans Parallel and Distributed Systems 2014; 25, 437–46.
[27] Grubbs P, McPherson R, Naveed M, Ristenpart T, Shmatikov V. Breaking web applications built on top of encrypted data. In: Proceedings of the 23rd ACM CCS. 2016.
[28] Grubbs P, Sekniqi K, Bindschaedler V, Naveed M, Ristenpart T. Leakage-abuse attacks against order-revealing encryption. In: IEEE Symposium S&P. 2017.
[29] Katz J, Lindell Y. Introduction to modern cryptography. CRC Press; 2020. ISBN 9781351133029.
[30] Oya S, Kerschbaum F. Hiding the access pattern is not enough: exploiting search pattern leakage in searchable encryption. In: Proc. 30th USENIX security symp. 2021.
[31] Li M, Zhao X, Chen L, Tan C, Li H, Wang S, et al. Encrypted databases made secure yet maintainable. In: Proceedings of the 17th USENIX symposium OSDI. 2023.
[32] Hartline JD, Hong ES, Mohr AE, Pentney WR, Rocke EC. Characterizing history independent data structures. Algorithmica 2005; 2518, 229–40.
[33] Naor M, Teague V. Anti-persistence: history independent data structures. In: Proceedings of the 33rd ACM symposium theory of computing. 2001.
[34] Poddar R, Boelter T, Popa RA. Arx: an encrypted database using semantically secure encryption. Cryptology ePrint Arch 2016; 12, 1664–78.
[35] Project TE. ENcrypted query operations. https://enquo.org; Vis. May 2025.
[36] Project TE. PG_Enquo reduced security mode. https://github.com/enquo/pg_enquo/tree/main/doc/data_types; Vis. May 2025.
[37] Ferretti L, Pierazzi F, Colajanni M, Marchetti M. Performance and cost evaluation of an adaptive encryption architecture for cloud databases. IEEE Trans Cloud Comput 2014; 2, 143–55.
[38] Grubbs P, Ristenpart T, Shmatikov V. Why your encrypted database is not secure. In: Proceedings of the 16th workshop hot topics in operating systems. 2017.
[39] Lindell Y. How to simulate it–a tutorial on the simulation proof technique. Tutorials Found Cryptography 2017; 0, 277–346.
[40] Tu S, Kaashoek MF, Madden S, Zeldovich N. Processing analytical queries over encrypted data. In: Proceedings of the conference VLDB. 2013.
[41] Vinayagamurthy D, Gribov A, Gorbunov S. StealthDB: a scalable encrypted database with full SQL query support. Proc Privacy Enhancing Technol 2019; 2019, 370–88.
[42] Antonopoulos P, Arasu A, Singh KD, Eguro K, et al. Azure SQL database always encrypted. In: Proceedings of the ACM Int'l conference management of data. 2020.
[43] CipherStash. Order-revealing encryption library. https://github.com/cipherstash/ore.rs; Vis. May 2025.
[44] LLC A WS. Aws govcloud. https://aws.amazon.com/govcloud-us/; Vis. May 2025.
[45] MongoDB I. MongoDB queryable encryption. https://www.mongodb.com/docs/manual/core/queryable-encryption; Vis. May 2025.
[46] Fuller B, Varia M, Yerukhimovich A, Shen E, Hamlin A, Gadepally V, et al. SoK: Cryptographically protected database search. In: IEEE Symposium S&P. 2017.
[47] Gentry C. Fully homomorphic encryption using ideal lattices. In: Proceedings of the 41st ACM symposium theory of computing. 2009.
[48] Michalas A. The lord shares: combining attribute-based encryption and searchable encryption for flexible data sharing. In: Proceedings of the 34th ACM symposium applied computing. 2019.
[49] Kerschbaum F. Frequency-hiding order-preserving encryption. In: Proceedings of the 22nd ACM CCS. 2015.
[50] Amjad G, Patel S, Persiano G, Yeo K, Yung M. Dynamic volume-hiding encrypted multi-maps with applications to searchable encryption. Proc Privacy Enhancing Technol 2023; 2023, 417–36.
[51] De Capitani D VS, Foresti S, Jajodia S, Paraboschi S, Samarati P. Over-encryption: management of access control evolution on outsourced data. In: Proceedings of the 33rd Int'l conference VLDB. 2007.
[52] Ferretti L, Colajanni M, Marchetti M. Access control enforcement on query-aware encrypted cloud databases. In: IEEE 5th Int'l conference cloud computing technology and science. 2013.
[53] Alansari S, Paci F, Sassone V. A distributed access control system for cloud federations. In: IEEE Conference distributed computing systems. 2017.
[54] Hadavi MA, Damiani E, Jalili R, Cimato S, Ganjei Z. AS5: a secure searchable secret sharing scheme for privacy preserving database outsourcing. In: Data privacy management and autonomous spontaneous security: 7th Int'l workshop DPM and SETOP. 2013.
[55] Yuan X, Guo Y, Wang X, Wang C, Li B, Jia X. EncKV: an encrypted key-value store with rich queries. In: Proceedings of the 12th ACM ASIACCS. 2017.

[56] Hacigümüş H, Iyer B, Li C, Mehrotra S. Executing SQL over encrypted data in the database-service-provider model. In: Proceedings of the ACM Int'l conference management of data. 2002.
[57] Damiani E, De Capitani VS, Jajodia S, Paraboschi S, Samarati P. Balancing confidentiality and efficiency in untrusted relational DBMSs. In: Proceedings of the 10th ACM CCS. 2003.
[58] Chase M, Kamara S. Structured encryption and controlled disclosure. In: Proceedings of the ASIACRYPT. 2010.
[59] Mayberry T, Blass E-O, Chan AH. Efficient private file retrieval by combining ORAM and PIR. In: Network and distributed system security symposium 2014.
[60] Priebe C, Vaswani K, Costa M. EnclaveDB: a secure database using SGX. In: IEEE Symposium S&P. 2018.
[61] Fuhry B, Jain HAJ, Kerschbaum F. EncDBDB: searchable encrypted, fast, compressed, in-memory database using enclaves. In: 51st IEEE/IFIP Int'l conference dependable systems and networks. 2021.
[62] Bajaj S, Sion R. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In: Proceedings of the 2011 ACM Int'l conference management of data. 2011.
[63] Arasu A, Blanas S, Eguro K, Kaushik R, Kossmann D, Ramamurthy R, et al. Orthogonal security with cipherbase. In: Conference innovative data systems research. 2013.
[64] Fei S, Yan Z, Ding W, Xie H. Security vulnerabilities of SGX and countermeasures: a survey. ACM Comput Surv 2021; 54, 1–36.
[65] Islam MS, Kuzu M, Kantarcioglu M. Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In: Network and distributed system security symposium 2012.
[66] Durak FB, DuBuisson TM, Cash D. What else is revealed by order-revealing encryption? In: Proceedings of the 23rd ACM CCS. 2016.
[67] Bindschaedler V, Grubbs P, Cash D, Ristenpart T, Shmatikov V. The tao of inference in privacy-protected databases. 2018; 11, 1715–28.
[68] Falzon F, Markatou EA, Akshima, Cash D, Rivkin A, Stern J, et al. Full database reconstruction in two dimensions. In: Proceedings of the 27th ACM CCS. 2020.
[69] Kornaropoulos EM, Papamanthou C, Tamassia R. The state uniform: attacks on encrypted databases beyond the uniform query distribution. In: IEEE Symposium S&P. 2020.
[70] Kellaris G, Kollios G, Nissim K, O'Neill A. Generic attacks on secure outsourced databases. In: Proceedings of the ACM CCS. 2016.
[71] Lacharité M-S, Minaud B, Paterson KG. Improved reconstruction attacks on encrypted data using range query leakage. In: IEEE Symposium S&P. 2018.
[72] Grubbs P, Lacharite M-S, Minaud B, Paterson KG. Pump up the volume: practical database reconstruction from volume leakage on range queries. In: Proceedings of the ACM CCS. 2018.
[73] Gui Z, Paterson KG, Tang T. Security analysis of MongoDB queryable encryption. In: 32nd USENIX security symposium 2023.
[74] Bethencourt J, Boneh D, Waters B. Cryptographic methods for storing ballots on a voting machine. In: NDSS. 2007.
[75] Bajaj S, Sion R. HIFS: history independence for file systems. In: Proceedings of the 20th ACM CCS. 2013.
[76] Goodrich MT, Kornaropoulos EM, Mitzenmacher M, Tamassia R. More practical and secure history-independent hash tables. In: Proceedings of the 21st european symp. research in computer security (ESORICS). 2016.
[77] Gao B, Chen B, Jia S, Xia L. EHIFS: an efficient history independent file system. In: Proceedings of the ACM ASIACCS. 2019.
[78] Golovin D. The B-skip-list: a simpler uniquely represented alternative to B-trees. 2010 arXiv:10050662.
[79] Bender MA, Farach-Colton M, Johnson R, Mauras S, Mayer T, Phillips CA, et al. Write-optimized skip lists. In: Proceedings of the 36th ACM symposium principles of database systems. 2017.
[80] Bajaj S, Sion R. FickleBase: looking into the future to erase the past. In: IEEE 29th Int'l conference data engineering. 2013.
[81] Roche DS, Aviv A, Choi SG. A practical oblivious map data structure with secure deletion and history independence. In: IEEE Symposium S&P. 2016.
[82] Stefanov E, Dijk MV, Shi E, Chan T-HH, Fletcher C, Ren L, et al. Path ORAM: an extremely simple oblivious RAM protocol. J. ACM 2018.
[83] Zhang B, Cui H, Yuan X, Yu Z, Guo B. {V-ORAM}: A versatile and adaptive {ORAM} framework with service transformation for dynamic workloads. In: 34th USENIX security symposium 2025.