HybrIDX: New Hybrid Index for Volume-hiding Range Queries in Data Outsourcing Services

Kui Ren[‡], Yu Guo[†], Jiaqi Li[‡], Xiaohua Jia[†], Cong Wang[†], Yajin Zhou[‡], Sheng Wang[§], Ning Cao[£], Feifei Li[§]
[‡]Zhejiang University, [†]City University of Hong Kong, [§]Alibaba Group, [£]Hangzhou Mishu Technology.
{kuiren, lijiaqi93, yajin_zhou}@zju.edu.cn, y.guo@my.cityu.edu.hk, {congwang, csjia}@cityu.edu.hk,
{sh.wang, lifeifei}@alibaba-inc.com, ncao@hzmstech.com

***Abstract*—An encrypted index is a data structure that assists untrusted servers to provide various query functionalities in the ciphertext domain. Although traditional index designs can prevent servers from directly obtaining plaintexts, the confidentiality of outsourced data could still be compromised by observing the volume of different queries. Recent volume attacks have demonstrated the importance of sealing volume-pattern leakage. To this end, several works are made to design secure indexes with the volume-hiding property. However, prior designs only work for encrypted keyword search. Due to the unpredictable range query results, it is difficult to protect the volume-pattern leakage while achieving efficient range queries.**

In this paper, for the first time, we define and solve the challenging problem of volume-hiding range queries over encrypted data. Our proposed hybrid index framework, called HybrIDX, allows an untrusted server to efficiently search encrypted data based on order conditions without revealing the exact result size. It resorts to the trusted hardware techniques to assist range query processing by moving the comparison algorithm to trusted SGX enclaves. To enable volume-hiding data retrieval, we propose to host encrypted file blocks outside the enclave in an encrypted volume-hiding structure. Apart from this novel hybrid index framework, we further customize a result caching method to obfuscate the results co-occurrence among different queries. We formally analyze the security strengths and complete the prototype implementation. Evaluation results demonstrate the feasibility and practicability of our designs.

I. INTRODUCTION

An encrypted index offers a secure interface for data owners to outsource private data to an untrusted server, while selectively retrieving data without decryption. It is recognized as a fundamental component of building privacy-preserving applications. Early studies on secure index construction mainly focus on the direction of keyword search [1]–[7]. Along with the development of data outsourcing paradigm, enabling efficient range queries over secure indexes has attracted a lot of attention from both academia and industry [8]–[11]. A typical example in the encrypted database community [12]–[17] is finding all matched files according to order conditions such as indexed values or timestamps.

Over the past decade, the demand for designing range-based indexes has been expected to be addressed by the innovation of cryptographic techniques, e.g., Order-preserving/revealing encryption (OPE/ORE) schemes [18]–[27], which allow order comparisons to be performed directly on ciphertexts. In practice, however, these schemes are developed as cryptographic primitives and can hardly accommodate system-wise security requirements [28]–[36]. Specifically, range queries reveal the different number of return results (i.e., volume-pattern) and the results co-occurrence pattern. Recent leakage-abuse attacks [28]–[32] have shown how these leakages can be exploited to learn sensitive information about the query content and outsourced data. To mitigate this security concern, a simple approach is to apply naive padding. However, this approach would introduce expensive storage overhead because the result set is padded to the maximum length.

In the literature, only a few recent works [37]–[39] have started to study practical index designs with the protection of volume-pattern. In [37], Kamara *et al.* proposed the first practical volume-hiding encryption scheme by using the multi-map data structure [38]. The follow-up design [39] utilized cuckoo hashing and differential privacy to further reduce the volume and storage overhead. Unfortunately, neither of the existing schemes can support volume-hiding range queries because of two challenging issues. Firstly, unlike keyword search functions, the result size of different range queries cannot be pre-defined. Clients without pre-knowledge of the result size have to download all the data in order to hide the volume-pattern leakage, which demands a large amount of bandwidth overhead. Secondly, different range queries reveal the results co-occurrence pattern, which can be exploited for inferring the plaintext distribution. For example, the result set associated with the endpoint values must appear in all the other range query results. By observing the results co-occurrence among different queries, an attacker can still learn the result distribution even if the volume-pattern has been well-protected [28]–[30]. Therefore, how to enable volume-hiding range queries over encrypted indexes is still challenging and remains to be fully explored.

In this paper, for the first time, we solve the problem of volume-hiding range queries in the data outsourcing paradigm. Our proposed hybrid index framework, called HybrIDX, can effectively address the aforementioned challenges while being efficient in supporting range queries. To achieve our design goals on both security and usability, we propose to bring together the advancement of both volume-hiding data structure and trusted hardware to build the desired index construction. To hide the result size of different range queries, we first propose to utilize the trusted enclave to securely convert any range query into multiple independent sub-queries with equal fixed volume. While seemingly inconvenient, this way of query

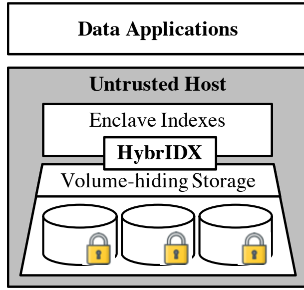


Fig. 1: Overview of main HybrIDX components.

processing is actually consistent with the rationale of many practical applications in information retrieval, which would on-demand display a subset of query results per round to the client upon request [40].[1]

In order to concretely achieve the fixed volume for sub-queries, we explore the volume-hiding storage structure from the bucketization-based padding method [37] that converts file blocks of different values into multiple block ciphertexts with a fixed length. As directly outsourcing the resulting volume-hiding ciphertexts cannot support range queries, we devise a range-based index inside the trusted enclave to indicate the corresponding encrypted file blocks, as shown in Fig.1. Besides, we customize a batch query protocol for our purpose to retrieve range query results in a volume-hiding manner.

Under this hybrid index framework, we then consider how to further mitigate the co-occurrence pattern leakage among different range queries. Particularly, we propose a secure caching method that caches retrieved results inside the enclave and refreshes them periodically. With the assistance of an enclave-based caching method, not only can we efficiently obfuscate the co-occurrence leakage among different queries, but also we can avoid high communicational cost between the client and servers for indexes refresh. We believe the proposed hybrid index design provides useful guidelines and new insights on designing secure indexes for modern data outsourcing services. The main contributions of this paper can be summarized as follows:

- We present HybrIDX, the first hybrid index framework that enabling volume-hiding range queries over encrypted data. It stores encrypted file blocks of indexed values in the form of volume-hiding structure and conducts efficient range comparisons inside the enclave.
- We carefully tailor a secure batch query protocol and a result caching method for volume-pattern protection and co-occurrence pattern obfuscation.
- We provide thorough analysis investigating security and efficiency guarantees of the proposed designs. By characterizing and analyzing the leakage profiles from both the index construction and the query protocol, we prove the security of our schemes.
- We implement the system prototype and conduct a comprehensive evaluation. The experiment results show that our range query protocol can fetch all matched results within a short latency, and achieve better query efficiency compared with existing cryptographic solutions.

[1] Similar insight has also been utilized by prior art Oblix [41], but for ranked keyword search that is different from our focus on range queries.

TABLE I: Security characterization of representative schemes for secure range query operations

Primitives	Query Efficiency	Leakage Protection			
		Order Result	Search Pattern	Access Pattern	Volume Pattern
OPE					
ROPF [23]	$O(\log m)$	BS	✗	✗	✗
Ideal-OPE [24]	$O(\log m)$	AS	✗	✗	✗
POPE [22]	$O(\log m)$	BS	✗	✗	✗
ORE					
CLWW [20]	$O(m)$	AS	✗	✗	✗
Lewi-Wu [18]	$O(m)$	AS	✗	✗	✗
PHORE [19]	$O(m)$	AS	✗	✗	✗
Enclave					
HardIDX [42]	$O(\log m)$	✓	✓	✗	✗
Opaque [43]	$O(\log m)$	N/S	✓	✓	✗
ZeroTrace [44]	$O(\log n)$	N/S	✓	✓	✗
Oblix [41]	$O(\log^2 n)$	N/S	✓	✓	✓*
EnclaveDB [45]	$O(\log m)$	✓	✓	✓	✗
HybrIDX	$O(\log m)$	✓	✓	✓	✓

1 ✓*: result may be lossy.
2 *BS*: before search ⊂ *AS*: after search.
3 n: number of total records; m: number of indexed values, $n >> m$.
4 *N/S*: the feature may be potentially supported, but not explicitly handled.

The rest of this paper is organized as follows. Section II introduces background knowledge and related work involved in our designs. Section III presents our system architecture and threat assumptions. Section IV introduces our system design. Our security analysis is conducted in Section V, and an extensive array of evaluation results is shown in Section VI. Finally, we conclude the paper in Section VII.

II. Related Work

A. *Encrypted Range Queries*

Searchable encryption (SE) scheme for secure range queries has been an active research area in the past decade [18]–[27]. To make a clear comparison, we have summarized the representative solutions and compiled their security features in Table I. The early studies, known as order-preserving encryption (OPE) [23], can preserve the original orders of plaintexts by using a random order-preserving function (ROPF). Thus, an untrusted server is able to make numeric comparisons between ciphertexts as if it had operated on plaintexts. However, orders are directly leaked from OPE ciphertexts, which make OPE ciphertexts suffer from the sorted attacks [34]. In [24], Popa *et al.* proposed an ideal OPE scheme (Ideal-OPE) by encrypting values via standard encryption, but it requires multiple interactions for client-side comparison. To mitigate the order leakage, the notion of order-revealing encryption (ORE) was proposed [26]. ORE ciphertext has no particular order, while order relations are revealed during dedicated comparison protocols [18]–[20]. However, ORE schemes would

not be necessarily applicable in real-world applications because orders are masked in each ORE ciphertext. It requires the server to perform a linear scan on the whole ORE-based indexes for range queries. Considering the potentially huge amount of outsourced data in servers, this problem is particularly challenging as it is extremely difficult to meet the performance requirement [17]. Moreover, recent leakage-abuse attacks [33], [36] and volume attacks [28]–[30] show that the plaintext distribution can still be determined by observing the comparison results and volume size.

B. Volume-hiding Scheme

Recent volume attacks [28]–[30] on searchable encryption schemes have brought an emphasis on the importance of volume-hiding property, which is the new concept behind leakage-abuse attacks. To hide the volume-pattern leakage, a simple solution is to apply naive padding so that each result size is the maximum length. However, naive padding results in too much storage overhead larger than the original dataset. In [37] and its follow-up work [38], Kamara *et al.* proposed the first volume-hiding encryption scheme for encrypted keyword search. The core idea is to map the file blocks of each keyword into multiple ciphertext blocks with a fixed length, and later use the (fixed) maximum volume of the input datasets to retrieve the matched results from these ciphertext blocks. Since the retrieved results of each query are padded up to the fixed size, the property of volume-hiding is achieved. In a concurrent and independent work [39], Patel *et al.* provided another different volume-hiding scheme based on cuckoo hashing and differential privacy to further reduce the padding overhead. However, none of the existing volume-hiding schemes can support range queries over encrypted data.

C. Hardware Enclaves and SGX

Trusted execution environments (TEE) or secure enclaves such as Intel SGX are recent advances in computer processor technology [46]. It provides three main security properties: *isolation*, *sealing*, and *remote attestation*. First, SGX enforces *isolation* by storing the pre-defined code and data in a hardware guarded memory (limited 128 MB) that only particular enclave code can access it. The processor ensures that any software outside the enclave cannot read or modify it. Second, *sealing* enables encrypting and authenticating the enclave data. Third, *remote attestation* can establish a secure channel between the external party and the enclave, which guarantees that the target code is indeed running securely and unmodified within the enclave of the remote system. By deploying the enclave code to the server, it allows the untrusted server to securely maintain data on behalf of the data owner.

Several attempts have been made to securely process sensitive data with trusted hardware [41]–[45], [47]–[50]. The work [42] used the trusted hardware to devise encrypted key-value stores, but did not support update operations and cannot protect the access-pattern leakage. To address this issue, most of works [41], [43], [44] mainly focus on using ORAM [51] for oblivious data processing. In Lightbox [49], the authors

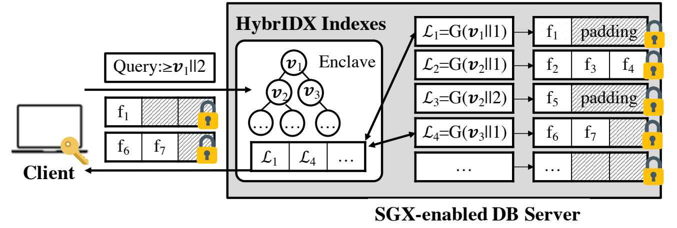


Fig. 2: Overview of HybrIDX architecture.

proposed an encrypted cache approach to improve the process efficiency, as demanded by their enclave-based stateful network middlebox. Very recently, EnclaveDB [45] was proposed to devise an encrypted database system by hosting all data in the enclave memory. However, such straightforward treatment would not be able to deploy in real-world systems because the memory size of an enclave is limited. Moreover, all above-mentioned solutions do not consider volume-pattern protection for range queries and the performance overhead of making all memory accesses through its ORAM-based index can be significant as shown in Table I.

III. SYSTEM OVERVIEW

A. Architecture Overview

Fig. 2 shows our system architecture, containing two types of entities: a *client* of the data application and an SGX-enabled storage *server*. HybrIDX is particularly suitable for the clients who wish to outsource their sensitive dataset to remote servers and later enjoy encrypted query services. Before outsourcing encrypted files to the server, the client extracts index values (Fig.2: v) from the dataset to generate range-based indexes. The related file of each value is divided into multiple file blocks with equal length. Then, the client transforms the file blocks (Fig.2: f) into multiple block ciphertexts with padding in a volume-hiding fashion. Finally, the client can deploy the range-based indexes on the enclave via SGX remote attestation and outsources the volume-hiding file blocks to the server in the form of encrypted label-value (L-V) pairs.

The storage server in HybrIDX consists of a tree-based index inside an enclave and a set of encrypted L-V pairs hosted by the untrusted storage. It provides an interface for clients to search and update L-V pairs by using batch query algorithms. Specifically, the client executes encrypted range queries by establishing a secure channel with the enclave for sending query tokens. The enclave processes the token over its range-based indexes via binary search and generates pseudo-random labels to fetch encrypted file blocks from external storage. In HybrIDX, different range query results are cached inside the enclave for later secure index refresh.

HybrIDX achieves efficient range queries by hosting tree-based indexes inside secure enclave memory. Obviously, this hardware-assisted construction can provide better performance than existing software-centric solutions (e.g., ORE-based index), which usually demand a linear scan. For volume-pattern protection, HybrIDX maps file blocks from different ranges to encrypted volume-hiding data structure. To search values that

lie in a range of domains, the enclave generates secure labels based on query requests from the client, which provides the server with a controlled capability to return a fixed number of results. Therefore, our hybrid index design fully scrambles the result distribution. With this observation, we believe that our proposed design could be a more practical starting point for developing secure data outsourcing services.

B. Threat Model

We consider a powerful adversary that can control the entire software stack on the server-side, except the code inside the enclave. It follows the protocols specified by the client but attempts to infer the plaintext information from the available data and background knowledge about the dataset distribution. In particular, the adversary can monitor query protocols and learn about query tokens, accessed index items, and encrypted results during each batch query. Due to the protection of SGX, the adversary cannot directly access the contents of protected memory pages and CPU registers. Denial-of-service attacks and side-channel attacks through cache-timing [52]–[54] are beyond the scope of this work but have been addressed by orthogonal studies like [55]. The client is always secure and trusted. The encryption keys are securely stored at the client-side and the enclave. Besides, there is a secure channel in place to support secure communications between the client and the server-side enclave.

C. Preliminaries

Notation: The set of all binary strings of length n is denoted as $\{0,1\}^n$. The output x of an algorithm $F(.)$ is denoted by $x \leftarrow F(.)$, and an element x being sampled randomly from a set X is denoted by $x \overset{\$}{\leftarrow} X$. We define $L[i]$ as the i-th element of the array L. We use "$\lfloor . \rfloor$" to denote a floor function, and use "$|.|$" to represent the number of data. We refer to "$||$" as the concatenation operation.

Symmetric encryption: A symmetric encryption scheme $(KGen, Enc, Dec)$ is a set of three polynomial time algorithms: the key generation algorithm $KGen$ takes a security parameter λ as input and outputs a secret key k; the encryption algorithm Enc takes a key k and a value $v \in \{0,1\}^n$ as inputs and outputs a ciphertext $c \in \{0,1\}^n$; the decryption algorithm Dec takes a key k and a ciphertext c as inputs and returns v.

Pseudo-random function: A pseudo-random function $f(.)$ transforms each element x of the set X to an output $y \in Y$ with a secret key $k_f \in K$ such that y is computationally indistinguishable from a truly random function. A pseudo-random function $f : X \times K \rightarrow Y$ is (t, q, ϵ_f) secure if for every oracle algorithm A making at most q oracle queries and with polynomial runtime at most t: $|P_r[A^{f(\cdot,k_f)} = 1 | k_f \leftarrow K] - P_r[A^g = 1 | g \leftarrow F : X \rightarrow Y]| < \epsilon_f$ where ϵ_f is a negligible function in f.

IV. THE PROPOSED DESIGN

In this section, we present the designs of our hybrid indexes in detail based on the hardware-assisted indexes and the volume-hiding data structure. We will show that HybrIDX can support secure and efficient range queries and effectively hide the volume-pattern and co-occurrence pattern. Features such as batch queries, results caching, and update operations are also presented for security and practical considerations.

A. Design Rationale

To enable volume-hiding range queries over encrypted data, the main challenging issue is to construct the underlying index framework. To this end, volume-hiding primitives for encrypted keyword search are designed [37]–[39]. The core idea is to map a set of file blocks with padding into multiple block ciphertexts with equal length, and later use the maximum (fixed) volume to retrieve these results. However, it is difficult to directly apply existing volume-hiding structure to build range query indexes because of two main reasons: $1)$ Volume-hiding encryption requires pre-knowledge on the maximum volume among different queries. Since the maximum volume of range queries is the entire dataset, it is impossible to use a pre-defined volume to obtain all range query results; $2)$ Different range queries reveal the results co-occurrence pattern, which can be exploited for inferring the plaintext values. For example, when conducting a "$<$" comparison, the minimum value is always a subset of the other range query results. Although the design of volume-hiding structure can protect the exact result size, the result co-occurrence among different range queries can still leak the data distribution. Therefore, we are aware that existing cryptographic solutions still fall short of meeting the volume-hiding requirements.

To resolve the aforementioned challenges, we identify that the major obstacle in designing the volume-hiding index is to learn the volume of different range queries. Our insight is to devise a new hybrid index framework that utilizes the latest trusted hardware techniques to compute the query volume and maintains the relevant file blocks with fixed size outside the enclave. In particular, we first construct a range-based index on data values and store it inside the enclave. The enclave index can be used for converting any range query results into multiple sub-set with fixed volume so that the number of return results in each round is fixed. Since the range-based index is delegated with lightweight comparison tasks, it is easy to implement within the constrained trusted memory to assist generic operations on encrypted files. Meanwhile, we leverage the bucketization-based padding method in [38] to build secure volume-hiding structure for storing file blocks with a fixed size outside the enclave. With this hybrid index framework, the range query results can be retrieved in a batched manner with a fixed volume. Since the search result is in (fixed size) cipher blocks with padding instead of individuals, the server cannot learn the exact volume of range query results.

The above index framework allows encrypted range queries while simultaneously achieving volume-hiding property. However, it does not directly facilitate results co-occurrence obfuscation because the locations of fetched results are not considered to be protected. To address this, we customize a result caching technique that uses the enclave to re-encrypt fetched results during each batch query. Specifically, different

Algorithm 1: *Build*: Build encrypted hybrid indexes

Input: Private keys $\{k_1, k_2\}$, secure PRF G, fixed size p, index node N, and file blocks $DB(v)$ for value v.
Output: Encrypted hybrid indexes $\{I_{\text{SGX}}, I_{\text{DB}}\}$.

1. **Client.Build**(v, $DB(v)$)**:**
2. **begin**
3.   **for** each index value v **do**
4. $\beta \leftarrow \left\lfloor \frac{|DB(v)|}{p} \right\rfloor$, divide $DB(v)$ into $\beta + 1$ blocks;
5.   Pad last block to p items if needed;
6.   Set states $\{c||t\} = 0$;
7.   **for** each file block **do**
8. $L \leftarrow G(k_1, v||c||t)$, $\gamma \xleftarrow{\$} \{0,1\}^\lambda$;
9. $V \leftarrow \{f_1||..||f_p\} \oplus G(k_2, \gamma)$, $c++$;
10.   Add $\{L, V, \gamma\}$ to I_{DB};
11.   Put states $\{c||t\}$ to new node $N[v]$;
12.   Store encrypted file blocks I_{DB} at untrusted storage;
13. **Enclave.Insert**(v, N)**:**
14. **begin**
15.   **for** each value v **do**
16.   **if** $N.value ==$ "*null*" **then**
17. $N[v] = \{c||t\}$;
18.   Add new node $N[v]$ to I_{SGX};
19.   **else if** $N.value < v$ **then**
20.   Run **Insert**(v, N_{right});
21.   **else if** $N.value > v$ **then**
22.   Run **Insert**(v, N_{left});
23.   Store range-based indexes I_{SGX} at trusted enclave;

Algorithm 2: *Search*: Secure batch query protocol

Input: Private keys $\{k_1, k_2\}$, secure PRFs $\{F, G\}$, enclave cache Q_{SGX}, indexes $\{I_{\text{SGX}}, I_{\text{DB}}\}$, query domain $[q_L, q_R]$, query value v, and order conditions $cmp \in \{\geq, \leq\}$.
Output: Plaintext file blocks.

1. **Client.Token**(v, cmp, $[q_L, q_R]$)**:**
2. **begin**
3. $k_0 \leftarrow F(k_1, s)$, $s++$;
4. $T \leftarrow Enc(k_0, v||cmp||[q_L, q_R])$;
5.   Send query token T to server-side enclave;
6. **Enclave.Search**(T, I_{SGX}, I_{DB})**:**
7. **begin**
8. $s++, k_0 \leftarrow F(k_1, s)$;
9. $\{v||cmp||[q_L, q_R]\} \leftarrow Dec(k_0, T)$;
10.   Find all range-match nodes $\{N[v_1], .., N[v_n]\}$ in I_{SGX};
11.   **for** each matched node $N[v_i]$, $i \in \{1, n\}$ **do**
12. $\{c_i||t_i\} \leftarrow N[v_i], c = 0$;
13.   **while** $c \leq c_i$ **do**
14. $L \leftarrow G(k_1, v_i||c||t_i)$, $c++$;
15.   **for** each label L_j, $j \in \{q_L, q_R\}$ **do**
16.   **if** $L_j \in Q_{\text{SGX}}$ **then**
17. $\{V_j, \gamma_j\} \leftarrow Q_{\text{SGX}}[L_j]$;
18. $\gamma_j^* \xleftarrow{\$} \{0,1\}^\lambda$;
19. $V_j^* \leftarrow V_j \oplus G(k_2, \gamma_j) \oplus G(k_2, \gamma_j^*)$;
20.   Return $\{V_j^*, \gamma_j^*\}$ to the client;
21.   **else if** $L_j \in I_{\text{DB}}$ **then**
22.   Fetch $\{V_j, \gamma_j\}$ via **Enclave.Fetch**(L_j, I_{DB});
23. $\gamma_j^* \xleftarrow{\$} \{0,1\}^\lambda$;
24. $V_j^* \leftarrow V_j \oplus G(k_2, \gamma_j) \oplus G(k_2, \gamma_j^*)$;
25.   Return $\{V_j^*, \gamma_j^*\}$ to the client;
26.   Return total size $R = Enc(k_0, \sum_{i=1}^{n} c_i)$ to client;
27. **Client.Decrypt**(R, V_j^*, γ_j^*)**:**
28. **begin**
29. $\sum_{i=1}^{n} c_i \leftarrow Dec(k_0, R)$;
30.   Initialize a queue Q_{result} with size $\sum_{i=1}^{n} c_i$;
31.   **for** each return result (V_j^*, γ_j^*), $j \in \{q_L, q_R\}$ **do**
32. $Q_{\text{result}}[j] = G(k_2, \gamma_j^*) \oplus V_j^*$;
33.   Get file blocks from Q_{result};

re-encrypted (previous) results are cached inside the enclave and later swapped with the external storage for fetching query results. Since the cached results are shuffled and re-encrypted before storing outside the enclave, an attacker cannot distinguish the result association among different range queries. Thus, the results co-occurrence pattern can be protected with our caching method. The detailed index building procedures and range query protocols are conducted in the next section.

B. Hybrid Index Construction

Based on the above design rationale, Algorithm 1 presents the building procedure of our encrypted hybrid indexes in detail, which is executed at the client-side. The client first divides the dataset $DB(v)$ of value v into $\beta+1$ blocks, where each block can store fixed p file blocks. Specifically, the client masks multiple file blocks $\{f_1||...||f_p\}$ into one ciphertext block via computing $V = \{f_1||...||f_p\} \oplus G(k_2, \gamma)$, where γ is a random nonce. In our design, the last block is padded, so that the number of items in each block is fixed. Then, the client generates pseudo-random labels $L = G(k_1, v||c||t)$ to index the corresponding encrypted results V, where G is a secure PRF, c is a self-incremental counter, and t is an index state that is used for tracking the number of times value v has been queried before. The purpose of introducing the state t is to prevent the storage server from knowing that updated indexed blocks are associated with the same value v. After secure data transformation, the client adds these encrypted items $\{L, V, \gamma\}$ to the storage I_{DB} outside the enclave.

To retrieve these L-V pairs via range queries, we leverage a hardware enclave to maintain the range-based index I_{SGX}, as shown from Line 13 to Line 23 of Algorithm 1. In our design, each binary tree node $N[v]$ maintains an index value

Algorithm 3: $Fetch$: Secure result fetch protocol

Input: Private keys $\{k_1, k_2\}$, secure PRF G, the related range-matched node $N[v_i]$, indexes $\{I_{\text{SGX}}, I_{\text{DB}}\}$, pseudo-random label L_j in I_{DB}.
Output: Re-encrypted indexes $\{L', V', \gamma'\}$.

```
1  Enclave.Fetch(L_j, I_DB):
2  begin
3      {V_j, γ_j} ← I_DB[L_j];
4      Load {L_j, V_j, γ_j} into Q_SGX;
5      Re-encrypt results via Enclave.Rebuild(N[v_i], V_j, γ_j);
6      Randomly select a cached label L† for eviction;
7      {V†, γ†} ← Q_SGX[L†];
8      Store {L†, V†, γ†} at I_DB;
9  Enclave.Rebuild(N[v_i], V_j, γ_j):
10 begin
11     t_i ← N[v_i], t'_i = t_i + 1;  /* update states */
12     Add state t'_i to the node N[v_i];
13     L'_j ← G(k_1, v_i||j||t'_i), γ'_j ←$ {0,1}^λ;
14     V'_j ← V_j ⊕ G(k_2, γ_j) ⊕ G(k_2, γ'_j);
15     Cache {L'_j, V'_j, γ'_j} inside Q_SGX;
```

v, the number of L-V pairs c, and its current index state t, which can later be used for the enclave to generate pseudo-random labels. With this well-ordered index, the enclave can efficiently conduct range queries via binary search. Finally, the hybrid indexes are uploaded to the server. That is, the range-based indexes I_{SGX} and the volume-hiding cipher blocks I_{DB} are securely deployed at the trusted enclave and server-side storage, respectively.

C. Batch Query Protocol

The secure batch query protocol following the hybrid index construction is presented in Algorithm 2. Given a query value v and order condition cmp, the client can retrieve a fixed number of search results on the matching condition during each batch query. The client firstly generates the query token $T = Enc(k_0, v||cmp||[q_L, q_R])$, where k_0 is a shared session key[2] and $[q_L, q_R]$ is the query domain to determine how many results should be returned per query. Then, the enclave can decrypt the query token and generates the query target labels $\{L_{q_L}, .., L_{q_R}\}$ to fetch the encrypted file blocks. Specifically, if the label is cached inside the Q_{SGX}, the enclave considers that the target results have been queried before, and returns previously cached results together with a one-time mask to the client. Otherwise, the enclave needs to fetch encrypted file blocks from I_{DB} via Algorithm. 3.

For obfuscating the results co-occurrence pattern, each access item should be cached inside the Q_{SGX} and re-encrypted by the enclave, as shown in Algorithm. 3. Specifically, the enclave randomly selects previous cached items inside the Q_{SGX} and swaps them with external target results in I_{DB}

[2]The secure channel is implemented via the Intel SGX SSL protocol.

Algorithm 4: $Update$: Secure data insertion protocol

Input: Private keys $\{k_1, k_2\}$, secure PRF G, indexes $\{I_{\text{SGX}}, I_{\text{DB}}\}$, and newly add file blocks f_{new} for the index value v.
Output: Updated hybrid indexes $\{I_{\text{SGX}}, I_{\text{DB}}\}$.

```
1  Client.AddToken(v, f_new):
2  begin
3      γ ←$ {0,1}^λ, T_add ← Enc(k_0, γ||v||f_new);
4      Send add token T_add to server-side enclave;
5  Enclave.Add(T_add, N, I_SGX, I_DB):
6  {γ||v||f_new} ← Dec(k_0, T_add);
7  begin
8      if N.value == v then
9          {c||t} ← N[v], c' = c + 1;
10         Update c' in the node N[v];
11         L_new ← G(k_1, v||c'||t);
12         V_new ← {f_new} ⊕ G(k_2, γ);
       /* Pad this block to p items if needed */
13     else if N.value < v then
14         Run Add(T_add, N_right, I_SGX, I_DB);
15     else if N.value > v then
16         Run Add(T_add, N_left, I_SGX, I_DB);
17     else
18         Add new node N[v] = {0||0} to I_SGX;
19         L_new ← G(k_1, v||0||0);
20         V_new ← {f_new} ⊕ G(k_2, γ);
21     Add new indexes {L_new, V_new, γ} to I_DB;
```

for cached eviction. Meanwhile, each returned result is re-encrypted with a updated state t'_i and a new nonce γ'_j. Finally, the re-encrypted results $\{L'_j, V'_j, \gamma'_j\}$ are cached inside Q_{SGX}.

After receiving encrypted results $\{V^*, \gamma^*\}$, the client firstly decrypts the total result size R and generates a queue Q_{result} to maintain file blocks. Then it unmasks each block ciphertext V^* via computing $\{f_1||...||f_p\} \leftarrow G(k_2, \gamma^*) \oplus V^*$, and stores the corresponding file blocks $\{f_1||...||f_p\}$ to the queue Q_{result}. In our design, the total result size is returned, which can help the client to adaptively adjust the response size for the next query. Thus, after a sequence of batch queries, the remaining file blocks can be retrieved from the server.

D. Dynamic Update Protocol

Following existing works on secure databases [4], we consider that it is critical to quickly index newly generated data and make them promptly available for search and utilization. To this end, we propose the update protocol in Algorithm 4. To insert a new file block f_{new} with an index value v, the client first sends the update request T_{add} to the enclave. After decrypting the request, the enclave searches the index value v with I_{SGX} and updates the corresponding counter c. After that, it generates the newly-added index items $\{L_{\text{new}}, V_{\text{new}}\}$ with the incremented counter c' and the new random nonce

γ. Since new index items are generated from the latest state, the server cannot learn whether the newly-added items contain a value that was searched before. Therefore, our design also achieves the security notion of forward-privacy [6], [50], i.e., keeping newly updated indexes unlinkable to previous search results. Meanwhile, new indexes are also padded to the fixed length, thus the volume-hiding property is preserved. Finally, these newly added indexes $\{L_{\text{new}}, V_{\text{new}}, \gamma\}$are inserted to the storage I_{DB}.

The proposed refresh scheme caches each accessed item inside the enclave and uses new random masks for index bulk updates. For security consideration, the size of the cache space should be much larger than the index item, i.e., $|Q_{\text{SGX}}| >> \sum_{j=q_L}^{q_R}(|L_j|+|V_j|+|\gamma_j|)$. This design has two features. First, it ensures that multiple query results can be cached together to prevent attackers from tracking the association between tokens and cached results. Second, the correct tracking probability is less than $\sum_{j=q_L}^{q_R}(|L_j|+|V_j|+|\gamma_j|)/|Q_{\text{SGX}}|$. By leveraging enclave caching, we carefully integrate the batch query protocol with this refresh scheme to achieve results co-occurrence pattern obfuscation.

Side-channel elimination: Our hybrid indexes prevent against adversaries who have a snapshot of the encrypted data. However, an adversary could potentially infer the index construction inside the enclave by tracing accessed enclave pages and branches (also known as side-channel attacks [55]).

As acknowledged in prior SGX-enabled systems [41], [43], [44], we note that the side-channel attacks are application dependent, and many countermeasures addressing certain types of attacks have been proposed. In our design, we propose to load I_{SGX} indexes into the enclave and conduct the range query operations. If the range-based indexes I_{SGX} can be stored at a single enclave page, this side-channel leakage is not exploitable because memory accesses within the same page are indistinguishable. However, since the binary tree size may exceed the page memory available in the enclave, enclave pages have to be swapped in and out in this case. In this worst case, our design could suffer from the page-level side-channel attacks. To eliminate such leakage, a recent oblivious-access data structure [56] can readily be adapted to our range-based indexes inside the enclave. The high-level idea is to use multiple random nodes to store the same index value instead of using a single node. Then the enclave can use a breadth-first search method with dummy operations to obfuscate the search path leakage. As a performance tradeoff, the query throughput would be downgrade due to the additional computation cost for dummy operations. By using this fine-grained implementation, we can achieve data-independent accesses and thwart the side-channel leakages.

V. COMPLEXITY AND SECURITY ANALYSIS

In this section, we conduct a formal security analysis for our proposed scheme. We first define leakages during protocols and quantify security guarantees following the adopted primitives. We prove that our scheme can achieve properties of volume-hiding and results co-occurrence obfuscation. Besides, we provide a rigorous complexity analysis for our scheme.

A. Space Complexity

As mentioned, directly applying the naive padding approach would consume much space cost because the volume size of items are all set to the maximum index length $L_{\max} = max_{v\in\mathcal{V}}|DB(v)|$. Thus, the space complexity of this approach is $O(m \cdot L_{\max})$, where m is the number of index items. In contrast, our design can use flexible padding blocks P_i to achieve optimal space complexity: $O(m \cdot \lambda + \sum_{i=1}^{m} P_i)$, where $\lambda > 0$ and $P_i \in \{0, .., L_{\max} - 1\}$. From a high level point of view, our design follows the same intuition in [38] to maintain encrypted results in a volume-hiding manner. Thus, we follow [38] to provide a formal space analysis for our proposed design. Without lose generality, we assume that the padding block P follows a uniform distribution. Then the storage overhead of our design can be bounded as follows.

***Theorem* 1.** *The size of hybrid index construction is at most*

$$m \cdot (\lambda + (L_{\max} - 1) \cdot (\frac{1}{2} + \sqrt{\frac{\ln(1/\varphi)}{2m}}))$$

with probability at least $1 - \varphi$.

Proof. Let P_i denotes the additional padding block for the i-th index item, where $i \in \{1, m\}$. As P_i is the uniform distribution over $\{0, L_{\max} - 1\}$, the additional storage overhead is $P = \sum_{i=1}^{m} P_i$ with the expectation $E[P] = m \cdot (L_{\max} - 1)/2$. According to the Hoeffding's second inequality, we have that for all $\sigma \geq 0$,

$$Pr[P \geq m \cdot (\frac{L_{\max} - 1}{2} + \sigma)] \leq exp(\frac{-2 \cdot m \cdot \sigma^2}{(L_{\max} - 1)^2})$$

By setting $\sigma = (L_{\max} - 1) \cdot \sqrt{\ln(1/\varphi)/2m}$, we have that

$$Pr[P \geq m \cdot (L_{\max} - 1) \cdot (\frac{1}{2} + \sqrt{\frac{\ln(1/\varphi)}{2m}})] \leq \varphi$$

In general, we refer to the parameter λ as the record length, so that the total storage overhead is $S = P + m \cdot \lambda$, which follows the Theorem 1 that

$$Pr[S \leq m \cdot (\lambda + (L_{\max} - 1) \cdot (\frac{1}{2} + \sqrt{\frac{\ln(1/\varphi)}{2m}}))] \geq 1 - \varphi$$

□

According to Theorem 1, we get that the storage overhead is at most $\alpha \cdot m \cdot (L_{\max} - 1)$ by setting $\lambda = (L_{\max} - 1) \cdot (2\alpha - 1)/4$ and $\varphi_1 = exp(-m \cdot (2\alpha - 1)^2/8)$. Since that $\lambda > 0$, we can get $\alpha > 1/2$; Meanwhile, only if $\alpha < 1$, the storage overhead of our proposed design can provide a better storage complexity than naive padding approach, i.e., $m \cdot L_{\max}$. Therefore, our design can achieve optimal space complexity when letting $1/2 < \alpha < 1$ with $1 - \varphi_1$ probability.

B. Security Analysis

In this subsection, we provide a formal security analysis to demonstrate the security guarantees of our proposed scheme. Recall that we uniquely bridge the trusted hardware techniques and volume-hiding data structure to build encrypted hybrid indexes. The security of the hardware enclave enables the client to perform range queries while protecting the order leakage. Meanwhile, the bucketization-based padding method and batch query protocols ensure that the server cannot learn the exact result size in each round of returning data. Following the security notion for searchable encryption scheme, we present a rigorous security analysis to demonstrate that HybrIDX achieves strong protection on query requests and outsourced files.

We first define the leakage functions of our design, including the partial access set pattern (A_q), cache eviction set (E_q), and eviction history set (H_q). Explicitly, A_q indicates the partial access set from I_{DB} returned for a query q. E_q leaks the eviction set from the enclave cache Q_{SGX} for q and H_q tracks the eviction history set. In our design, we use a one-time key to encrypt query requests so that the adversary cannot trace the dependencies between each query and results. It can only learn repeated queries by observing whether results are cached inside the enclave. Formally, we define the leakage functions as below:

$$A_q = \{L|(L, v) \in \text{SGX}_{\text{out}}\},$$
$$E_q = \{L|(L, v) \in \text{SGX}_{\text{in}}\},$$
$$H_q = \{q|(L, v) \in A_q \text{ or } (L, v) \in E_q, q \in Q\}.$$

where Q is the query list. In particular, SGX_{in} denotes that results are cached inside the enclave and SGX_{out} denotes results are fetched from I_{DB}. We define the setup leakage $\mathcal{L}^{\text{step}}$ for a given $DB(v)$:

$$\mathcal{L}^{\text{Step}} = (m, \langle |L|, |V| \rangle),$$

where m is the number of encrypted L-V pairs, and $\langle |L|, |V| \rangle$ are ciphertext lengths of labels and file blocks. When a client sends a search request, the view of an adversary is defined in the leakage $\mathcal{L}^{\text{Srch}}$ as:

$$\mathcal{L}^{\text{Srch}} = (A_q, E_q, H_q),$$

where the leakage contains access items and fixed number of encrypted results with padding. During file insertion, the leakage $\mathcal{L}^{\text{Updt}}$ captured by an adversary is defined as:

$$\mathcal{L}^{\text{Updt}} = (op, \{L, \mu\}),$$

where op represents update operations and $\{L, \mu\}$ is the set of updated items with the number μ. Note that our design leverages fresh random masks generated from the latest state to encrypt the newly added values and previous query results. Thus, the adversary cannot learn the association between the newly added indexes and any query made in the past [6], [50]. Following the simulation-based security definition in [2]–[4], we give the formal security definition:

***Definition* 1.** *Let $\Omega = (\mathsf{Build}, \mathsf{Search}, \mathsf{Update})$ be the secure index-based scheme of encrypted range queries, λ be the security parameter, and let $\mathcal{L}^{\text{Step}}$, $\mathcal{L}^{\text{Srch}}$ and $\mathcal{L}^{\text{Updt}}$ be the leakage functions. We define the following probabilistic experiments $\mathbf{Real}_{\Omega,\mathcal{A}}(1^\lambda)$ and $\mathbf{Ideal}_{\Omega,\mathcal{A},\mathcal{S}}(1^\lambda)$ with a probabilistic polynomial time (PPT) adversary $\mathcal{A}$ and a PPT simulator $\mathcal{S}$:*

$\mathbf{Real}_{\Omega,\mathcal{A}}(1^\lambda)$*: $\mathcal{A}$ selects a dataset $DB(v)$ and asks the client to generate the index and ciphertexts via the* Build *and* Update *algorithms. Then $\mathcal{A}$ launches a polynomial number of batch queries and asks the client for the tokens and resulting ciphertexts via the algorithm* Search. *Finally, $\mathcal{A}$ outputs a bit as the output.*

$\mathbf{Ideal}_{\Omega,\mathcal{A},\mathcal{S}}(1^\lambda)$*: $\mathcal{A}$ selects a dataset $DB(v)$, and $\mathcal{S}$ simulates an index and ciphertexts for $\mathcal{A}$ based on $\mathcal{L}^{\text{Step}}$. From $\mathcal{L}^{\text{Updt}}$, $\mathcal{S}$ performs the update operations. Then $\mathcal{A}$ adaptively performs a polynomial number of queries. From the leakage $\mathcal{L}^{\text{Srch}}$ in each batch query, $\mathcal{S}$ simulates tokens and ciphertexts, which are processed over the simulated index. Finally, $\mathcal{A}$ outputs a bit as the output.*

Let $\mathcal{L} = (\mathcal{L}^{\text{Step}}, \mathcal{L}^{\text{Srch}}, \mathcal{L}^{\text{Updt}})$, Ω is adaptively $\mathcal{L}$-secure for all PPT adversaries $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that: $Pr[\mathbf{Real}_{\Omega,\mathcal{A}}(1^\lambda) = 1] - Pr[\mathbf{Ideal}_{\Omega,\mathcal{A},\mathcal{S}}(1^\lambda) = 1] \leq negl(\lambda)$, where $negl(\lambda)$ is a negligible function in λ.

Lemma 1. *Ω is an adaptive $\mathcal{L}$-secure scheme under the random-oracle model if F, G are secure PRFs.*

***Definition* 2.** *Let S_0=$(v_0, q(v_0))$ and S_1=$(v_1, q(v_1))$ be two signatures from adversary $\mathcal{A}$ where $q(v)$ is the length of result size associated with v. We define the game $\mathbf{Game}^{\mu}_{\mathcal{A},\mathcal{L}}(m, q)$ for leakage function $\mathcal{L} = (\mathcal{L}^{\text{Step}}, \mathcal{L}^{\text{Srch}})$ with m total records, fixed volume q, and $\mu \in \{0, 1\}$:*

$\mathbf{Game}^{\mu}_{\mathcal{A},\mathcal{L}}(m, q)$*: $\mathcal{A}$ generates two signatures S_0, S_1 and asks the client to generate the index with the signature S_μ. The client sends $\mathcal{L}^{\text{Step}}$ to the adversary $\mathcal{A}$. The $\mathcal{A}$ launches a polynomial number of batch queries. For each query, the client computes $\mathcal{L}^{\text{Srch}}$. Finally, $\mathcal{A}$ outputs a bit as the output.*

Let $Pr^{\mu}_{\mathcal{A},\mathcal{L}}(m, q)$ be the probability that $\mathcal{A}$ outputs 1 when playing game $\mathbf{Game}^{\mu}_{\mathcal{A},\mathcal{L}}(m, q)$. A leakage function is volume-hiding if and only if for all adversaries $\mathcal{A}$ and for all queries $q(v) \in \{1, m\}$: $Pr^{0}_{\mathcal{A},\mathcal{L}}(m, q)$=$Pr^{1}_{\mathcal{A},\mathcal{L}}(m, q)$.

Lemma 2. *Leakage function $\mathcal{L}$=$(\mathcal{L}^{\text{Step}}, \mathcal{L}^{\text{Srch}})$ of our query scheme is volume-hiding.*

Therefore, we can obtain the main theorem of this section by combining the lemma 1 and lemma 2 as follows:

***Theorem* 2.** *Ω is $\mathcal{L}$-secure volume-hiding scheme under the random-oracle model if F, G are secure PRFs.*

Proof of lemma 1. Our proposed design is essentially based on the SE scheme in [38], except that we employ the hardware enclave to realize the range query protocol. More precisely, it leverages the enclave to re-encrypt the accessed indexes with fresh random masks for result set obfuscation. Besides, query tokens are encrypted with a one-time key for preserving the security strength. We prove the existence of a simulator $\mathcal{S}$ such that for all polynomial-time adversaries $\mathcal{A}$, the output

of $\mathbf{Real}_{\Omega,\mathcal{A}}(1^\lambda)$ and $\mathbf{Ideal}_{\Omega,\mathcal{A},\mathcal{S}}(1^\lambda)$ are computationally indistinguishable. Given $\mathcal{L}^{\mathrm{Step}}$, the simulator $\mathcal{S}$ can generate the simulated encrypted indexes, which is indistinguishable from the real one. It initializes a dictionary with m items, where each item contains $|L|$-bit and $|V|$-bit random strings as a label-value pair. From $\mathcal{L}^{\mathrm{Srch}}$, $\mathcal{S}$ can simulate the first query and its corresponding results. In particular, for each token in the query, it generates a random string as a simulated token in the simulated indexes. Then, $\mathcal{S}$ operates a random oracle to point at randomly selected items in the dictionary and reveals the same simulated label to match the real ones observed from the leakage $\mathcal{L}^{\mathrm{Srch}}$. The identical number of simulated records are considered as the simulated results. The simulation can be extended to a number of adaptive queries. $\mathcal{S}$ records the appeared results by observing whether results are cached inside the enclave $\mathrm{SGX}_{\mathrm{in}}$. The cached results can be simulated with fresh strings and stored at the simulated indexes. When adding new indexes, the results can also be simulated according to the $\mathcal{L}^{\mathrm{Updt}}$. Due to the pseudo-randomness of PRF and the semantic security of one-time key encryption, $\mathcal{A}$ should not be able to distinguish between the real interactions and the simulated ones. The outputs of experiments $\mathbf{Real}_{\Omega,\mathcal{A}}(1^\lambda)$ and $\mathbf{Ideal}_{\Omega,\mathcal{A},\mathcal{S}}(1^\lambda)$ are computationally indistinguishable. This completes the proof.

□

Proof of lemma 2. The objective is to prove that the adversary $\mathcal{A}$ cannot learn the exact number of matched files associated with any values and only leak the fixed volume of a query. In particular, we consider any two signatures S_0, S_1 as defined in Definition 2. We note that the leakage $\mathcal{L}^{\mathrm{Step}}$ for I_{DB} consists of m items, and $\mathcal{L}^{\mathrm{Srch}}$ consists of only fixed volume q. The leakage functions of $\mathcal{L}^{\mathrm{Step}}$ and $\mathcal{L}^{\mathrm{Srch}}$ are identical for both signatures. As a result, the adversary $\mathcal{A}$ cannot distinguish any two signatures, i.e., $Pr^0_{\mathcal{A},\mathcal{L}}(m,q)$=$Pr^1_{\mathcal{A},\mathcal{L}}(m,q)$. □

VI. Experimental Evaluation

A. Prototype Implementation

To assess the performance of HybrIDX, we implement a prototype in C++ and deploy it to a Ubuntu server (16.04). We conduct a thorough experimental evaluation on the SGX-enabled server with an Intel(R) Core(TM) i7-7700 processor (3.6 GHz) and 16GB RAM. The SGX physical memory is set to 128MB due to the hardware constraints. To evaluate the query efficiency when handling a large-scale dataset, we pre-insert 160K data records and randomly assign them to 1K index values. We generate a Redis (v5.0.3) cluster to maintain the encrypted volume-hiding storage. Our system uses Apache Thrift (v0.9.2) to implement the remote procedure call (RPC) between the client and servers. For cryptographic primitives, we use Intel SGX SSL and OpenSSL (v1.1.0g) to implement the symmetric encryption via AES-128 and the pseudo-random function via HMAC-256. Overall, our implementation consists of 16885 lines of code (LOC).

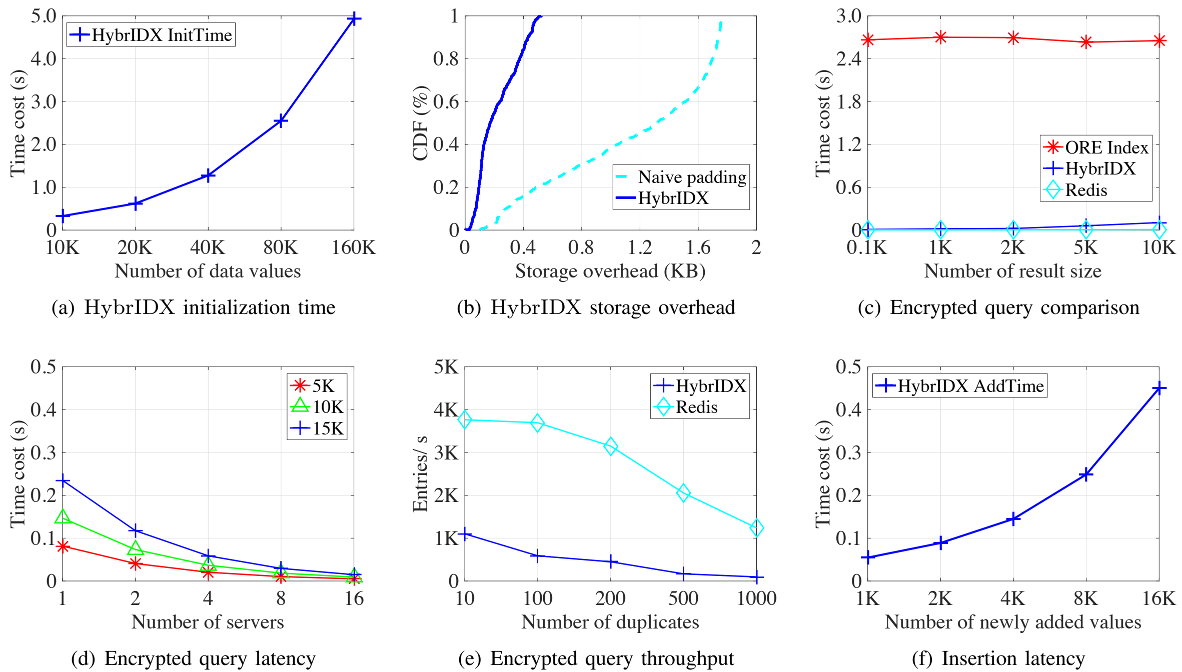
B. Performance Evaluation

Our experimental evaluation targets on testing the practicality of the proposed hybrid index design, including initialization time, memory cost, range query efficiency, and throughput. We also compare our hybrid index design with pure cryptographic solutions and the plaintext version in this section.

Evaluation on hybrid indexes: We first evaluate the total time cost of building the encrypted hybrid index when using algorithm 1. As shown in Fig. 3(a), we observe that the time cost of system initialization increases linearly with the number of data records. Specifically, the client only takes less than 5s to finish the build procedure when encrypting 160K records. In addition, we emphasize that this initialization process is a one-time cost, and the server can efficiently add new indexes without influencing the other built indexes.

In Fig. 3(b), we investigate the CDF of the padding cost under different padding schemes. For the naive padding scheme, each index item is padded to the maximum length of the query results. Thus, the additional padding cost for each index item is $(L_{max} - |DB(v)|)$, where L_{max} is the maximum result size. As we can see from Fig. 3(b), such a simple approach would introduce a huge storage overhead. For instance, the introduced space cost for over 70% of index items is more than 0.8 KB, of which approximately 20% of index items are more than 1.8 KB. Regarding the padding cost of the volume-hiding scheme, it depends on the bit length of each block p. Specifically, each padded block introduces $(p - |DB(v)| \mod p)$ padding cost. Besides, it also generates new labels ($\lfloor \frac{|DB(v)|}{p} \rfloor \times 256$ bits) to index the resulting block ciphers. As shown in Fig. 3(b), the introduced overhead for over 80% volume-hiding indexes are less than 0.4 KB. The results confirm that our hybrid index framework can achieve optimized storage consumption.

Evaluation on query efficiency: To assess the practicality of our query protocol, we further measure the range query latency over encrypted hybrid indexes. The total latency consists of the time cost for cryptographic operations at the server-side and the time cost for secure token generation at the client-side. In particular, Fig. 3(c) measures the performance comparison between our hybrid index design with the ORE-based indexes in [18] and a plaintext version. In our proposed design, the enclave conducts range queries over its in-memory index via binary search, and then generates the corresponding pseudo-random labels based on the range query results to fetch the encrypted file blocks. Since the in-memory index is well-ordered, so that the query performance can reach the processing speed similar to the plaintext version. In contrast, an ORE-based index does not directly present order relations and thus a sorted index cannot be used. It requires a linear scan on the whole ORE-based indexes to evaluate the order results. Specifically, when the number of matched results is 10K, the query latency of our design is around 0.14s, which is almost 18× faster compared to the ORE-based scheme. The evaluation result confirms that HybrIDX benefits from the hardware-assisted index framework and can support secure range queries efficiently.



(a) HybrIDX initialization time (b) HybrIDX storage overhead (c) Encrypted query comparison

(d) Encrypted query latency (e) Encrypted query throughput (f) Insertion latency

Fig. 3: Evaluation for HybrIDX performance.

To evaluate the scalability of our index design, we accordingly evaluate the query latency when using multiple data servers. As shown in Fig. 3(d), we can find that as the number of data servers increases, the query latency that returns a fixed number of search results is reduced dramatically in a similar proportion. When the number of matched records is 15K, the query latency with 8 servers is around 0.29s, which is roughly half of the latency with 4 servers. The results confirm that HybrIDX performs satisfactorily at scale and can effectively handle queries in parallel.

To gain a deeper understanding of the query performance, we also compare the query throughput with a plaintext Redis version in Fig. 3(e) when varying the number of duplicates per indexed value. In our design, the file blocks for an indexed value are randomly mapped into multiple block ciphertexts. During each query, the enclave increments a counter to generate the labels for fetching these encrypted blocks. Thus, the number of labels corresponding to a query value increases with the number of duplicates. This volume-hiding design can protect the relations between indexed values and different file blocks, but it introduces an additional computational cost of cryptographic operations. In contrast, a plaintext index can map duplicates to a single reference and fetches these records all in a scan. As shown in Fig. 3(e), we find that both HybrIDX and Redis indexes follow a similar downward trend as the number of duplicates increases. The throughput of HybrIDX decreases from about 587 items/s to 166 items/s as the number of duplicates increases from 100 to 500. When all results of each index value can be mapped to a single block, the query throughput can achieve up to 1.1K items per second, which is fast enough for practical applications. According to evaluation results, our hybrid index design is shown to be capable of providing a flexible balance on data security, space utilization, and query efficiency.

Recall that our proposed design also supports incremental updates for newly added files. In Fig. 3(f), we accordingly evaluate the incremental scalability by measuring the time cost for index item insertion. Note that the latency of the update processing includes the time cost of index traversal, state update, and new index generation. Nonetheless, the update latency is still slightly faster than the time cost of system initialization. This is because the enclave index is already sorted so that it can efficiently locate the index node that needs to be updated. As shown in Fig. 3(f), when the number of newly added files is 16K, it just takes 0.45s to insert these newly added index items.

VII. CONCLUSION

In this paper, we introduce a new hybrid index framework, called HybrIDX, that enables volume-hiding range queries over encrypted data. HybrIDX is designed to carefully combine the trusted hardware techniques and volume-hiding structures while mitigating individual disadvantages of these technologies. Compared with previous designs, HybrIDX simultaneously enables a dramatic speed-up over pure cryptographic designs and protection from volume attacks. Besides, an enclave caching method is proposed for secure results obfuscation that is compatible with our hybrid index framework. Extensive experimental results have shown that the proposed design is provably secure and highly efficient.

Acknowledgment

This work was supported by Alibaba-Zhejiang University Joint Research Institute of Frontier Technologies, Renyun Digital, the Research Grants Council of Hong Kong under Grants CityU 11202419, CityU 11212717, CityU 9042819, and C1008-16G, and the National Natural Science Foundation of China under Grants 61732022, 61772236, and 61572412.

References

[1] E.-J. Goh, "Secure indexes," *in IACR Cryptology ePrint Archive*, vol. 2003, p. 216, 2003.

[2] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: Improved definitions and efficient constructions," *Journal of Computer Security*, vol. 19, no. 5, pp. 895–934, 2011.

[3] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *Proc. of ACM CCS*, 2012.

[4] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Dynamic searchable encryption in very large databases: Data structures and implementation," in *Proc. of NDSS*, 2014.

[5] M. Chase and S. Kamara, "Structured encryption and controlled disclosure," in *Proc. of ASIACRYPT*, 2010.

[6] Y. Guo, C. Zhang, and X. Jia, "Verifiable and forward-secure encrypted search using blockchain techniques," in *Proc. of IEEE ICC*, 2020.

[7] Q. Wang, Y. Guo, H. Huang, and X. Jia, "Multi-user forward secure dynamic searchable symmetric encryption," in *Proc. of NSS*, 2018.

[8] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: protecting confidentiality with encrypted query processing," in *Proc. of ACM SOSP*, 2011.

[9] Y. Guo, C. Wang, X. Yuan, and X. Jia, "Enabling privacy-preserving header matching for outsourced middleboxes," in *Proc. of IWQoS*, 2018.

[10] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner, "Rich queries on encrypted data: Beyond exact matches," in *Proc. of ESORICS*, 2015.

[11] "Bigquery project." Online at https://cloud.google.com/bigquery/, 2014.

[12] Y. Guo, X. Yuan, X. Wang, C. Wang, B. Li, and X. Jia, "Enabling encrypted rich queries in distributed key-value stores," *IEEE TPDS*, vol. 30, no. 7, pp. 1283–1297, 2018.

[13] V. Pappas, B. Vo, F. Krell, S. Choi, V. Kolesnikov, A. Keromytis, and T. Malkin, "Blind Seer: A Scalable Private DBMS," in *Proc. of IEEE S&P*, 2014.

[14] R. Poddar, T. Boelter, and R. A. Popa, "Arx: A strongly encrypted database system," in *Proc. of VLDB Endowment*, 2019.

[15] B. Fuller, M. Varia, A. Yerukhimovich, E. Shen, A. Hamlin, V. Gadepally, R. Shay, J. D. Mitchell, and R. K. Cunningham, "Sok: Cryptographically protected database search," in *Proc. of IEEE S&P*, 2017.

[16] S. Wu, Q. Li, G. Li, D. Yuan, X. Yuan, and C. Wang, "Servedb: Secure, verifiable, and efficient range queries on outsourced database," in *Proc. of ACM ICDE*, 2019.

[17] X. Yuan, Y. Guo, X. Wang, C. Wang, B. Li, and X. Jia, "Enckv: An encrypted key-value store with rich queries," in *Proc.of ACM AsiaCCS*, 2017.

[18] K. Lewi and D. J. Wu, "Order-Revealing Encryption: New Constructions, Applications, and Lower Bounds," in *Proc.of ACM CCS*, 2016.

[19] D. Cash, F.-H. Liu, A. ONeill, M. Zhandry, and C. Zhang, "Parameter-hiding order revealing encryption," in *Proc. of ASIACRYPT*, 2018.

[20] N. Chenette, K. Lewi, S. A. Weis, and D. J. Wu, "Practical Order-Revealing Encryption with Limited Leakage," in *Proc. of FSE*, 2016.

[21] C. Mavroforakis, N. Chenette, A. O'Neill, G. Kollios, and R. Canetti., "Modular order-preserving encryption," in *Proc. of SIGMOD*, 2015.

[22] D. S. Roche, D. Apon, S. G. Choi, and A. Yerukhimovich, "Pope: Partial order preserving encoding," in *Proc. of ACM SIGSAC*, 2016.

[23] A. Boldyreva, N. Chenette, and A. O'Neill, "Order-preserving encryption revisited: Improved security analysis and alternative solutions," in *Proc. of CRYPTO*, 2011.

[24] R. A. Popa, F. H. Li, and N. Zeldovich, "An ideal-security protocol for order-preserving encoding," in *Proc. of IEEE S& P*, 2013.

[25] F. Kerschbaum, "Frequency-hiding order-preserving encryption," in *Proc. of ACM CCS*, 2015.

[26] D. Bogatov, G. Kollios, and L. Reyzin., "A comparative evaluation of order-revealing encryption schemes and secure range-query protocols," in *Proc. of VLDB Endowment*, 2019.

[27] Y. Jing, Y. Zheng, Y. Guo, and C. Wang, "Sok: A systematic study of attacks in efficient encrypted cloud data search," in *Proc. of SBC*, 2020.

[28] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill, "Generic attacks on secure outsourced databases," in *Proc. of ACM CCS*, 2016.

[29] M.-S. Lacharit and B. Minaud, "Improved reconstruction attacks on encrypted data using range query leakage," in *Proc. of IEEE S&P*, 2018.

[30] Z. Gui, O. Johnson, and B. Warinschi, "Encrypted databases: New volume attacks against range queries," in *Proc. of ACM SIGSAC*, 2019.

[31] I. Demertzis, D. Papadopoulos, C. Papamanthou, and S. Shintre, "Seal: Attack mitigation for encrypted databases via adjustable leakage," in *USENIX Security*, 2020.

[32] L. Blackstone, S. Kamara, and T. Moataz, "Revisiting leakage abuse attacks," in *Proc. of NDSS*, 2020.

[33] F. B. Durak, T. M. DuBuisson, and D. Cash, "What else is revealed by order-revealing encryption?" in *Proc. of ACM CCS*, 2016.

[34] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *Proc. of ACM CCS*, 2015.

[35] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart, "Leakage-abuse attacks against order-revealing encryption," in *Proc. of IEEE S&P*, 2017.

[36] M. Naveed, S. Kamara, and C. V. Wright, "Inference attacks on property-preserving encrypted databases," in *Proc. of ACM CCS*, 2015.

[37] S. Kamara, T. Moataz, and O. Ohrimenko, "Structured encryption and leakage suppression," in *Proc. of CRYPTO*, 2018.

[38] S. Kamara and T. Moataz, "Computationally volume-hiding structured encryption," in *Proc. of EUROCRYPT*, 2019.

[39] S. Patel, G. Persiano, K. Yeo, and M. Yung, "Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing," in *Proc. of CCS*, 2019.

[40] A. William and T. Tullis, "Measuring the user experience: collecting, analyzing, and presenting usability metrics," in *Newnes*, 2013.

[41] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *Proc. of IEEE S&P*, 2018.

[42] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A.-R. Sadeghi, "Hardidx: Practical and secure index with sgx," in *Proc. of DBSec*, 2017.

[43] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform," in *Proc. of NSDI*, 2017.

[44] S. Sasy, S. Gorbunov, and C. W. Fletcher, "Zerotrace: Oblivious memory primitives from intel sgx," in *IACR Cryptology ePrint Archive*, 2017.

[45] C. Priebe, K. Vaswani, and M. Costa, "Enclavedb: A secure database using sgx," in *Proc. of IEEE S&P*, 2018.

[46] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proc. of ACM HASP*, 2013.

[47] D. Vinayagamurthy, A. Gribov, and S. Gorbunov, "Stealthdb: a scalable encrypted database with full sql query support," in *Proc. of Privacy Enhancing Technologies*, 2019.

[48] S. Eskandarian and M. Zaharia, "Oblidb: oblivious query processing for secure databases," in *Proc. of VLDB Endowment*, 2019.

[49] H. Duan, C. Wang, X. Yuan, Y. Zhou, Q. Wang, and K. Ren, "Lightbox: Full-stack protected stateful middlebox at lightning speed," in *Proc. of ACM CCS*, 2019.

[50] V. Viet, S. Lai, X. Yuan, S.-F. Sun, S. Nepal, and J. K. Liu, "Accelerating forward and backward private searchable encryption using trusted execution," in *Proc.of ACNS*, 2020.

[51] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: an extremely simple oblivious ram protocol," in *Proc. of ACM CCS*, 2013.

[52] F. Brasser, U. Muller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi, "Software grand exposure: Sgx cache attacks are practical," in *Proc. of WOOT*, 2017.

[53] J. Gotzfried, M. Eckert, S. Schinzel, and T. Muller, "Cache attacks on intel sgx," in *Proc. of EUROSEC*, 2017.

[54] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using sgx to conceal cache attacks," in *Proc. of DIMVA*, 2017.

[55] M.W.Shih, S.Lee, T.Kim, and M.Peinado, "T-sgx:eradicating controlled-channel attacks against enclave programs," in *Proc. of NDSS*, 2017.

[56] X. Wang, K. Nayak, C. Liu, T. H. Chan, E. Shi, E. Stefanov, and Y. Huang, "Oblivious data structures," in *Proc. of CCS*, 2014.