EncKV: An Encrypted Key-value Store with Rich Queries

Xingliang Yuan[†‡], Yu Guo[†], Xinyu Wang[†‡], Cong Wang[†‡], Baochun Li[§], Xiaohua Jia[†]
[†] Department of Computer Science, City University of Hong Kong, China
[‡] City University of Hong Kong Shenzhen Research Institute, China
[§] Department of Electrical and Computer Engineering, University of Toronto, Canada
{xl.y, congwang,csjia}@cityu.edu.hk, {y.guo, xy.w}@my.cityu.edu.hk, bli@ece.toronto.edu

ABSTRACT

Distributed data stores have been rapidly evolving to serve the needs of large-scale applications such as online gaming and real-time targeting. In particular, distributed key-value stores have been widely adopted due to their superior performance. However, these systems do not guarantee to provide strong protection of data confidentiality, and as a result fall short of addressing serious privacy concerns raised from massive data breaches.

In this paper, we introduce EncKV, an encrypted key-value store with secure rich query support. First, EncKV stores encrypted data records with multiple secondary attributes in the form of encrypted key-value pairs. Second, it leverages the latest practical primitives for searching over encrypted data, i.e., searchable symmetric encryption and order-revealing encryption, and provides encrypted indexes with guaranteed security to support exact-match and range-match queries via secondary attributes of data records. Third, it carefully integrates these indexes into a distributed index framework to facilitate secure query processing in parallel. To mitigate recent inference attacks on encrypted database systems, EncKV protects the order information during range queries, and presents an interactive batch query mechanism to further hide the associations across data values on different attributes. We implement an EncKV prototype on a Redis cluster, and conduct an extensive set of performance evaluations on the Amazon EC2 public cloud platform. Our results show that EncKV effectively preserves the efficiency and scalability of plaintext distributed key-value stores.

Keywords

Encrypted Key-value Store; Searchable Encryption; Order-revealing Encryption

1. INTRODUCTION

In the last decade, a new group of distributed storage systems — also known as NoSQL data stores — are rapidly evolving to handle data in large-scale applications, such as online gaming and product recommendation [18,32]. Among others, key-value (KV) stores are considered as one of the most popular types of distributed data stores, due to their strength in terms of both performance and scalability. Exemplary systems include Redis [29], DynamoDB [9], and RAMCloud [22]. Recent advances on KV stores have further leveraged secondary indexes to enrich their features, i.e., supporting rich queries via secondary attributes other than the primary key [11, 15].

However, privacy concerns are becoming increasingly more serious with large volumes of data stored in distributed KV stores, from the public clouds to private data warehouses, with recent incidents of massive data breaches [13]. Indeed, these KV stores do not provide strong protections of data confidentiality. Conventional mechanisms resort to access control that specifies the access scope at the user or group levels [7], or transparent server-side encryption that asks the servers (not the data owners) to encrypt data [21]. These mechanisms are not able to fully defend against serious threats of stealing data.

Recent work that seeks to protect the data while preserving the query functionality falls into two categories. In the *first* category, generic cryptographic primitives have been designed to enable specific query functions over encrypted data, such as searchable encryption [4, 8, 14] for keyword search, and order-revealing encryption[1] [1, 10, 19] for range search. In the *second* category, encrypted database systems that utilized various primitives to support a wide range of query functions have been implemented. Representative systems include CryptDB [27], BlindSeer [24], Arx [25], and Seabed [23]. Depending on the primitives adopted, they have different trade-offs on security, functionality, and efficiency. Unfortunately, neither category is specifically tailored for distributed KV stores.

To bridge this gap, we start from a very recent encrypted KV store design [33] that preserves advantages of modern KV stores while initiating an encrypted local index framework for efficient queries via secondary attributes of data. The core idea of this framework is to co-locate the encrypted data records and the corresponding encrypted indexes in the same nodes, so as to avoid inter-node interaction during query protocols and facilitate parallel query processing. Unfortunately, this initial framework serves only as a blueprint, as it did not present how to securely support different types of queries over distributed encrypted data records.

ASIA CCS '17, April 02-06, 2017, Abu Dhabi, United Arab Emirates

DOI: http://dx.doi.org/10.1145/3052973.3052977

[1]Early results [1] that only support numeric comparisons are also called order-preserving encryption.

Our objectives in this paper are to address the following two challenges. *First*, appropriate cryptographic primitives need to be carefully chosen to balance security and practicality, and further customized and integrated into the aforementioned index framework. The selected primitives should be both sufficiently efficient for deployment scenarios and provably secure with guaranteed strength. Meanwhile, their integration should not diminish the advantages of the local index framework.

Second, leakage in our system should be minimized. Driven by recent inference attacks [16, 20] and leakage-abuse attacks [3, 12], we follow the latest primitives with the "best-possible" security notion in the literature. Further, we note that implementing rich queries in real systems might reveal auxiliary information which is not considered in generic primitives. For example, the associations between data values on different attributes are sometimes exposed unnecessarily, and have been shown to be exploitable, leading to compromised data confidentiality [10, 20]. Therefore, our proposed query protocols should protect such information.

In this paper, we propose EncKV, an encrypted key-value store with secure rich query support. We first classify common queries of KV stores into two categories, i.e., exact-match queries (keyword search, equality test and counting) and range-match queries (range search and prefix match). To support these two types of queries, EncKV leverages two primitives respectively: searchable symmetric encryption (SSE) [4, 8] and order-revealing encryption (ORE) [6, 19]. For low latency queries, EncKV follows the guideline of the encrypted local index framework given in [33]; that is, the client needs to track the location of each data record when it builds the local encrypted indexes[2] that index the data records on each node respectively.

For exact-match queries, EncKV carefully integrates Cash *et al.*'s elegant and efficient SSE scheme [4] into the local index framework, and customizes it to support exact-match queries via encrypted single or multiple secondary attributes of data. As a result, EncKV's encrypted local indexes hold the security of SSE, and can readily be stored in any KV store back end for easy deployment.

For range-match queries, EncKV utilizes Lewi and Wu's ORE scheme [19], which achieves the "best-possible" security notion for practical ORE. Like exact-match queries, the chosen ORE scheme is firstly customized and integrated into EncKV's index framework. Furthermore, we observe that the order information (i.e., ">" and "<") is not necessarily revealed to for the correctness of range queries. Accordingly, EncKV enhances the ORE scheme by using randomized orders in ciphertext comparisons, and only allows the server to know whether the query condition is matched.

Finally, EncKV provides an engineered approach to mitigate the inference attacks. Our empirical observation is that a certain query is commonly conducted in two phases. The first is to process the corresponding encrypted indexes to find matched primary keys (i.e., record IDs) on a given query condition, and then the second phase is to fetch the specified data values associated. And provisioning the server an ability to complete the two phases seamlessly will expose data correlations across different attributes. Therefore, EncKV splits the two and introduces an interactive batch query mechanism to reduce the leakage.

[2]In distributed data stores, "local index" implies that each node stores an index that only indexes its local records [15].

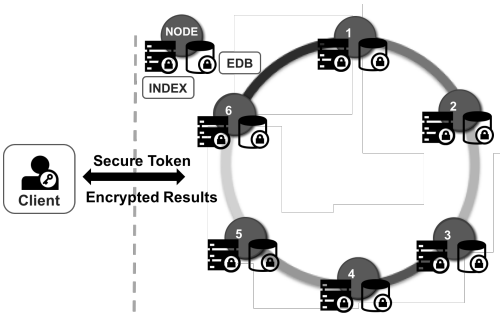


Figure 1: The architecture of EncKV.

EncKV has the following salient features:

- It stores data records with multiple attributes in the form of encrypted key-value pairs, and distributes them to the nodes via a standard data partitioning algorithm.
- It supports rich queries over distributed encrypted data records. The supported queries include keyword search, equality, count, join, range, like, sum, average, group by, max, and min.
- It offers guaranteed security, i.e., SSE's security notion for exact-match queries, and ORE [19]'s security notion for range-match queries.
- It preserves linear scalability of KV stores with respect to their performance. The query throughput increases linearly with the number of nodes in the cluster. Meanwhile, it enables parallel query processing. The query latency decreases when more nodes are deployed.

The paper is organized as follows. Section 2 presents our system architecture and threat assumptions. Section 3 presents our system design. Our security analysis is conducted in Section 4, and an extensive array of evaluation results is shown in Section 5. Finally, we introduce related work in Section 6, and conclude the paper in Section 7.

2. OVERVIEW

2.1 System Architecture

The system architecture of EncKV is shown in Figure 1, containing two entities, the *client* and *server node*. EncKV serves the clients who wish to store their sensitive data records in KV stores. A cluster of server nodes can be leased from the public cloud or be deployed at on-premise data centers. These nodes store encrypted data records and provide secure query functions to the clients. Correspondingly, EncKV implements two separate modules for the client and server nodes. The client module performs data encryption and decryption, encrypted index construction, as well as query token generation. It maintains the client's master key which is used to derive different private keys for the functions above. A *node* in the server module handles query requests from the client. It processes query tokens over the encrypted indexes, and utilizes the APIs of the underlying KV stores to put/get encrypted data records.

To insert a data record to EncKV, the client uses its private key to generate encrypted label-value (LV) pair(s)[3]. If the data is formated in the rich data model other than the

[3]We use the term "label" instead of "key" to avoid ambiguity.

simple key-value model, it will randomly be mapped into a set of encrypted LV pairs. This treatment allows EncKV to use the standard data partitioning algorithm (i.e., consistent hashing [9]) to distribute encrypted data records evenly across the nodes.

For the purpose of building local indexes, EncKV asks the client to maintain a small-sized consistent hashing ring which indicates the label range associated with each node. As a result, the client can directly insert LV pairs to targeted nodes and build the encrypted indexes for them on each node. To submit a secure query via secondary attributes of data, the client first generates a token set from the query condition attribute, and then broadcasts the tokens to each node respectively. After that, each node processes the tokens on its local index, and returns the matched, encrypted record IDs. Finally, the client decrypts the record IDs and generates labels to fetch the encrypted result values.

Remarks: (1) In our current implementation, EncKV does not add dummy records, while they can be inserted to mitigate inference attacks and leakage-abuse attacks (see our discussion in Section 4.3). (2) EncKV's query protocols require two rounds of interaction. The first is to obtain the encrypted record IDs, and the second is to fetch the matched results. This treatment facilitates an immediate security improvement to hide the associations between data values on different attributes (see more details in Section 3.5). (3) EncKV's index framework needs the client to generate query tokens for all the nodes, and each node produces partial results. Nevertheless, we show that this broadcast will not introduce too much overhead (see our evaluation in Section 5.2). One salient advantage of local indexes is that all nodes can process query tokens in parallel.

2.2 Threat Assumption

Following most of the prior studies on search over encrypted data [8, 19, 25, 27], EncKV considers the case that the client is secure and trusted. It will not expose the keys to server nodes, and the keys are securely stored at the client. EncKV assumes that the attackers will never have access to a client's private keys, but they can dump all the encrypted indexes and KV pairs from server nodes. They can also monitor the query protocols and learn about the query tokens, accessed index entries, and encrypted results. EncKV does not consider the case where attackers can access the background information about the queries and datasets, e.g., the partial (entire) distribution or the content of queries or records [3, 20]. Nevertheless, discussions on how to mitigate those threats are conducted in Section 4.3. Finally, EncKV does not handle the case where malicious attackers modify or delete the indexes and records intentionally, which has been addressed by orthogonal studies like [2].

2.3 Cryptographic Primitives

A symmetric encryption scheme $(KGen, Enc, Dec)$ contains three algorithms: The key generation algorithm $KGen$ takes a security parameter λ to return a secret key k. The encryption algorithm Enc takes a key k and a value $v \in \{0,1\}^n$ to return a ciphertext $v^* \in \{0,1\}^n$; The decryption algorithm Dec takes k and v^* to return v.

Define a family of pseudo-random functions $F : \{0,1\}^\lambda \times \{0,1\}^m \to \{0,1\}^n$, if for all probabilistic polynomial-time distinguishers D, $|Pr[D^{F(k,\cdot)} = 1 | k \xleftarrow{\$} \{0,1\}^\lambda] - Pr[D^g = 1 | g \xleftarrow{\$} \{\mathsf{Func}[m,n]\}]| < negl(\lambda)$, where $negl(\lambda)$ is a negligible function in λ.

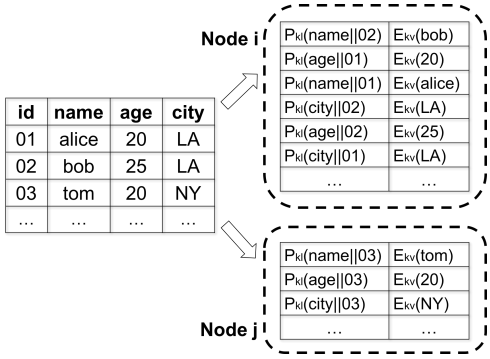


Figure 2: Construction summarized from [33]

3. THE ENCKV DESIGN

This section presents the designs of EncKV's encrypted exact-match and range-match indexes in detail, based on which several types of queries are then instantiated. Features such as batch queries and incremental updates are also presented for security and practical considerations.

3.1 The Underlying Encrypted KV Store

EncKV builds on top of an encrypted KV store [33]. This prior design has two features. First, it proposes a secure data partition algorithm that dispatches encrypted data records across distributed nodes, while preserving horizontal scalability. Second, it sketches an encrypted local index framework towards efficient queries via secondary attributes of data in distributed data stores. EncKV's index designs are carefully integrated into this framework for the practical performance of secure rich queries. Before introducing EncKV in greater detail, we shall first summarize the underlying encrypted KV store proposed in [33].

As an example, Figure 2 illustrates the column-oriented data model in [33], and other data models are supported as well. The core idea of its secure data partition algorithm is to map data records into encrypted label-value pairs. Specifically, each LV pair in EncKV is constructed as $\langle P(k_l, C||R), Enc(k_v, v)\rangle$, where k_l, k_v are private keys, P is a secure PRF, R is the record ID, C is a column (secondary) attribute, v is a value on C, and Enc is a symmetric key encryption algorithm.

Different from [33] which uses $P(k_l, C||R)$ as the label for partition, EncKV uses the unique record ID R instead to preserve the locality for queries via multiple attributes. As a result, all the encrypted values for a given record are stored at the same node, but they are still fully scrambled to protect the auxiliary information such as the number of columns and the associations between the underlying values.

Regarding the encrypted local index framework, the client is asked to maintain a consistent hashing ring so that it can trace the locations of values and build encrypted indexes that index the values stored on the same node. The benefits are two-fold: 1) Inter-node interaction is avoided during the query process, because if we directly adopt generic primitives, additional dedicated nodes are needed to store the encrypted global indexes. 2) Nodes can process the queries in parallel. The query workload is also balanced.

Algorithm 1 $Build_{ext}$: build exact-match indexes

Input: Private key k_e; secure PRFs $\{G1, G2, H1, H2\}$; values $\{v_1, \cdots, v_m\}$ on attribute C.
Output: Encrypted indexes $\{I_1^{ext}, \cdots, I_n^{ext}\}$.

Initialize a hash table S to maintain counters;
for $v_j \in \{v_1, \cdots, v_m\}$ **do**
  $i \leftarrow route(R)$; // R is v_j's ID, $i \in \{1, n\}$ is node ID
  $t_1 \leftarrow G1(k_e, C||v_j||i)$;
  $t_2 \leftarrow G2(k_e, C||v_j||i)$;
  if $S.find(i||j) = \perp$ **then**
    $c_i^j \leftarrow 0$;
  else
    $c_i^j \leftarrow S.find(i||j)$;
  end if
  $\alpha \leftarrow H1(t_1, c_i^j)$;
  $\beta \leftarrow H2(t_2, c_i^j) \oplus Enc(k_R, R)$;
  c_i^j++;
  $S.put(i||j, c_i^j)$;
  $I_i^{ext}.put(\alpha, \beta)$;
end for

3.2 Exact-match Index and Query Protocol

In this subsection, we will articulate the designs of EncKV's encrypted exact-match indexes and query protocols.

3.2.1 Encrypted Index Design

As mentioned, the construction of EncKV's encrypted indexes for secure exact-match queries is inspired by a recently proposed (and elegant) SSE scheme [4]. This design uses KV pairs to index files that match the same keyword, and each file of this keyword is distinguished by a stateful counter. EncKV adopts this idea and indexes the record IDs that match the same values on a certain column attribute. To integrate the design into the distributed local index framework, EncKV's client tracks the values and maintains the counters for each distinct value on different nodes during the index building procedure.

The detailed algorithm to index values $\{v_1, \cdots, v_m\}$ for a given column attribute C is shown in Algorithm 1. This procedure is executed at the client. For each v_j for j from 1 to m, n counters are first initialized, where n is the number of nodes. Then the client finds the target node i for v_j based on the position of its record ID R on the consistent hashing ring. After that, it generates two tokens by embedding the value securely via secure PRF, i.e., $t_1 = G1(k_e, C||v_j||i)$ and $t_2 = G2(k_e, C||v_j||i)$, and further uses the corresponding counter c_i^j to generate the encrypted index entry, i.e., $\langle \alpha = H1(t_1, c_i^j), \beta = H2(t_2, c_i^j) \oplus Enc(k_R, R) \rangle$.

The encrypted index above holds the security notion of SSE. The index size is known to the nodes. Without querying, no other information about the underlying content is learned. Note that the counters will not be used to generate query tokens at the client, and thus can be dropped after the indexes are uploaded. If new records are incrementally added, those counters can be cached at the nodes in their encrypted form to generate new index entries. More details can be found in Section 3.6.

3.2.2 Secure Query Protocol

The corresponding query protocol following the index construction is executed between the client and nodes as presented in Algorithm 2. Given a query via two attributes, the client wants to find all the values $\{v_r\}$ in attribute C_r on the matching condition such that another attribute C_v's value should exactly be the value v. First, the client generates query tokens for each node $\{t_1, t_2\}$, where $t_1 = G1(k_e, C_v||v||i)$ and $t_2 = G2(k_e, C_v||v||i)$. Each node processes these tokens in parallel. In particular, each node increments a counter c_i to locate all the matched index entries via $H1(t_1, c_i)$ till no entry is returned, and each entry is unmasked via XORing $H2(t_2, c_i)$ to get r the encrypted record ID. After that, all matched $\{r\}$ are sent back to the client for decryption. For each decrypted ID R, the client generates the corresponding label via $P(k_l, C_r||R)$ to fetch the encrypted result value.

During the query procedure, data values and attributes are strongly protected. Each node only learns the query tokens, accessed index entries, and encrypted result values. Due to the deterministic property of tokens, it also learns the repeated queries on the same attribute. Note that the proposed query protocol requires two rounds of interaction between the client and each node. Such treatment prevents the server from generating labels of other values which are never queried. Each node only learns the matched values associated to the same column attribute, and it will not learn the associations between values in different attributes of a record, thereby effectively addressing inference attacks. Formal security analysis will later be conducted in Section 4.1.

Algorithm 2 $Query_{ext}$: secure exact-match query protocol

Input: Private key k_e; query condition value v; query condition attribute C_v; result value attribute C_r.
Output: Encrypted matched results $\{v_r\}$.

Client.Token

for $i \in \{1, \cdots, n\}$ **do**
  $t_1 \leftarrow G1(k_e, C_v||v||i)$;
  $t_2 \leftarrow G2(k_e, C_v||v||i)$;
  Send (t_1, t_2) to node i;
end for

$Node_i$.ExtQuery

$c_i \leftarrow 0$;
$\alpha \leftarrow H1(t_1, c_i)$;
while $find(\alpha) \neq \perp$ **do**
  $\beta \leftarrow find(\alpha)$;
  $r \leftarrow I_i^{ext}.get(H2(t_2, c_i) \oplus \beta)$;
  c_i++;
  $\alpha \leftarrow H1(t_1, c_i)$;
  Return r to client for decryption;
  Client
  $R \leftarrow Dec(k_R, r)$;
  $l \leftarrow P(k_l, C_r||R)$;
  Fetch v_r via l;
end while
// Implementation note: all matched $\{r\}$ are sent back in a batch, and $\{v_r\}$ are also fetched in a batch.

3.3 Range-match Index and Query Protocol

In this subsection, we will present EncKV's encrypted range-match indexes and the corresponding query protocols.

Algorithm 3 $Build_{rng}$: build range-match indexes

Input: Private keys k_r, k_o; secure PRFs $\{G1, G2, H1, H3\}$; values $\{v_1, \cdots, v_m\}$ on attribute C.
Output: Encrypted indexes $\{I_1^{rng}, \cdots, I_n^{rng}\}$.

Initialize a hash table S to maintain counters;
for $v_j \in \{v_1, \cdots, v_m\}$ **do**
  $i \leftarrow route(R)$; // R is v's ID, $i \in \{1, n\}$ is node ID
  $t_1 \leftarrow G1(k_r, C||i)$;
  $t_2 \leftarrow G2(k_r, C||i)$;
  if $S.find(i) = \perp$ **then**
    $c_i \leftarrow 0$;
  else
    $c_i \leftarrow S.find(i)$;
  end if
  $\alpha \leftarrow H1(t_1, c_i)$;
  $ct_R \leftarrow OREenc(k_o, v_j, c_i)$; // shown in Algorithm 4
  $\beta \leftarrow H3(t_2, c_i) \oplus (ct_R||Enc(k_R, R))$;
  $c_i + +$;
  $S.put(i, c_i)$;
  $I_i^{rng}.put(\alpha, \beta)$;
end for

3.3.1 Design Rationale

In the current literature, a challenge of enabling secure range queries is how to design secure comparison schemes with minimized leakage. Most of the existing ORE primitives leak more information (i.e., order information) than SSE. Recent attacks show that order information can be exploited to recover the underlying values of ciphertexts [10,12, 20]. To address this issue, the desired ORE primitive must achieve a strong security notion; that is, the ciphertexts should be semantically-secure encryptions of their underlying values [19]. Then if the attackers obtain the encrypted database, they can never learn any useful information.

However, only achieving the requirement above is not necessarily secure, because attackers still know order information during the queries. For example, attackers can learn which values are smaller than or greater than the query value. Such information can be combined with partial knowledge of the query distribution and record distribution to compromise the database [16].

To this end, we start from one of the latest ORE schemes [19] that defend against inference attacks, and it is also the most efficient ORE scheme currently. As this scheme does not fully address the second issue, we further enhance it to protect the order information. Here, we consider the order as ">" or "<". Our observation is that secure range queries can still be supported without leaving the order in cleartext. The core idea of our design is to hide the order in both query token generation and ciphertext comparison. As a result, the only known information is that the index entries match an encrypted order condition. The attackers will learn neither the order of the underlying values on a column attribute, nor whether two different queries are conducted in the same order condition.

3.3.2 Encrypted Index Design

First of all, the construction of encrypted range-match indexes follows the same treatment as the encrypted exact-matched indexes. For security, each index entry should be strongly encrypted, and the information on which entries associated with the same column attribute should also be hidden before querying. This objective can be achieved through searchable encryption techniques. For index and data locality, EncKV's client is still required to track the locations of data values. Algorithm 3 presents the index building procedure. For each value v_j on a column attribute C, the client first locates the node where the record is stored, and then generates the encrypted index entry $\langle \alpha, \beta \rangle$ by securely embedding C and the counter c_i. Note that the underlying content of β also contains the ORE ciphertext ct_R which is computed from our enhanced ORE scheme introduced in the next paragraph. As a result, the encrypted range-match index is integrated into EncKV's local index framework.

Algorithm 4 $OREenc$: enhanced ORE encryption

Input: Private key k_o; secure PRFs $\{F1, F2, F3\}$; secure PRP π; value v; counter c;
Output: ORE ciphertext ct_R

Derive k_1, k_2, k_3 from k_o;
Generate a nonce γ;
for $i \in \{1, B\}$ **do**
  for $j \in \{1, 2^b\}$ **do**
    $j^* \leftarrow \pi^{-1}(F2(k_2, v_{|i-1}), j)$;
    if $CMP(j^*, v_{|i}) \neq 0$ **then**
      $s_{i,j} \leftarrow F3(k_3, CMP(j^*, v_{|i}))||C||j)$;
      $z_{i,j} \leftarrow Q1(s_{i,j}, c) + Q2(F1(k_1, v_{|i-1}||j), \gamma)$;
    else
      $z_{i,j} \leftarrow$ "$equal$" $+ Q2(F1(k_1, v_{|i-1}||j), \gamma)$;
    end if
  end for
  $ct_{R|i} \leftarrow z_{i,1}, \cdots, z_{i,2^b}$;
end for
$ct_R \leftarrow \gamma, ct_{R|1}, \cdots, ct_{R|B}$;

Algorithm 5 $OREcmp$: ORE compare operation

Input: ORE query token ct_L; ORE ciphertext ct_R;
Output: *true* or *false*.

$\gamma, u'_1, ..u'_B \leftarrow ct_R$;
$u_1, .., u_B \leftarrow ct_L$;
for $i \in \{1, \cdots B\}$ **do**
  $x_i, \tilde{v_{|i}}, q_i \leftarrow u_i$;
  $z_{i,1}, ..., z_{i,2^b} \leftarrow u'_i$;
  $s_i \leftarrow z_{i,\tilde{v_{|i}}} - Q2(x_i, \gamma)$;
  if $s_i \neq 0$ and $s_i = Q1(q_i, c)$ **then**
    return *true*; // condition matched
  end if
end for
return *false*;

Enhanced ORE scheme: The idea of the ORE scheme proposed in [19] is to split a message into bit blocks with equal length, and conduct comparison from the significant least blocks of two messages. For example, if the message space is 4 bits, the block size is 2 bits, each message will then be encrypted into 2 blocks. Specifically, each block has total 2^2 possible values $\{00, 01, 10, 11\}$. The message block, say "10" to be encrypted, will be transformed to 4 sub blocks, where the order information $\{>, >, =, <\}$ to

Algorithm 6 $Query_{rng}$:secure range-match query protocol

Input: Private key k_r, k_o; query condition value v; order condition $cmp \in \{>, <\}$; query condition attribute C_v; result value attribute C_r.

Output: encrypted matched results $\{v_r\}$.

Client.Token

1: **for** $i \in \{1, \cdots, n\}$ **do**
2: $t_1 \leftarrow G1(k_r, C_v||i)$;
3: $t_2 \leftarrow G2(k_r, C_v||i)$;
4: **for** $i \in \{1, B\}$ **do**
5: $\tilde{v_{|i}} \leftarrow \pi(F2(k_2, v_{|i-1}), v_{|i})$;
6: $q_i \leftarrow F3(k_3, cmp||C||\tilde{v_{|i}})$;
7: $u_i \leftarrow F1(k_1, v_{|i-1}||\tilde{v_{|i}}), \tilde{v_{|i}}, q_i$;
8: **end for**
9: $ct_L \leftarrow (u_1, \cdots, u_B)$;
10: Send (t_1, t_2, ct_L) to node i;
11: **end for**

$Node_i$.RngQuery

1: $c_i \leftarrow 0$;
2: $\alpha \leftarrow H1(t_1, c_i)$;
3: **while** $find(\alpha) \neq \perp$ **do**
4: $\beta \leftarrow find(\alpha)$;
5: $r \leftarrow I_i^{rng}.get(H3(t_2, c_i) \oplus \beta)$;
6: Parse r as $r_x \leftarrow Enc(k_R, R)$, $r_y \leftarrow ct_R$;
7: c_i++;
8: // ORE compare operation shown in Algorithm 5
9: **if** $OREcmp(ct_L, ct_R) = true$ **then**
10: Return r_x to the client;
11: **end if**
12: $\alpha \leftarrow H1(t_1, c_i)$;
13: **end while**
14: // Note: we ignore the steps to fetch final results, which is the same in Line 7 to 10 in Algorithm 2.

each value above is securely embedded with its prefix block[4]. Here, the order cmp is defined as the output of the comparison $CMP(m1, m2)$ for block $m1$ and $m2$. For more details, we refer the readers to [19].

We note that the original scheme reveals the order between the query value and each ciphertext on the column. Such leakage tells partial order information between ciphertexts, i.e., some ciphertexts are smaller than or greater than others. Even the order is transformed as a pseudo-random tag, such tags should be sent along the queries, which is exploitable if the attackers know the query distribution.

To minimize the leakage, we propose to protect the order by embedding it securely via PRF with the column attribute, the block index, and the stateful counter, as shown in Line 7 and Line 8 of Algorithm 4, i.e., our enhanced ORE encryption algorithm. The sub block j in block i is encrypted as $Q1(s_{i,j}, c) + Q2(F1(k_1, v_{|i-1}||j), \gamma)$, where $s_{i,j} = F3(k_3, CMP(j^*, v_{|i}))||C||j)$, $v_{|i}$ is the block value, $v_{|i-1}$ is the prefix block value, C is the column attribute, and c is the counter of this value on the column. j^* is the securely permuted j in one of the possible values to this block, where $j \in [1, 2^b]$, b is the bit length of each block, and B is the number of blocks.

[4]The prefix block will be set as "null", if the encrypted block is the first block [19].

ct_L (00 10)		
PRP	$\pi(F2_{k2}(\perp),00) \rightarrow 1$	$\pi(F2_{k2}(00),10) \rightarrow 2$
ct_L	$F3_{k3}$["<"||age||1], $F1_{k1}(\perp\|\|1),1$	$F3_{k3}$["<"||age||2], $F1_{k1}(00\|\|2),2$

ct_R (00 01)		
1	**00= +Q2($F1_{k1}(\perp\|\|1)$,r1)**	Q1[$F3_{k3}$(11>||age||1),c1]+Q2($F1_{k1}$(00||1),r1)
2	Q1[$F3_{k3}$(10>||age||2),c1]+Q2($F1_{k1}(\perp\|\|2)$,r1)	**Q1[$F3_{k3}$(10>||age||2),c1]+Q2($F1_{k1}$(00||2),r1)**
3	Q1[$F3_{k3}$(11>||age||3),c1]+Q2($F1_{k1}(\perp\|\|3)$,r1)	01= +Q2($F1_{k1}$(00||3),r1)
4	Q1[$F3_{k3}$(01>||age||4),c1]+Q2($F1_{k1}(\perp\|\|4)$,r1)	Q1[$F3_{k3}$(00<||age||4),c1]+Q2($F1_{k1}$(00||4),r1)

ct_R (00 11)		
1	**00= +Q2($F1_{k1}(\perp\|\|1)$,r2)**	11= +Q2($F1_{k1}$(00||1),r2)
2	Q1[$F3_{k3}$(10>||age||2),c2]+Q2($F1_{k1}(\perp\|\|2)$,r2)	**Q1[$F3_{k3}$(10<||age||2),c2]+Q2($F1_{k1}$(00||2),r2)**
3	Q1[$F3_{k3}$(11>||age||3),c2]+Q2($F1_{k1}(\perp\|\|3)$,r2)	Q1[$F3_{k3}$(01<||age||3),c2]+Q2($F1_{k1}$(00||3),r2)
4	Q1[$F3_{k3}$(01>||age||4),c2]+Q2($F1_{k1}(\perp\|\|4)$,r2)	Q1[$F3_{k3}$(00<||age||4),c2]+Q2($F1_{k1}$(00||4),r2)

Figure 3: Our proposed ORE compare operation

This improved construction guarantees that the order in each sub block is different, and the order conditions for different values and attributes are also different. Due to the deterministic property of PRF, the query comparison can still correctly be performed via token matching, which will later be introduced in the query protocol.

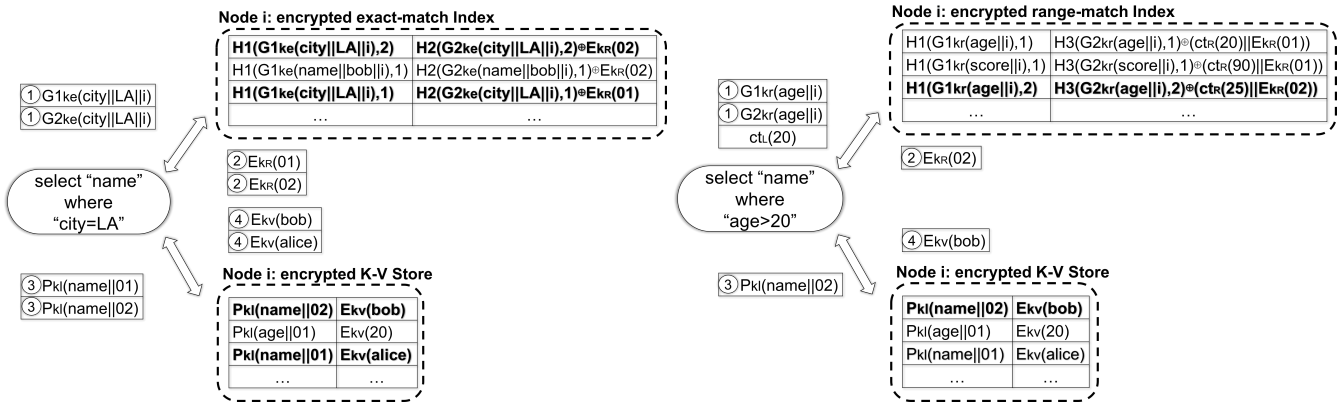
3.3.3 Secure Query Protocol

Based on the index construction, we present the range-match query protocol in details in Algorithm 6. Given a query via two attributes, the client wants to find all values $\{v_r\}$ in attribute C_r on the matching condition such that another attribute C_v's value should smaller than the value v. Similar to the exact-match query, the client generates query tokens for each node $\{t_1, t_2\}$ from C_v. For ORE comparison, the client needs to compute another token ct_L which contains the encrypted blocks $\{u_1, \cdots, u_B\}$ with distinct encrypted order condition q_i of each block. Each node processes $\{t_1, t_2\}$ in parallel, i.e., unmasking the corresponding ORE index entries via incremental counters. After that, each node calls the ORE compare operation $OREcmp$ to compare ct_L and ct_R in each entry above, as presented in Algorithm 5. The process is conducted from the most significant block. Symmetric to the block encryption, the encrypted order is obtained via $s_i = z_{i,\tilde{v_{|i}}} - Q2(x_i, \gamma)$, where $z_{i,\tilde{v_{|i}}}$ is the block of ct_R, x_i is the corresponding block of ct_L, and γ is the nonce of this ciphertext. With the encrypted query condition q_i, $Q1(q_i, c)$ is computed via counter c to check whether it is matched to s_i. If matched, the encrypted record ID will be sent to the client to fetch the final result values on attribute C_r.

Note that this design reveals the repeated queries, and the equality of query values and ciphertexts. It also indicates the position of the first block in which two values differ, which is the same to the adopted ORE scheme [19]. More detailed security analysis will be given in Section 4.1. The query time complexity in the current treatment is $O(m_C)$, where m_C is the number of values on attribute C at a certain node. The performance can further be improved via binary search, i.e., sorting values before encryption as indicated in [19].

3.4 Secure Rich Query Instantiation

EncKV's encrypted indexes readily enable rich queries supported in existing NoSQL data stores [9,11]. These stores implement SQL-like query language for easy data manage-



(a) Exact-match query (b) Range-match query

Figure 4: Secure query protocol illustration

ment. Accordingly, this section will use SQL-like query examples to introduce how EncKV support those queries.

Keyword Search, Equality and Count queries: Given a keyword search or equality query "SELECT *name* WHERE $city = LA$ "in Figure 4(a), the client first generates a pair of tokens $\{t_1, t_2\}$ for each node based on the keyword condition "$city = LA$", where $t_1 = G1(k_e, city||LA||i)$, $t_2 = G2(k_e, city||LA||i)$, and i is the node ID. Then, each node processes these tokens in parallel. Specifically, all the matched index entries are located via $H1(t_1, c_i)$ in node i, where c_i is a counter. The encrypted record ID ($Enc(k_R, 01)$ and $Enc(k_R, 02)$) after XORing $H2(t_2, c_i)$ are returned. Finally, the client decrypts them and obtains the encrypted result values via labels $P(k_l, name||01)$, $P(k_l, name||02)$. Regarding count queries, the client needs to count the values returned from each node, and aggregate the counts .

Range and Like queries: In terms of range-match queries such as "SELECT *name* WHERE $age >$20" in Figure 4(b). The client firstly generates tokens $t_1 = G1(k_r, age||i)$, $t_2 = G2(k_r, age||i)$ with ORE ciphertext of $ct_L(20)$ containing randomized " $>''$. Each node scans index entries via $H1(t_1, c_i)$ with incremented c_i, and gets ct_R by XORing $H3(t_2, c_i)$ with entries. Then the node i returns the encrypted recored ID ($Enc(k_R, 02)$) to the client as $ORE_{cmp}(ct_L(20), ct_R)$ algorithm outputs *true*. Finally, the client fetches the result value ($E(k_v, bob)$) via label $P(k_l, name||02)$.

EncKV also supports LIKE (aka prefix) query. For instance, the query "SELECT $City$ WHERE $City$ LIKE $'A\%'$" obtains answers like "Argentina" or "Australia" and so on. Our adopted ORE scheme [19] supports comparison on both numeric numbers and alphanumeric strings. Recall that each ORE ciphertext is encrypted by blocks, and previous block content is embedded in the current block ciphertext for prefix matching. During the comparison, the first different blocks between ct_L and ct_R tell that their previous blocks are the same.

Join Queries: EncKV supports Join queries such that any attributes of two tables can be joined together. Here, we define a generic join query statement, FROM T_1 JOIN T_2 ON $T_1.C_1 = T_2.C_2$ WHERE $field$, where T_1 and T_2 are two tables being joined, C_1 and C_2 are attributes of T_1 and T_2, and $field$ is a join condition such as an exact-match or range-match operation. For example, given a query "SELECT $T_2.GPA$ from T_1 JOIN T_2 on $T_1.id$=$T_2.id$ where $T_1.age >$ 20", the client parses the query and performs "SELECT $T_1.id$ from T_1 WHERE $T_1.age > 20$" via the range-match indexes, which derive the matched record ID set R. Then, the client generates label $P(k_l, T_2.GPA||R_i)_{R_i \in R}$ to get final results.

Sum and Average queries: Following the treatment in prior encrypted databases, nodes in EncKV can perform aggregation on encrypted data values by using addictive homomorphic encryption (HOM) scheme [27]. Values to be aggregated are encrypted via a certain HOM encryption scheme, i.e., $\langle P(k_l, C||R||HOM), HEnc(k_v, v)\rangle$ When the client issues a query in the form "SELECT $SUM(score)$ FROM $Score$ WHERE $age > 20$", it firstly queries the matched record ID set R via the range-match indexes from "SELECT stu_ID FROM $Score$ WHERE $age > 20$". Then each node locates and aggregates the HOM ciphertext via label $\{P(k_l, score||R_i||HOM)\}_{R_i \in R}$. This procedure can be done in parallel under the local index framework. Finally, each node returns intermediate results to the client for aggregation. And the average value can further be computed at the client.

Group By queries: EncKV performs Group By queries via combining exact-match queries with aggregation computation. Suppose the client issues a group by request as "SELECT city, sum(age) GROUP BY city". It firstly finds the specific record ID set R for each group such as "LA" via the exact-match query "SELECT stu_ID where city=LA". Here, we assume that the client knows all the distinct city names. Then it can use the HOM label $\{P(k_l, age||R_j||HOM)\}_{R_j \in R}$ to ask the corresponding nodes to compute the aggregation of age values which associate "LA". Aggregation results are also finalized at the client just like Sum queries. For other group entries, it follows the same treatment.

Max and Min queries: To support Max and Min queries, the client inserts a specific LV pairs for the MAX/MIN data values on a column attribute. For example, if the maximum value of the attribute "age" is "100", the client generates LV pair: $\langle P(k_l, age||MAX), E(k_v, 100)\rangle$. When the client wants to query the maximum data, it computes the label $P(k_l, age||MAX)$ to get the maximum value. Obtaining the minimum value can be realized in the same way.

3.5 Batch Queries

EncKV's query protocols across different attributes are conducted in two phases. The first is to find the encrypted record IDs that matches the query condition on a specific query attribute, and the second phase is to fetch the values of these matched records on a targeted attribute. As mentioned in Section 3.2.2, EncKV implements those two phases in two rounds respectively to hide the associations of encrypted KV pairs in same records. However, we observe that such treatment still might allow the nodes to know the associations between index entries and result values (see analysis in Section 4).

To reduce the above leakage, an engineered approach is to conduct queries in a batched manner. For a batch of queries, the client first parses the query conditions in a way that the overlapped or repeated query conditions will be queries only once in this batch. After receiving the encrypted result IDs, the client decrypts them and generates the labels from distinct IDs and the targeted attributes. In addition, the client permutes those labels and fetches final result values from the corresponding nodes. We note that such improvement can be realized via a dedicated query planner which is also used for improving performance in [25, 31]. Based on the above approach, the associations between values and index entries on same records can be well protected.

3.6 Secure Update Operations

EncKV provides two ways of update operations when new data records are added, i.e., bulk update and incremental update. Bulk update is suitable for the case of adding a large number of records, i.e., migrating an unencrypted database to EncKV. Encrypted exact-match and range-match indexes can be built via their index building functions introduced in Algorithm 1 and Algorithm 3 respectively. Yet, we note that building new indexes require the client to generate additional tokens for new indexes. Thus, periodical index consilidations are demanded to preserve the complexity of token generation.

Incremental update is suitable for the case when data records are occasionally inserted into EncKV. As a result, new index entries need to be added to existing indexes. To implement incremental update, the state information (i.e., counters) on each indexed attribute should carefully be maintained either at the client or at the nodes in the encrypted form so that the client can generate the corresponding index entries without affecting the following queries. We note that this operation will leak additional information just like most of the prior dynamic SSE schemes [4, 14]. Once the attributes are queried, the nodes will know whether the newly inserted index entires are associated with those attributes. One recent scheme addresses this issue via token permutation [2], and we will integrate this technique to EncKV in future.

Currently, we consider the single-client setting. The multi-client setting will be addressed as future work. In addition to access control, concurrent query operations will also be investigated when encrypted index entries and data records are accessed from different clients simultaneously.

4. SECURITY ANALYSIS

This section will analyze EncKV's security strength. The values of different attributes in each record are stored as encrypted LV pairs. The encrypted underlying data storage defends against offline inference attacks [20], because auxiliary information such as schema information is completely hidden even if the attacker obtains the entire encrypted database. Our security analysis focuses on our proposed secure exact-match queries and secure range-match queries. We will follow the primitives we adopted, SSE [4] and ORE [19], to quantify the security guarantees of EncKV's query protocols respectively. In addition, we will discuss EncKV on the protection against a series of recent attacks on search over encrypted data.

4.1 Security on Exact-match Queries

Since EncKV's secure exact-match queries are realized in the framework of SSE [8], the nodes only learn the controlled leakage, but never learn the underlying contents of queries and results. Basically, the index size will be learned once the index is uploaded to the server. Search and access pattern will be learned along the queries, where search pattern indicates the repeated queries, and access pattern indicates the accessed ciphertexts. In our targeted queries which contain multiple query attributes, and thus access pattern also includes the associations between values of those attributes. Following the notion of SSE, we first define the leakage functions in EncKV as follows:

$$\mathcal{L}_1^{ext}(\mathbf{C}) = (\{m_i\}_n, \langle|\alpha|, |\beta|\rangle)$$

where $\mathbf{C}$ is the set of secondary attributes, m_i is the size of local index I_i^{ext} of node i, n is the number of nodes, and $|\alpha|, |\beta|$ are the lengths of label and value in the index entry.

$$\mathcal{L}_2^{ext}(v_C, C_v, C_r) = (\{t_1^i, t_2^i\}_n, \{\{\langle\alpha, \beta\rangle, \langle l, v^*\rangle\}_{c_i}\}_n)$$

where v_C is the query value, C_v is the attribute of v_C, C_r is the attribute of result values, and $\{t_1^i, t_2^i\}_n$ are tokens for n nodes respectively. Given a query, the matched index entries and results $\{\langle\alpha, \beta\rangle, \langle l, v^*\rangle\}_{c_i}$ at each node are known.

$$\mathcal{L}_3^{ext}(\mathbf{Q}) = (M_{q\times q}, T_{v^*\to\alpha})$$

where $\mathbf{Q}$ is q number of adaptive queries, and $M_{q\times q}$ is a symmetric bit matrix to trace the same queries. $M_{i,j}$ and $M_{j,i}$ are equal to 1 if $t_1^i = t_1^j$ for $i, j \in [1, q]$. Otherwise, they are equal to 0. $T_{v^*\to\alpha}$ is an inverted list that traces index entries that match each result value, which is also known as inference information [20]. For each posting list $v^*|\{\alpha_1, \cdots, \alpha_a\}$ in T, the associations between the queried index entries of different attributes and each queried result value are learned.

In terms of the quantified leakage, we present the security definition of exact-match queries in Definition 1 in Appendix, and give the following theorem.

Theorem 1. *Ext is adaptively secure with ($\mathcal{L}_1^{ext}$, $\mathcal{L}_2^{ext}$, $\mathcal{L}_3^{ext}$) under the random-oracle model if $G1, G2, H1, H2, P$ are secure PRF.*

We leave the detailed proof in Appendix.

The security notion of EncKV's exact-match queries is stronger than deterministic encryption (DET), which is used in several existing encrypted databases [27]. DET-based designs expose the server all the same values on an attribute, while EncKV will not tell that information. For other auxiliary information, associations between values across attributes (aka inter-column and intra-column associations) are directly exposed in existing encrypted databases with

legacy compatibility, while such information in EncKV will greatly be reduced. On the one hand, the attribute is secretly embedded in the encrypted index, the server will never learn whether the tokens of different query values on the same attribute or not. On the other hand, EncKV's batch query mechanism further hides the associations of columns.

4.2 Security on Range-match Queries

EncKV's secure range-match queries are designed on top of the ORE scheme proposed in [19]. Therefore, the security achieves the same level as the scheme in [19]. That is briefly, the ciphertexts are semantically secure, and the first different block that differs between two values in the comparison. To achieve general protection and integrate the indexes into the local index framework, EncKV leverages SSE techniques as an overlay to mask ORE ciphertexts. Similar to exact-match queries, inference information will also be learned since queries may involve multiple query attributes. Accordingly, we define the leakage functions as follows:

$$\mathcal{L}_1^{rng}(\mathbf{C}) = (\{m_i\}_n, \langle|\alpha|, |\beta|\rangle)$$

where $\mathbf{C}$ is the set of secondary attributes, m_i is the size of local index I_i^{rng} of node i, n is the number of nodes, and $|\alpha|, |\beta|$ are the lengths of label and value in the index entry.

$$\mathcal{L}_2^{rng}(v_C, C_v, C_r) = (\{t_1^i, t_2^i\}_n, ct_L, \{\{\langle\alpha, \beta\rangle, \langle l, v\rangle\}_{c_i}\}_n)$$

where v_C is the query value, C_v is the query attribute, C_r is the attribute of result value , ct_L is the token for ORE comparison, and $\{t_1^i, t_2^i\}_n$ are tokens for n nodes respectively. Given a query, the matched index entries and result pairs $\{\langle\alpha, \beta\rangle, \langle l, v*\rangle\}_{c_i}$ at each server node are known. In addition, the rest of index entries on this column will also be learned.

$$L_3^{rng}(v_C, cmp) = (\{\{b_{dif}\}_{c_i}\}_n)$$

where b_{dif} is the first block that differs in the comparison of matched ORE ciphertexts.

$$L_4^{rng}(\mathbf{Q}) = (M_{q\times q}, T_{v*\to\alpha})$$

where $\mathbf{Q}$ is q number of queries, and $M_{q\times q}$ is a symmetric bit matrix to trace the same queries. $M_{i,j}$ and $M_{j,i}$ are equal to 1 if $t_1^i = t_1^j$ for $i, j \in [1, q]$. Otherwise, they are equal to 0. $T_{v^*\to\alpha}$ is an inverted list that indicates the associations between the index entries of different attributes and the result values as defined in exact-match queries. Accordingly, we present the security definition of range-match queries in Definition 2 in Appendix and give the theorem below.

Theorem 2. *Rng is non-adaptively secure with* ($\mathcal{L}_1^{rng}$, $\mathcal{L}_2^{rng}$, $\mathcal{L}_3^{rng}$, $\mathcal{L}_4^{rng}$) *if* $G1, G2, H1, H3, P, F1, F2, F3$ *are secure PRF.*

We give the proof in Appendix.

Our enhanced ORE scheme protects the order information in queries and ciphertexts along the comparison. Recall that in line 6 of Algorithm 6, the query order is protected in the ORE query token, i.e., $q_i = F2(k_3, cmp||C||\tilde{v_{|i}})$, where *cmp* is the order, C is the query attribute, and $v_{|i}$ is the ith block of query value. As a result, different query values or attributes will result in different query tokens. Then q_i is used by the server node to compute $Q1(q_i, c)$ in line 7 of Algorithm 5. If the output is matched with s_i in the ciphertext, this entry will be considered to be matched.

4.3 Further Discussion

Remark on inference attacks: Recent attacks [16, 20] use auxiliary information to compromise the confidentiality of encrypted databases. In [20], reference databases, table structures, order information, and frequency statistics are exploited to attack databases in property-preserving encryption, i.e., order-preserving encryption (OPE) and deterministic encryption (DET).

We note that EncKV effectively defends against these attacks. Our exact-match and range-match indexes are built via SSE [8] and ORE [19], which are semantically secure. Exact-match queries will not reveal matched values without querying, and range-match queries via our enhanced ORE scheme will not even tell the order information. In addition, the associations between data values are also protected in the proposed batch query mechanism. Again, note that one may always add dummy records to improve security.

In [16], a generic attack is proposed, which only relies on prior knowledge on query distribution. The attackers can recover the database without knowing the orders of query results. This attack samples queries and observes the size of results for each to find the order of ciphertext in high probability. We are aware that none of existing encrypted databases design can address this attack. Nevertheless, we argue that attackers need to make more efforts in EncKV, i.e., compromising all EncKV's nodes to obtain the full size of results in each query. Otherwise, this attack cannot be launched in the first place.

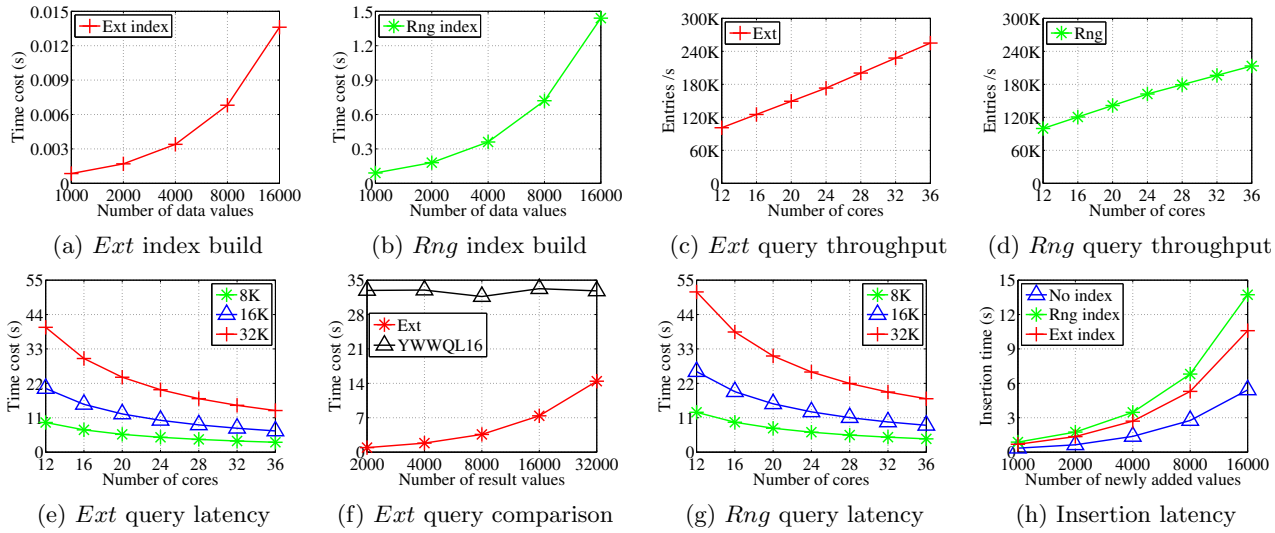
Remark on leakage-abuse attacks: In [3], several attacks on SSE are proposed by abusing search and access patterns. Despite their effectiveness, these attacks heavily rely on the amount of prior information about the databases and queries. Besides, as shown in their countermeasure evaluation, random padding largely reduces the reconstruction ratio of queries and data. A very recent attack is proposed to compromise SSE schemes that support legacy applications [28]. The targeted schemes leak more information than schemes that build an encrypted index with minimized leakage, and thus that attack is not applicable to EncKV.

Attacks on ORE schemes are also proposed in [10, 12]. However, we note that those attacks target on specific schemes with specific leaky information [1,6,17], i.e, orders of underlying ciphertexts or orders of some bit of ciphertexts, which will not be learned in our adopted ORE scheme. Therefore, they are not applicable to EncKV. Besides, they require auxiliary information such as inter-column correlations, which can be protected in EncKV's batch query mechanism. And intra-column correlations are also somehow protected, because query tokens of the same query attributes are not the same for different nodes. If the attackers only get the partial views from some of the server nodes, those attacks are further mitigated effectively. In the future, we will analyze EncKV's strength on defending those attacks empirically.

5. EXPERIMENTAL EVALUATION

5.1 Prototype Implementation

To assess the performance of EncKV, we implement a prototype and deploy it to Amazon Web Services. We create 4 AWS M4-xlarge instances as the clients, and a Redis (v3.2.0) cluster that consists of 9 AWS M4-xlarge instances as the nodes to store encrypted indexes and records. Each instance



(a) *Ext* index build (b) *Rng* index build (c) *Ext* query throughput (d) *Rng* query throughput

(e) *Ext* query latency (f) *Ext* query comparison (g) *Rng* query latency (h) Insertion latency

Figure 5: Performance

# Indexed values	400K	600K	800K
Size (GB)	0.012	0.018	0.024

(a) Encrypted exact-match index

# Indexed values	400K	600K	800K
Size (2bit block) (GB)	0.209	0.313	0.417
Size (4bit block) (GB)	0.399	0.599	0.799
Size (8bit block) (GB)	3.070	4.604	6.139

(b) Encrypted range-match index

Table 1: Encrypted index space consumption

is assigned with 4 vcores (2.4 GHz Intel Xeon® E5-2676 v3 CPU), 16GB RAM and 40GB SSD, and Ubuntu server 14.04 are installed. EncKV's prototype utilizes Apache Thrift (v0.9.2) to implement the remote procedure call (RPC).

EncKV uses OpenSSL (v1.01f) for the implementation of cryptographic building blocks. Secure PRF is implemented via AES cipher (128 bits). The enhanced ORE scheme is implemented on top of the implementation[5] of the ORE scheme [19]. In this evaluation, we set 8 bits as the block size for ORE encryption. In the future, more parameter settings will be evaluated. The encrypted exact-match and range-match indexes are integrated into the implementation[6] of the distributed encrypted index framework [33]. In total, EncKV contains about 10144 lines of C++ code.

5.2 Performance Evaluation

The evaluation on EncKV mainly focuses on the encrypted index and query performance.

Index evaluation: We first report the index space consumption in Table 1. For the encrypted exact-match index, the size of each entry $\langle \alpha, \beta \rangle$ is 256 bits, where α and β are 128-bit long. Table 1(a) shows that the index size increases linearly from 0.012 GB (400K indexed values) to 0.024 GB (800K indexed values). For the encrypted range-match index, each entry also needs to store ORE ciphertext ct_R for comparison. As an ORE ciphertext is encrypted by blocks, the size of ct_R depends on the length of block b. And each block ciphertext contains 2^b sub blocks, where each is 64 bit-long (truncated from AES cipher output). With α, β, a 128-bit nonce γ, and ct_R, the size of an entry for a 32-bit value is $128 + 128 + 64 \times 2^b \times 32/b + 128$ bits. As mentioned in [19], there is a tradeoff in security and space. The larger block size has stronger security while introducing more space cost, which is also shown in Table 1(b).

Figure 5(a) and Figure 5(b) measure the index building time at the client. Both time cost increases linearly with the number of indexed values. The range-match index takes more time because it needs to generate ORE ciphertexts in addition to masking the encrypted record IDs.

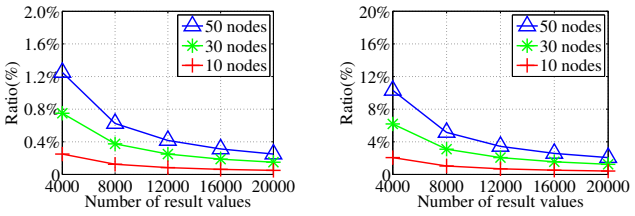
Query evaluation: To evaluate the scalability of EncKV, we evaluate the query throughput in Figure 5(c) and Figure 5(d). The result shows that the total number of index entries processed per second in both indexes increases with the number of cores. Specifically, we can find that the throughput of exact-match queries achieves up to 255K entries per second in 9 nodes, while the throughput of range-match queries is lower, 213K entries per second. The overhead comes from the cost of PRP and PRF operations in compared blocks during the ORE comparison. The results confirm that EncKV performs satisfactorily at scale.

To gain a deeper understanding on the query performance of EncKV, we evaluate the latency for exact-match and range-match queries, respectively. In Figure 5(e), we can find that as the number of nodes increases, the latency of exact-match queries that return a fixed number of results is reduced dramatically in similar proportions. The latency of exact-match queries with 32 cores is roughly half of the latency with 16 cores for returning 32K matched encrypted values. Likewise, Figure 5(g) shows that the latency of range-match queries follows a similar downward trend as the number of cores increases. When the number of compared ciphertexts is 32K, the query latency with 36 cores is around 17s which is almost one-third of the latency with 12 cores. Thus, we can confirm that EncKV benefits from the encrypted local index framework and can effectively handle queries in parallel.

Figure 5(f) compares the exact-match query performance with the scheme proposed in [33] denoted as YWWQL16,

[5]An implementation of order-revealing encryption: online at https://github.com/kevinlewi/fastore.

[6]An encrypted, distributed, and searchable key-value store: online at https://github.com/CongGroup/BlindDB.



(a) Exact-match query (b) Range-match query

Figure 6: Query token bandwidth overhead

which conducts token matching by enumerating all values on a column. Here, we pre-insert around 70K records for both designs. As seen, the query time of our design only scales in the number of matched results, while YWWQL16 scans the entire column no matter how many values are matched. Specifically, the number of matched values from 2K to 32K increases the query time from approximately 0.92s to 14.4s. In contrast, the query time of YWWQL16 is a constant-time operation, about 32.7s.

Recall that EncKV also supports incremental updates for newly added records. We accordingly evaluate the cost for index entry insertion in Figure 5(h). The comparison result shows that inserting new records into both encrypted indexes will not introduce too much overhead compared to the case without indexing. Note that such cost includes network transmission for each new indexed value, and thus it is much higher than the index building cost (bulk update). For 1000 values, it takes 0.686s and 0.875s to index entries for exact-match and range-match indexes respectively.

The adopted local index framework requires the client to generate query tokens for each node. To understand the bandwidth overhead, we show the ratio between the query token size and the result size in Figure 6. The result from Figure 6(a) indicates that the ratio of exact-match query decreases gradually with the increased size of results. Specifically, the ratio for 50 nodes drops from about 1.25% to approximately 0.25% when the number of retrieved result values increases from 4K to 20K. On the other hand, we can find that the increasing number of nodes can render a rise in the bandwidth. The ratio of 8K result size increases from around 0.13% to about 0.63% when the number of nodes rises from 10 to 50. Nevertheless, the bandwidth overhead of query tokens is negligible to the size of results. The range-match queries follow a similar trend as provided in Figure 6(b), but the corresponding ratio is higher than the exact-match queries. The reason is that the range query token contains an additional ORE ciphertext ct_L to perform ORE comparison, which enlarges the size of the query token. As shown, the ratio reaches 10.3% for 50 nodes to return 4K results.

6. RELATED WORK

Encrypted database systems: To enable rich queries over encrypted data records, a line of encrypted database systems are implemented [24, 25, 27] (just to list a few). The first fully functional system is CryptDB [27], which uses property-preserving encryption (PPE) schemes and a dedicated query planner [31] to support legacy SQL queries. Although CryptDB applies randomized encryption on top of PPE ciphertexts, the security strength after queries is reduced to the security of PPE such as DET and OPE.

BuildSeer [24] devises secure rich query protocols based on secure multi-party computation, which requires an assumption on non-colluded index and data servers. Arx [25] is proposed to minimize the leakage in range queries. It designs a scheme based on Yao's garbled circuits, where tree (range index) nodes are compared with the encoded query value via circuit evaluation. Because circuits cannot be reused for security, the circuits on tree nodes need to be updated every time they are accessed. This might introduce considerable bandwidth overhead, and limit the potential throughput of range queries. Very recently, an encrypted data analytics system called Seabed [23] is designed. It leverages a symmetric key based homomorphic encryption scheme for fast aggregation over encrypted data records. Additionally, it designs a customized schema to partition one column into multiple columns to address inference attacks. However, that customization requires prior knowledge on query and data distribution for partition, and thus it is hardly dynamic. Also, it introduces large overhead from random padding, e.g., a storage overhead of about 10×.

Searchable Symmetric Encryption: Another line of related work [4, 8, 14, 30] (just to list a few) is a cryptographic primitive for keyword search over encrypted data, i.e., searchable symmetric encryption (SSE). In [8], the security notion of SSE is formalized. And later in [14], the notion of dynamic SSE is further formalized. In [4], encrypted documents are encrypted into keyword and document ID pairs, and packing mechanisms are proposed to improve the read efficiency when the index is too large to be in memory. Other schemes like [30] considers the multi-client setting of SSE. The scheme in [30] proposed for boolean queries makes existing SSE multi-client query protocols non-interactive so as to reduce the communication overhead.

Order-revealing Encryption: Order-revealing encryption (ORE) is the primitive to enable comparison on ciphertexts for secure range queries [1, 5, 6, 17, 19, 26] (just to list a few). The early result [1], also known as order-preserving encryption (OPE), only supports numeric comparison, and the orders are directly learned from ciphertexts. To improve the security, new OPE schemes are designed [17, 26], but these schemes introduce multiple rounds of interaction (i.e., $O(logN)$, N is the number of indexed values) for each query. The first ORE scheme is proposed in [6], while this scheme leaks the first bit where two messages differ. Very recently, a new ORE scheme is proposed in [19], where the ciphertexts achieve semantic security. The comparison only shows the first different block. Concurrently, another secure ORE scheme is introduced in [5] based on pairings.

7. CONCLUSION

This paper introduces a functionally rich key-value store, called EncKV, with guaranteed data protection. EncKV stores encrypted data records with multiple secondary attributes in the form of encrypted key-value pairs. It leverages the latest cryptographic primitives (i.e., SSE and ORE) to design encrypted indexes for exact-match and range-match queries, respectively. For practical query performance, EncKV integrates those indexes into an encrypted local index framework so that each node can process queries in parallel. A formal security analysis is given to quantify the security strength of the proposed secure query protocols. EncKV's prototype is deployed on a Redis cluster, and our evaluation on Amazon AWS demonstrates its efficiency.

Acknowledgment

This work was supported in part by the Research Grants Council of Hong Kong (Project CityU 11276816 and CityU 11205014), the Natural Science Foundation of China (Project No. 61572412), Innovation and Technology Commission of Hong Kong under ITF Project ITS/307/15, and an AWS in Education Research Grant award. The authors would like to thank David J. Wu for providing us their preliminary ORE code in the early stage of EncKV's implementation.

8. REFERENCES

[1] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-Preserving Symmetric Encryption. In *Proc. EUROCRYPT*, 2009.

[2] R. Bost, P.-A. Fouque, and D. Pointcheval. Verifiable Dynamic Symmetric Searchable Encryption: Optimality and Forward Security. Cryptology ePrint Archive, Report 2016/062, 2016.

[3] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-Abuse Attacks against Searchable Encryption. In *Proc. ACM CCS*, 2015.

[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. NDSS*, 2014.

[5] D. Cash, F.-H. Liu, A. O'Neill, and C. Zhang. Reducing the Leakage in Practical Order-Revealing Encryption. Cryptology ePrint Archive, Report 2016/661, 2016.

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

[7] C. Coronel, S. Morris, and P. Rob. *Database Systems: Design, Implementation, and Management*. Cengage Learning, 2009.

[8] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. *Journal of Computer Security*, 19(5):895–934, 2011.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007.

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

[11] FoundationDB. FoundationDB: Data Modeling. Online at http://www.odbms.org/wp-content/uploads/2013/11/data-modeling.pdf, 2013.

[12] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart. Leakage-Abuse Attacks against Order-Revealing Encryption. Cryptology ePrint Archive, Report 2016/895, 2016.

[13] Information is Beautiful. World's Biggest Data Breaches. Online at http://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/, 2016.

[14] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *Proc. ACM CCS*, 2012.

[15] A. Kejriwal, A. Gopalan, A. Gupta, Z. Jia, S. Yang, and J. Ousterhout. Slik: Scalable low-latency indexes for a key-value store. In *Proc. USENIX ATC*, 2016.

[16] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill. Generic Attacks on Secure Outsourced Databases. *Proc. ACM CCS*, 2016.

[17] F. Kerschbaum. Frequency-hiding order-preserving encryption. In *Proc. ACM CCS*, 2015.

[18] N. Leavitt. Will NoSQL Databases Live Up to Their Promise? *IEEE Computer*, 43(2):12–14, 2010.

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

[20] M. Naveed, S. Kamara, and C. V. Wright. Inference Attacks on Property-Preserving Encrypted Databases. In *Proc. ACM CCS*, 2015.

[21] Oracle. Transparent Data Encryption. Online at http://www.oracle.com/technetwork/database/options/advanced-security/index-099011.html/, 2016.

[22] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, et al. The RAMCloud Storage System. *ACM TOCS*, 33(3):7, 2015.

[23] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan. Big Data Analytics over Encrypted Datasets with Seabed. In *Proc. USENIX OSDI*, 2016.

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

[25] R. Poddar, T. Boelter, and R. A. Popa. Arx: A strongly encrypted database system. Cryptology ePrint Archive, Report 2016/591, 2016.

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

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

[28] D. Pouliot and C. V. Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *Proc. ACM CCS*, 2016.

[29] Redis. An advanced key-value cache and store. Online at http://redis.io/, 2015.

[30] S.-F. Sun, J. K. Liu, A. Sakzad, R. Steinfeld, and T. H. Yuen. An efficient non-interactive multi-client searchable encryption with support for boolean queries. In *Proc. ESORICS*, 2016.

[31] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. In *Proc. VLDB*, 2013.

[32] K.-Y. Whang. NoSQL vs. Parallel DBMS for Large-scale Data Management. *DASFAA Panel*, 2011.

[33] X. Yuan, X. Wang, C. Wang, C. Qian, and J. Lin. Building an encrypted, distributed, and searchable key-value store. In *Proc. ACM AsiaCCS*, 2016.

Appendix

Definition 1. *Let $Ext = (KGen, Build_{ext}, Query_{ext})$ be EncKV's encrypted exact-match index construction. Given leakage $\mathcal{L}_1^{ext}$, $\mathcal{L}_2^{ext}$ and $\mathcal{L}_3^{ext}$, and a probabilistic polynomial time (PPT) adversary $\mathcal{A}$ and a PPT simulator $\mathcal{S}$, define the following experiments.*

Real$_{\mathcal{A}}(k)$*: The client calls $KGen(1^k)$ to output a private key K. $\mathcal{A}$ selects a dataset $\mathbf{D}$ and asks the client to build $\{I_1^{ext}, \cdots, I_n^{ext}\}$ via $Build_{ext}$. Then $\mathcal{A}$ performs a polynomial number of q adaptive queries, and asks the client for tokens and ciphertexts. Finally, $\mathcal{A}$ outputs a bit.*

Ideal$_{\mathcal{A},\mathcal{S}}(k)$*: $\mathcal{A}$ selects $\mathbf{D}$. $\mathcal{S}$ generates $\{I_1'^{ext}, \cdots, I_n'^{ext}\}$ for $\mathcal{A}$ based on $\mathcal{L}_1^{ext}$. $\mathcal{A}$ performs a polynomial number of adaptive q queries. From $\mathcal{L}_2^{ext}$ and $\mathcal{L}_3^{ext}$, $\mathcal{S}$ returns the simulated ciphertexts and tokens. Finally, $\mathcal{A}$ outputs a bit.*

Ext is adaptively secure with $(\mathcal{L}_1^{ext}, \mathcal{L}_2^{ext}, \mathcal{L}_3^{ext})$ if for all PPT adversaries $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that $Pr[\mathbf{Real}_{\mathcal{A}}(k) = 1] - Pr[\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k) = 1] \leq negl(k)$, where $negl(k)$ is a negligible function in k.

Theorem 1. *Ext* is adaptively secure with ($\mathcal{L}_1^{ext}$, $\mathcal{L}_2^{ext}$, $\mathcal{L}_3^{ext}$) under the random-oracle model if $G1, G2, H1, H2, P$ are secure PRF.

Proof. The objective is to prove that the adversary $\mathcal{A}$ cannot differentiate the real game from the simulated game defined in Definition 1. We define $\{\mathcal{H}_{G1}, \mathcal{H}_{G2}, \mathcal{H}_{H1}, \mathcal{H}_{H2}, \mathcal{H}_P\}$ are random oracles.

The simulator $\mathcal{S}$ simulates the encrypted exact-match indexes $\{I_1'^{ext}, \cdots, I_n'^{ext}\}$ on each of n nodes. For node i, $\mathcal{S}$ obtains $m_i, \langle |\alpha|, |\beta| \rangle$ from $\mathcal{L}_1^{ext}$, where m_i is the number of index entries, and $|\alpha|$ and $|\beta|$ are the bit length of encrypted label-value (LV) pair of each entry. Based on the above information, $\mathcal{S}$ inserts total m_i random LV pairs $\{\alpha, \beta\}_{m_i}$ with the same length of the real index entry to $I_i'^{ext}$.

$\mathcal{S}$ simulates the first query (v_C, C_v, C_r) that returns the encrypted values on attribute C_r if their records match v_C on attribute C_v. In particular, for node i, $\mathcal{S}$ generates simulated query tokens $t_1' = (\mathcal{H}_{G1}(k_e'||C_v||v_C||i))$ and $t_2' = (\mathcal{H}_{G2}(k_e'||C_v||v_C||i))$, where k_e' is a random string. After that, from learning the number of matched entries c_i in L_2^{ext}, $\mathcal{S}$ locates c_i index entries from $I_i'^{ext}$ by computing $\alpha' = \mathcal{H}_{H1}(t_1', c_i)$ from 0 to c_i, and returns $R'^* = \mathcal{H}_{H2}(t_2', c_i) \oplus \beta'$ at the same time. R'^* can be derived from $(\lambda, \mathcal{H}_R(k_R'||\lambda) \oplus R))$, where λ and k_R' are random stings, $\mathcal{H}_R$ is a random oracle, and R is the record ID. Next, with a random string k_l', $\mathcal{S}$ simulate $l' = \mathcal{H}_P(k_l'||C_r||R)$, and then generates a random $v^{*'}$ with the same length of v^*. From L_3^{ext}, $\mathcal{S}$ updates $M_{1,1}' = 1$ in a matrix $M_{q\times q}'$. Besides, it creates an inverted list $T'_{v^{*'}\to\alpha'}$ and inserts $v^{*'}|\alpha'$ for each $v^{*'}$.

For the subsequent jth query (j from 2 to q), if the query is conducted before, say the same as the first query, the corresponding element in $M_{1,j}'$ and $M_{j,1}'$ need to be updated to "1". All simulated query tokens and results of this query can directly be copied from the tokens and results simulated from the first query. If the query is not conducted before, the tokens and results will be simulated in the above step as shown in the first query. Note that because $T_{v^*\to\alpha}$ from L_3^{ext} tells the repeated result values queried from different attributes, $\langle l', v^{*'} \rangle$ appeared before can be obtained from $T'_{v^{*'}\to\alpha'}$.

Due to the semantic security of secure PRF, $\mathcal{A}$ cannot differentiate the simulated tokens and results from the real tokens and results. □

Definition 2. *Let $Rng = (KGen, Build_{rng}, Query_{rng})$ be EncKV's encrypted exact-match index construction. Given leakage $\mathcal{L}_1^{rng}$, $\mathcal{L}_2^{rng}$, $\mathcal{L}_3^{rng}$, and $\mathcal{L}_4^{rng}$, and a PPT adversary $\mathcal{A}$ and a PPT simulator $\mathcal{S}$, define the following experiments.*

Real$_{\mathcal{A}}(k)$*: The client calls $KGen(1^k)$ to output a private key K. $\mathcal{A}$ selects a dataset $\mathbf{D}$ and asks the client to build $\{I_1^{rng}, \cdots, I_n^{rng}\}$ via $Build_{rng}$. Then $\mathcal{A}$ performs a polynomial number of q adaptive queries, and asks the client for tokens and ciphertexts. Finally, $\mathcal{A}$ outputs a bit.*

Ideal$_{\mathcal{A},\mathcal{S}}(k)$*: $\mathcal{A}$ selects $\mathbf{D}$. $\mathcal{S}$ generates $\{I_1'^{rng}, \cdots, I_n'^{rng}\}$ for $\mathcal{A}$ based on $\mathcal{L}_1^{rng}$. $\mathcal{A}$ performs a polynomial number of non-adaptive q queries. From $\mathcal{L}_2^{rng}$, $\mathcal{L}_3^{rng}$, and $\mathcal{L}_4^{rng}$, $\mathcal{S}$ returns the simulated ciphertexts and tokens. Finally, $\mathcal{A}$ outputs a bit.*

Rng is non-adaptively secure with ($\mathcal{L}_1^{rng}$, $\mathcal{L}_2^{rng}$, $\mathcal{L}_3^{rng}$, $\mathcal{L}_4^{rng}$) if for all PPT adversaries $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that $Pr[\mathbf{Real}_{\mathcal{A}}(k) = 1] - Pr[\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k) = 1] \leq negl(k)$, where $negl(k)$ is a negligible function in k.

Theorem 2. *Rng* is non-adaptively secure with ($\mathcal{L}_1^{rng}$, $\mathcal{L}_2^{rng}$, $\mathcal{L}_3^{rng}$, $\mathcal{L}_4^{rng}$) if $G1, G2, H1, H3, P, F1, F2, F3$ are secure PRF.

Proof. The objective is to prove that the adversary $\mathcal{A}$ cannot differentiate the real game from the simulated game defined in Definition 2.

For each of n nodes, $\mathcal{S}$ iterates over q queries all at once. $\mathcal{S}$ generates random keys $\{k_r', k_o', k_R', k_1', k_2', k_3', k_l'\}$. Regarding the jth query (cmp, v_C, C_v, C_r) ($cmp \leftarrow \{<, >\}$) that returns the encrypted values on attribute C_r if their records match (cmp, v_C) on attribute C_v, $\mathcal{S}$ simulates tokens and ciphertexts from $\mathcal{L}_2^{rng}$, $\mathcal{L}_3^{rng}$, $\mathcal{L}_4^{rng}$). For node i, $\mathcal{S}$ computes $t_1' = G1(k_r', C_v||i)$, $t_2' = G1(k_r', C_v||i)$, and then computes $\alpha = H1(t_1', c_i)$ from 0 to c_i.

To simulate the ORE query token ct_L', $\mathcal{S}$ splits v_C into b blocks, and generates simulated encrypted blocks as $\tilde{v'_{|i}} = \pi(F(k_2', v_{|i-1}), v_{|i}))$, $q_i' = F2(k_3', cmp||C_v||\tilde{v'_{|i}})$, and $u_i' = F1(k_1', v_{|i-1}||\tilde{v'_{|i}}), \tilde{v'_{|i}}, q_i')$. To simulate an ORE ciphertext ct_R' on C_v, $\mathcal{S}$ obtains b_{dif} the first block that differs between ct_L and ct_R from $\mathcal{L}_3^{rng}$. Then $\mathcal{S}$ simulates 2^b sub blocks in each block. For b_{dif}, $\mathcal{S}$ simulates the matched sub block as $z' = Q1(q_i', c) + Q2(u_i', \gamma')$ where c is the counter of ct_R, and generate random strings for the rest of sub blocks. For the previous blocks, $\mathcal{S}$ simulates equal sub blocks as $z' = Q2(u_i', \gamma')$, and generates random strings for the rest. Likewise, $\mathcal{S}$ generates random strings for blocks after b_{dif}.

After that, $\mathcal{S}$ computes $\beta' = H3(t_2') \oplus (ct_R'||Enc(K_R', R))$, and inserts $\{\alpha', \beta'\}$ to $I_i'^{rng}$. Total c_i index entries will be inserted. For the matched index entries, $\mathcal{S}$ computes $l' = P(k_l', R||C_r)$. In the meantime, $\mathcal{S}$ can also generate tokens and ciphertexts appeared before from $\mathcal{L}_4^{rng}$) just like in simulating exact-match queries. When all queries are simulated, $\mathcal{S}$ inserts random index entries till $I_i'^{rng}$ has m_i entries, where m_i is obtained from $\mathcal{L}_1^{rng}$.

Due to the semantic security of secure PRF, $\mathcal{A}$ cannot differentiate the simulated tokens and results from the real tokens and results. □