Privacy-Preserving Logarithmic-time Search on Encrypted Data in Cloud

Yanbin Lu
University of California, Irvine
yanbinl@uci.edu

Abstract

Ideally, a privacy-preserving database-in-the-cloud environment would allow a database owner to outsource its encrypted database to a cloud server. The owner would retain control over what records can be queried and by whom, by granting each authorized user a search token and a decryption key. A user would then present this token to cloud server who would use it to find encrypted matching records, while learning nothing else. A user could then use its owner-issued decryption key to learn the actual matching records.

The main challenge is how to enable efficient search over encrypted data without sacrificing privacy. Many research efforts have focused on similar problems, however, none supports efficient logarithmic-complexity *search. In this paper, we construct the first provably secure logarithmic search mechanism suitable for privacy-preserving cloud setting. Specifically, we propose an efficient and provably secure range predicate encryption scheme. Based on this scheme, we demonstrate how to build a system that supports logarithmic search over encrypted data. Besides privacy guarantees, we show that the proposed system supports query authentication and secure update.*

1 Introduction

Cloud computing refers to massive computing and storage resources offering on-demand services over a network. In a cloud computing environment, data storage and software execution are outsourced to a cloud server which may comprise a group of computers. A user only needs to have a compact operating system with limited storage and computing resources.

One of the most popular and basic cloud computing services is storage-as-a-service (SAAS). We cite two examples of SAAS application scenarios. The first involves a **hospital** that maintains database of patients medical records. The hospital is the database owner that outsources the database to a cloud server. Later, physicians (database users) can access patients' records through the cloud server by searching on certain attributes, e.g. SSN, last name, DoB or age. The second example is **Personal Data Vault (PDV)** wherein database owners are individuals who outsource personal data (e.g. temperature, blood pressure or heart rate) collected from their devices. A database owner can later authorize someone (e.g. her cardiologist) to analyze this data during certain time interval, e.g. heart rate during the night.

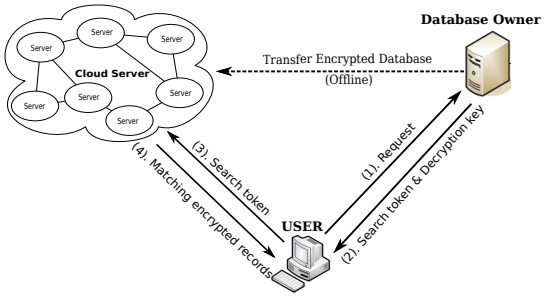


Figure 1. Idealized privacy-preserving cloud storage scenario.

Although cloud storage is an attractive concept, many prospective users are reticent about embracing it. Not surprisingly, one major concern is privacy. In our hospital scenario, a personal record contains one's medical history, including details of lifestyle habits, family medical history, lab test results, prescribed medication, etc. Such data is clearly very sensitive for most people and must be kept in confidentiality by law [2]. In the PDV example, monitoring vital signs – such as heart beat or blood pressure – reveals sensitive information about one's health.

In an idealized privacy-preserving cloud storage setting shown in Fig.1, the database owner encrypts its records under a set of searchable attributes and outsources them to a cloud server. In step 1, the user requests search authorization from the database owner who then decides whether the user is authorized. If so, in step 2, the database owner issues the user a search token and a decryption key. These two items restrict the records that can be searched and de-

crypted, respectively. In step 3, the user supplies the search token to the cloud server which allows the latter to identify all matching encrypted records. The search token reveals no information about the query. In step 4, the cloud server returns matching encrypted records to the user who decrypts them. The decryption key should only lead to the decryption of matching records and nothing else.

The main obstacle to achieving the above "nirvana" is how to conduct secure and efficient search over encrypted data. Early database-focused work does not provide provable security [15]. Recent results on provably secure search over encrypted data only support *linear-complexity* search. However, given massive (and constantly growing) amounts of outsourced data, linear search is becoming inefficient. For this reason, this paper focuses on provably secure techniques providing truly efficient and flexible search over encrypted data. Concretely, we propose a scheme that achieves this with logarithmic complexity (in the number of records).

Contribution: To achieve efficient search over encrypted data, we first design a novel cryptographic primitive – range predicate encryption. Then we use it to construct a system that supports logarithmic search, query authentication and provable data update. Furthermore, we analyze and prove security of the proposed system and evaluate its performance.

Organization: Sec. 2 overviews related work. Next, we propose an efficient range predicate encryption scheme, used as a building block in Sec. 3. Then, Sec. 4 defines the problem and the security model for our cloud system. Sec. 5 presents the logarithmic search scheme over encrypted data. Query authentication and provable data update are discussed in Sec. 6 and 7, respectively. An extension to the scheme is presented in Sec. 8. Limitation is discussed in Sec. 9. Next, Sec. 10 includes an in-depth performance evaluation. Sec. 11 concludes this paper. The appendix contains security proofs, an extention to multi-dimentional query and an example of the inner-product predicate encryption scheme.

2 Related Work

This section overviews related work; it can be skipped with no lack of continuity.

2.1 Searchable Encryption

Searchable encryption can be divided into symmetric-key and public-key versions. The former [23] allows a client to outsource its symmetrically encrypted data to an untrusted server and later to search for a specific keyword by giving the server a search token that does not reveal the keyword or any plaintext. The public-key version [10] is used in a similar scenario, except that anyone can generate and store encrypted data on an untrusted server.

We stress that these searchable encryption schemes alone do not fit our requirements. If we use the client referred in the searchable encryption as the data owner, it is clear that the data owner is unable to generate decryption keys for data users. In other words, these searchable encryption schemes lack support of decryption key delegation. Moreover, existing schemes do not support logarithmic-complexity range search.

Bellare et al. [5] proposed a deterministic efficiently searchable public-key encryption scheme. The basic idea is to attach a deterministic searchable tag that can be queried by clients to each ciphertext. Since searchable tags are deterministic, the server can organize them in a sorted way and match them in logarithmic time. Although it is efficient, this scheme has several drawbacks. First, it only supports equality search. Second, it is hard to deal with duplicate attribute values. Records with duplicate attribute values will end up with same ciphertext, exposing plaintext frequency.

Goodrich and Mitzenmacher [13] applied Oblivious RAM to cloud storage environment to hide access pattern with sublinear amortized data request cost. However, they assume the client who outsources database is the same as the one who searches database. Therefore, their solution does not fit our requirements.

2.2 Order Preserving Encryption (OPE)

OPE schemes are deterministic schemes where the encryption function preserves numerical ordering of plaintexts such that a comparison operation can be used on ciphertexts. Agrawal et al. [3] proposed the first OPE scheme for numeric data. Later, Boldyreva et al. [9] proposed another (provably secure) way to achieve the same functionality.

With OPE, achieving logarithmic-complexity search is trivial since a comparison between any two ciphertext is possible. However, OPE does not match our security requirements. First, since it is deterministic OPE cannot be IND-CPA secure. Plus, it complicates handling of duplicate values. Second, it assumes that plaintenxt domain distribution is fixed and the encryption function is aware of this distribution, which is not always possible for dynamic data.

2.3 Attribute-based encryption (ABE)

Sahai and Waters [20] introduced the concept of ABE where a user's keys and ciphertexts are labeled with sets of descriptive attributes and a particular key can decrypt a particular ciphertext only if the cardinality of the intersection of their labeled attributes exceeds a certain threshold.

Later, Goyal, et al. [14] introduced the notion of Key-Policy ABE (KP-ABE) where the trusted authority (master key owner) generates users' private keys associated with arbitrary monotonic access structures consisting of **AND, OR** or threshold gates. Only ciphertexts that satisfy the private key's access structure can be decrypted. Bethencourt, et al. [7] explored the concept of Ciphertext-Policy ABE (CP-ABE) where each ciphertext is associated with an access structure that specifies which type of secret keys can decrypt it. Ostrovsky, et al. [19] extended the result in [14] by allowing negative constraints in a key access structure.

One unfortunate drawback of ABE is that attributes are revealed in ciphertext, which is not acceptable in the cloud scenario.

2.4 Predicate Encryption

Predicate encryption can be viewed as ABE supporting attribute hiding. A ciphertext is associated with a set of hidden attributes I. The master secret key owner has fine-grained control over access to encrypted data by generating a secret key sk_g corresponding to a predicate g. sk_g can be used to decrypt a ciphertext associated with hidden attribute I if and only if $g(I) = 1$.

Shi, et al. [22] proposed a range predicate encryption scheme. Used in a cloud setting, it allows the database owner to encrypt messages under an integer attribute and store them on an untrusted server. A client then requests a key that identifies messages with integer attributes within a certain range. This scheme also supports multi-dimentional range queries. However, it does not protect token privacy, i.e. a cloud server can learn the range a client is querying. This is because encryption is public key based and a cloud server can encrypt messages under various attribute values and launch dictionary attacks against client's submitted tokens. Furthermore, this scheme cannot hide attributes for messages that are matched by a token.

Boneh and Waters [11] developed a public-key based hidden vector encryption scheme that can be extended to handle range, subset and conjunctive queries. It also hides attributes for messages that match a query. Blundo, et al. [8] proposed a private-key version of hidden vector encryption and showed that it also guaranteed security for key patterns.

Katz, et al. [16] proposed a public-key based predicate-encryption scheme that supports inner products. Attrapadung and Libert [4] improved the efficiency of the inner-product encryption by sacrificing attribute privacy. Shi, et al. [21] noticed that public-key predicate encryption might inherently reveal the query predicate inside a token and proposed a symmetric-key inner product encryption scheme to address this problem. As shown in Sec. 3.3, although inner-product predicate encryption can be extended to range predicate encryption in a straightforward way, it is too expensive to do so.

3 Range Predicate Encryption (RPE)

In this section, we introduce range predicate encryption (RPE) that is later used as a cryptographic building block in Sec. 5. Specifically, we construct an RPE scheme that improves the security definitions and performance of previous work. From now on, we use $\mathcal{DO}$ to denote database owner, $\mathcal{U}$ to denote database user and $\mathcal{S}$ to denote cloud server. Our notation is reflected in Table 1.

3.1 Definitions

We distinguish between symmetric and public-key RPE. We also distinguish between predicate and predicate-only encryption where the former decrypts data while the latter only outputs a flag indicating whether the decryption key matches the encryption predicate. For simplicity, we only define a symmetric-key range predicate-only encryption.

Definition 1. *A symmetric-key range predicate-only encryption scheme consists of the following probabilistic polynomial time algorithms.*

RPE_Setup$^1(1^k, [0, T-1])$: On input of security parameter 1^k and range $[0, T-1]$, outputs private key SK.

RPE_Encrypt$^1(SK, t)$: On input of SK and value point t, outputs ciphertext C.

RPE_ExtractKey$^1(SK, \mathcal{Q})$: On input of SK and search range $\mathcal{Q}$, outputs decryption key $sk_{\mathcal{Q}}$.

RPE_Decrypt$^1(sk_{\mathcal{Q}}, C)$): On input of decryption key $sk_{\mathcal{Q}}$ and ciphertext $C = \mathbf{RPE\_Encrypt}(SK, t)$, outputs 1 if $t \in \mathcal{Q}$ and 0 otherwise.

Note that we use superscript 1 to denote algorithms in *predicate-only* and superscript 2 to *predicate* version. In the latter, all functions are the same as in their predicate-only counterpart, except that **RPE_Encrypt**^{2} takes an additinal payload input m and **RPE_Decrypt**^{2} outputs m iff $t \in \mathcal{Q}$. We stress that a predicate version can be easily obtained from a predicate-only version using techniques such as [16].

Also note that *public-key range predicate(-only) encryption* differs from its symmetric counterpart in that an additional public key PK is generated in **RPE_Setup** and PK instead of SK is used as input to **RPE_Encrypt**. Since public-key range predicate(-only) encryption is not used in this paper, we omit its details.

3.2 Security Definitions

The following security definitions apply to symmetric-key predicate(-only) encryption: Def. 2 describes plaintext privacy and Def. 3 defines predicate privacy that is

$\mathcal{U}$	database user	$\mathcal{DO}$	database owner
$\mathcal{S}$	cloud server	$\mathcal{Q}$	a query range
q_s	start value point of $\mathcal{Q}$	q_e	end value point of $\mathcal{Q}$
T	the domain limit	h	$= \log T$
$u(v)$	unique integer assigned to node v (Sec. 3.4)	$w(v)$	binary string label of node v (Sec. 3.4)
$\mathcal{LC}(v)$	label cover of node v (Sec. 3.4)	$\mathcal{CP}(x)$	cover path for value x (Sec. 3.4)
$\mathcal{MCS}(\mathcal{Q})$	minimum cover set for range $\mathcal{Q}$ (Sec. 3.4)	$P(\cdot)$	polynomial

Table 1. Notation.

unique to the symmetric version. For simplicity, we focus on symmetric-key predicate-only encryption.

Definition 2. *A symmetric range predicate-only encryption scheme offers* selectively secure plaintext privacy *if all polynomial-time adversaries have at most a negligible advantage in the selective security game defined below:*

- **Init**: $\mathcal{A}$ submits two points $t_0, t_1 \in [0, T-1]$ upon which it wishes to be challenged.
- **Setup**: The challenger runs **RPE_Setup**$^1(1^k, [0, T-1])$ to generate SK.
- **Phase 1**: $\mathcal{A}$ adaptively issues two types of queries:
  - Decryption key query. On the ith query, range $\mathcal{Q}_i$ is submitted, such that either: $(t_0 \notin \mathcal{Q}_i) \wedge (t_1 \notin \mathcal{Q}_i)$ or $(t_0 \in \mathcal{Q}_i) \wedge (t_1 \in \mathcal{Q}_i) \wedge (z_j \notin \mathcal{Q}_i)$ for any previous ciphertext query of value point z_j. The challenger runs **RPE_ExtractKey**$^1(SK, \mathcal{Q}_i)$ and returns its output to $\mathcal{A}$.
  - Ciphertext query. On the ith query, a value point z_i is submitted such that, for any previous decryption key query of range $\mathcal{Q}_i$ where $(t_0 \in \mathcal{Q}_i) \wedge (t_1 \in \mathcal{Q}_i)$, $z_i \notin \mathcal{Q}_i$. The challenger runs **RPE_Encrypt**$^1(SK, z_i)$ and returns its output to $\mathcal{A}$.
- **Challenge**: The challenger flips a random coin b, and responds with **RPE_Encrypt**$^1(SK, t_b)$ to $\mathcal{A}$.
- **Phase 2**: $\mathcal{A}$ may continue to issue queries, subject to the same restrictions as in phase 1.
- **Guess**: The adversary outputs a guess b' for b.

The advantage of the adversary in the above game is defined as: $\mathbf{Adv}_{\mathcal{A}} = |\mathsf{Pr}[b' = b] - \frac{1}{2}|$.

We stress that the above definition is stronger than that in [22] in the sense that it allows the adversary to query a range key $sk_{\mathcal{Q}}$ where both $t_0, t_1 \in \mathcal{Q}$, so-called match-concealing security model of [22], but subject to the additional requirement that no plaintexts in $\mathcal{Q}$ have been queried before. Note that this additional requirement is not necessary for general range predicate encryption security definition and is only necessary for our scheme (Sec. 3.5) to be secure. In match-concealing model, even if a ciphertext is matched and decrypted by a key, the attribute is still hidden. Whereas, match-revealing model does not protect matched ciphertext attribute privacy. The range predicate encryption scheme presented in [22] only deals with match-revealing model.

Definition 3. *A symmetric range predicate-only scheme offers* selectively secure predicate privacy *if all polynomial-time adversaries have at most a negligible advantage in the selective security game defined below:*

- **Init**: $\mathcal{A}$ submits two ranges $\mathcal{Q}_0, \mathcal{Q}_1$ where it wishes to be challenged.
- **Setup**: The challenger runs **RPE_Setup**$^1(1^k, [0, T-1])$ to generate SK.
- **Phase 1**: $\mathcal{A}$ adaptively issues two types of queries:
  - Decryption key query. On the ith query, a range $\mathcal{Q}_i$ is submitted. Then the challenger runs **RPE_ExtractKey**$^1(SK, \mathcal{Q}_i)$ and returns its output to the adversary.
  - Ciphertext query. On the ith query, a value point z_i is submitted such that $z_i \notin \mathcal{Q}_0 \wedge z_i \notin \mathcal{Q}_1$. The challenger runs **RPE_Encrypt**$^1(SK, z_i)$ and returns its output to $\mathcal{A}$.
- **Challenge**: The challenger flips a random coin, b, and responds with **RPE_ExtractKey**$^1(SK, \mathcal{Q}_b)$ to the adversary.
- **Phase 2**: The adversary may continue to issue queries as in Phase 1.
- **Guess**: The adversary outputs a guess b' of b.

The advantage of the adversary in the above game is defined as: $\mathbf{Adv}_{\mathcal{A}} = |\mathsf{Pr}[b' = b] - \frac{1}{2}|$.

3.3 Strawman construction

This section presents a strawman construction of symmetric-key range predicate-only encryption based on the symmetric-key inner-product predicate-only encryption scheme [21] summarized in the Appx. E.

RPE_Setup$^1(1^k, [1,T])$: Outputs SK that algorithm **SSW_Setup**(1^k) outputs.

RPE_Encrypt$^1(SK, t)$: Builds a vector $\vec{x} = (x_1, \ldots, x_i, \ldots, x_T)$ where $x_i = 1$ if $i = t$ and $x_i = 0$ otherwise. Outputs what algorithm **SSW_Encrypt**$(SK, \vec{x})$ outputs.

RPE_ExtractKey$^1(SK, \mathcal{Q})$: Builds a vector $\vec{y} = (y_1, \ldots, y_i, \ldots, y_T)$ where $y_i = 0$ if $i \in \mathcal{Q}$ and $y_i = 1$ otherwise. Outputs what algorithm **SSW_ExtractKey**$(SK, \vec{y})$ outputs.

RPE_Decrypt$^1(tk_{\mathcal{Q}}, C)$: Outputs what algorithm **SSW_Query**$(tk_{\mathcal{Q}}, C)$ outputs.

It is easy to see that, if $t \in \mathcal{Q}$, the inner product $\langle \vec{x}, \vec{y} \rangle = 0$. Recall that, for inner-product predicate-only encryption, decryption works iff $\langle \vec{x}, \vec{y} \rangle = 0$. Therefore, the above strawman construction works. However, the problem is that the vector encapsulated inside the ciphertext and the decryption key have the same length as the domain limit, T. Considering the cost of **SSW_Encrypt** and **SSW_Query** is linear to the vector length, the above construction is impractical.

Note that we can also build strawman construction based on hidden vector encryption proposed in [8]. However, that scheme's cost is linear in T as well.

As a result, we need a more efficient construction. Below, we first represent integer domain in a more efficient way. Then we show how to transform the range query into an efficient inner-product query.

3.4 Efficient Representation of Ranges

A natural way to improve the efficiency of range representation is to use a *segment tree* [12] which is essentially a binary tree of height $h = \log T$. Each node v has a binary-string label $w(v)$. The root is labeled by an empty string. A non-leaf node that is labeled with the binary string w has left child labeled $w0$ and right child labeled $w1$. Each leaf node v at depth $\log T$ represents an integer value point from 0 to $T-1$ whose binary form is the same as $w(v)$. This tree is never constructed in its entirety, but is instead built up by $\mathcal{DO}$ as needed.

We say that a node v covers a value point x if the path from the root to the node representing x comes accross node v. For example, in Fig. 2(a), node labeled by '10' covers value points 4 and 5. We define *label cover* $\mathcal{LC}(v)$ as the range of labels of leaves that node v covers, which can be determined by padding $w(v)$ with $(\log T - |w(v)|)$ 0s to get the lower bound, and with $(\log T - |w(v)|)$ 1s to get the upper bound. For example, $\mathcal{LC}$('1') in Fig. 2(a) is {'100', '101', '110', '111'}.

We define $\mathcal{CP}(x)$ as the cover path for value point x, i.e. the set of nodes on the path from the root to the leaf node representing x (including leaf). Clearly, $\mathcal{CP}(x)$ is the set of nodes that cover x. As an example in Fig. 2(a), $\mathcal{CP}(5)$ includes those nodes marked by a circle. The binary-string label of each node in $\mathcal{CP}(x)$ can be easily identified by removing the tailing digit from binary representation of x one by one.

Algorithm 1: $\mathcal{MCS}$ Procedure.

input : A range $\mathcal{Q}$
output: The minimum cover set for $\mathcal{Q}$.

```
1: Interprets q_s and q_e in binary-string label form as b_s
   and b_e respectively;
2: Queue.push(root);
3: while Queue not empty do
4:    v ← Queue.top();
5:    Queue.pop();
6:    if LC(v) ⊆ [b_s, b_e] then
7:       MCS(Q) ← MCS(Q) ∪ v;
8:    else
9:       for each child v' of v do
10:         if LC(v') ∩ [b_s, b_e] ≠ φ then
11:            Queue.push(v');
12:         end if
13:      end for
14:   end if
15: end while
```

We define $\mathcal{MCS}(\mathcal{Q})$ as the minimum cover set for a range $Q = [q_s, q_e]$, i.e. the minimum required nodes that cover and *only* cover the leaf nodes representing integers in Q. Obviously, nodes in $\mathcal{MCS}(\mathcal{Q})$ follow three properties: (1) For any $v, w \in \mathcal{MCS}(\mathcal{Q})$, $\mathcal{LC}(v) \cap \mathcal{LC}(w) = \phi$; (2) $\cup_{v \in \mathcal{MCS}(\mathcal{Q})} \mathcal{LC}(v)$ covers binary-string labels of Q completely; (3) $\cup_{v \in \mathcal{MCS}(\mathcal{Q})} \mathcal{LC}(v)$ covers only binary-string labels in Q and no more. As an example, in Fig. 2(a), $\mathcal{MCS}(Q)$ for $Q = [1, 6]$ includes those nodes marked by a square.

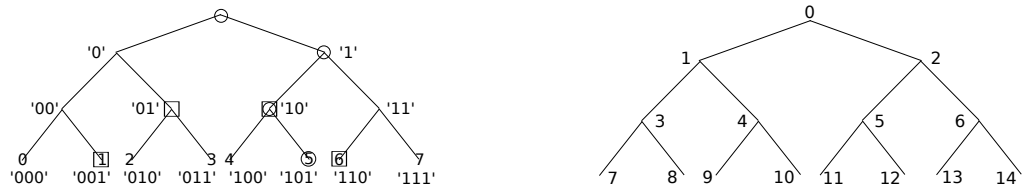
$\mathcal{MCS}(\mathcal{Q})$ can be efficiently computed by Alg 1. Given a range $Q = [q_s, q_e]$, the algorithm first interprets q_s and q_e in binary-string label form as b_s and b_e respectively. Then it traverses the tree in a breadth-first way and inserts in the $\mathcal{MCS}(\mathcal{Q})$ all nodes v such that $\mathcal{LC}(v)$ is completely contained between $[b_s, b_e]$. A node v' such that $\mathcal{LC}(v')$ does not intersect with $[q_s, q_e]$ is excluded from entering the search queue.

For cardinality of $\mathcal{MCS}(\mathcal{Q})$, we have the following theorem whose proof is shown in Appx. A.

Theorem 1. *For any range $\mathcal{Q} \in [0, T-1]$, the largest possible $|\mathcal{MCS}(\mathcal{Q})|$ is $2 \cdot (\log T - 1)$ if $T \geq 4$.*

The following proposition was proposed in [21].

Proposition 1. *If $x \in Q$, then $\mathcal{CP}(x)$ and $\mathcal{MCS}(Q)$ intersects at only one node.*



(a) Segment tree representation of $[0, 7]$

(b) Example of integer $u(v)$ for each node v

Figure 2. (a) nodes marked with circle are $\mathcal{CP}(5)$ and nodes marked with square are $\mathcal{MCS}([0, 7])$. (b) binary labels in (a) are transformed to integers.

We assign each node v a unique integer $u(v)$ in following order. Starting from the root which is assigned 0, increasing integers are assigned in zigzag fashion for each node in the tree. One example is shown in Fig. 2(b). Note that $u(v)$ for a leaf node is different from the integer the leaf represents. We use $w(v)(i)$ to denote its ith binary character (from right to left) of $w(v)$. The following lemma shows the transformation from node label to the unique integer.

Lemma 1. *Each node v's binary-string label $w(v)$ can be transformed to a unique integer $u(v)$ as follows:*

$$u(v) = \begin{cases} \sum_{i=0}^{|w(v)|-1} (2^i + 2^i \cdot w(v)(i)) & \text{if } |w(v)| > 0 \\ 0 & \text{if } |w(v)| = 0 \end{cases}$$

3.5 Improved Construction of Range Predicate Encryption

Now, we start to show how to transform a symmetric-key inner-product predicate-only encryption to a symmetric-key range predicate-only encryption. The main idea is to use a polynomial $P(\cdot)$ to capture all the nodes in $\mathcal{CP}(x)$ such that if $v \in \mathcal{CP}(x)$, then $P(u(v)) = 0$. Then we can test whether a node in $\mathcal{MCS}(\mathcal{Q})$ appears in $\mathcal{CP}(x)$ by evaluating $P(\cdot)$ at $u(v)$ for each $v \in \mathcal{MCS}(\mathcal{Q})$. Last, by expanding polynomial $P(\cdot)$, we can express the zero test of a polynomial as the zero test of an inner product. The detailed construction of each algorithm is shown below:

RPE_Setup$^1(1^k, [1, T])$: The setup algorithm outputs the SK that **SSW_Setup**(1^k) outputs.

RPE_Encrypt$^1(SK, t)$: The encryption algorithm first identifies all nodes in $\mathcal{CP}(t)$. Next it constructs a polynomial $P(X) = \prod_{v \in \mathcal{CP}(t)} (X - u(v)) = \sum_{i=0}^{h} \alpha_i X^i$ where $h = \log T$. Then it constructs a vector $\vec{x} = (\alpha_0, \ldots, \alpha_h)$. Last it runs $C \leftarrow$**SSW_Encrypt**$(SK, \vec{x})$ and outputs C.

RPE_ExtractKey$^1(SK, \mathcal{Q})$: The extraction algorithm first identifies all nodes in $\mathcal{MCS}(\mathcal{Q})$. Next it makes a set $U = \{u(v)\}_{\forall v \in \mathcal{MCS}(\mathcal{Q})}$. If U's size is smaller than $2 \cdot (\log T - 1)$, the upper bound of $\mathcal{MCS}$ cardinality, it will append U with enough random integer numbers bigger than $2 \cdot T$ (so that it will not collide with $u(v)$ for any node v). It also shuffles U for security purpose. For each $u_i \in U$, it then creates a vector $\vec{y}_i = (u_i^0, \ldots, u_i^h)$ where $h = \log T$. For each $\vec{y}_i$, it runs $sk_i \leftarrow$ **SSW_ExtractKey**$(SK, \vec{y}_i)$. Then, the decryption key for range $\mathcal{Q}$ becomes $sk_{\mathcal{Q}} = \{sk_i\}_{1 \le i \le 2 \cdot (\log T - 1)}$. Last, it outputs $sk_{\mathcal{Q}}$.

RPE_Decrypt$^1(SK, C)$: The decryption algorithm outputs 1 iff there exists $sk_i \in SK$ such that **SSW_Decrypt**(sk_i, C) outputs 1.

If we use predicate version of SSW to replace the predicate-only version of SSW, our range predicate-only encryption scheme will become range predicate encryption scheme.

We have following theorems regarding security of our range predicate-only encryption scheme and their proofs are provided in Appx. B.1 and B.2 respectively.

Theorem 2. *If SSW has selectively secure plaintext privacy and predicate privacy, then our symmetric-key range predicate-only scheme has selectively secure plaintext privacy.*

Theorem 3. *If SSW has selectively secure predicate privacy, then our symmetric-key range predicate-only scheme has selectively secure predicate privacy.*

4 Logarithmic Search on Encrypted Data (LSED)

In this section, we show how to use the range predicate(-only) encryption scheme constructed in Sec. 3.5 to build a logarithmic search over encrypted data (LSED) system.

4.1 Problem Definition

To simplify description, we define the following algorithms that form the LSED system. We use q_s and q_e to denote the boundaries of range $\mathcal{Q}$. We say a value point $t < \mathcal{Q}$ if $(t < q_s) \wedge (t < q_e)$. We say a value point $t > \mathcal{Q}$

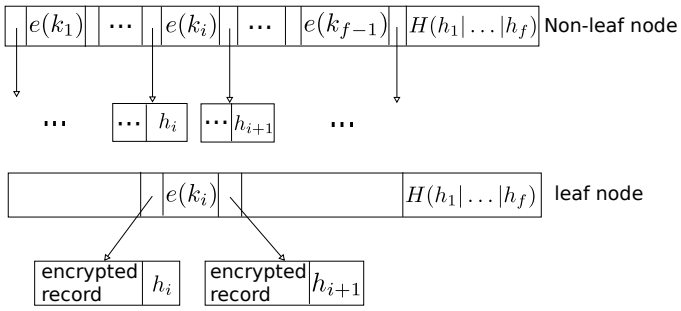


Figure 3. The encrypted B^+-tree used in LSED.

if $(t > q_s) \wedge (t > q_e)$. We refer to SEncrypt and SDecrypt as symmetric encryption and symmetric decryption respectively.

Definition 4. *A Logarithmic Search over Encrypted Data (LSED) system consists of the following probabilistic polynomial time algorithms.*

LSED_Setup$(1^k, [0, T-1])$: on input a security parameter 1^k and a range $[0, T-1]$, outputs $\mathcal{DO}$'s master private key $msk_{\mathcal{DO}}$.

LSED_Encrypt$(msk_{\mathcal{DO}}, t, m)$: $\mathcal{DO}$ on input $msk_{\mathcal{DO}}$, a value point t and a record m, outputs a ciphertext C. m can be empty.

LSED_ExtractToken$(msk_{\mathcal{DO}}, \mathcal{Q})$: $\mathcal{DO}$, on input $msk_{\mathcal{DO}}$ and a search range $\mathcal{Q}$, outputs a search token $tk_{\mathcal{Q}}$.

LSED_ExtractKey$(msk_{\mathcal{DO}}, \mathcal{Q})$: $\mathcal{DO}$, on input $msk_{\mathcal{DO}}$ and a search range $\mathcal{Q}$, outputs a decryption key $sk_{\mathcal{Q}}$.

LSED_Test$(tk_{\mathcal{Q}}, C)$: $\mathcal{S}$ on input a search token $tk_{\mathcal{Q}}$ and a ciphertext $C = \mathbf{LSED\_Encrypt}(msk_{\mathcal{DO}}, t, m)$, outputs "$>$" if $t > \mathcal{Q}$ and outputs "$<$" if $t < \mathcal{Q}$. Otherwise it outputs "$=$".

LSED_Decrypt$(sk_{\mathcal{Q}}, C)$: $\mathcal{U}$ on input a decryption key $sk_{\mathcal{Q}}$ and a ciphertext $C = \mathbf{LSED\_Encrypt}(msk_{\mathcal{DO}}, t, m)$, outputs m if $t \in \mathcal{Q}$ and $\perp$ otherwise.

At this moment, we assume that the database table has only one numeric searchable attribute. We will discuss how to deal with multi-dimentional attributes in Appx. D. Note that we require the encrypted database be organized in a B^+-tree before being transmitted to the cloud. This is necessary to facilitate logarithmic search. Possible privacy leaks are discussed in Sec. 9. We assume the searchable attributes can be encoded using discrete integers from 0 to $T-1$. For example, an IP address can be encoded using integers through $[0, 2^{32}-1]$. We will deal with real-value and string attribute in Sec. 8.

Now we discuss how the LSED system works based on the above six algorithms. We assume private communication channel (TLS/SSL) between any two entities. Before starting, $\mathcal{DO}$ runs **LSED_Setup** to initialize some parameters. Next, $\mathcal{DO}$ organizes records into a B+ tree using records' searchable attribute value as keys in order to support logarithmic search. Then $\mathcal{DO}$ encrypts each node in the B+ tree. Specifically, each node's keys are treated as value point input to $\mathbf{LSED\_Encrypt}$. Records are encrypted by $\mathbf{LSED\_Encrypt}$ with their attribute value and content as input. Fig. 3 shows an example of the resulting B+ tree node. We use $e(k_i)$ to denote the ciphertext after running $\mathbf{LSED\_Encrypt}$ over the ith key k_i in one node. We refer to f as the B^+-tree branching factor. At the end of each node, there is a Merkle tree-like hash, the purpose of which will be explained in Sec. 6. Last, $\mathcal{DO}$ outsources the encrypted B^+-tree to $\mathcal{S}$.

Whenever $\mathcal{U}$ forms a range query, it submits the range to the $\mathcal{DO}$ for authorization. If the query is approved, $\mathcal{DO}$ runs **LSED_Extract** to extract a search token and a decryption key for the queried range and gives the token and key to $\mathcal{U}$. Then, $\mathcal{U}$ hands the search token to $\mathcal{S}$ who runs a *logarithmic search* over the encrypted B^+-tree. Specifically, $\mathcal{S}$ initiates two top-down B^+-tree traversals, one to find the left most and the other to find the right most range query result. We use left boundary traversal path and right boundary traversal path to denote the nodes met in these two traversals respectively. With the help of the search token, $\mathcal{S}$ can decide whether a key, even though encrypted, is bigger or smaller than the search value embedded in the search token through **LSED_Test**. Therefore, during the B^+-tree traversal, at a node of depth p, $\mathcal{S}$ can decide the correct pointer to the node at depth $p+1$. The cost of the whole search is $O(\log_2 n)$ if we assume binary search inside each B^+-tree node. The matching encrypted results are sent back to $\mathcal{U}$ who uses the decryption key to run **LSED_Decrypt** to recover the plaintext records.

4.2 Adversary Model

We assume $\mathcal{S}$ is untrusted and can be compromised. An adversary at $\mathcal{S}$ can read all the ciphertexts stored on it and can read all search tokens transmitted from $\mathcal{U}$. The goal of the adversary is to break the ciphertext or to learn the queries issued by $\mathcal{U}$. We don't assume this type of adversary can learn useful information from the communication between $\mathcal{U}$ and $\mathcal{DO}$ since private communication channel is employed.

We also assume that $\mathcal{U}$ can be compromised and it can collude with $\mathcal{S}$. The goal of this adversary is to break the ciphertext that it is not authorized to search.

The only entity we trust is $\mathcal{DO}$ who takes care of encryption, token/key generation and database update.

4.3 Security Definition

Def. 5 describes the plaintext security, with which, $\mathcal{DO}$ can safely outsource encrypted database to the cloud.

Definition 5. *A LSED scheme has* selectively secure plaintext privacy *if all polynomial-time adversaries have at most a negligible advantage in the selective security game defined below:*

- **Init**: $\mathcal{A}$ submits two points $t_0, t_1 \in [0, T-1]$ where it wishes to be challenged.
- **Setup**: The challenger runs the **LSED_Setup**$(1^k, [0, T-1])$ algorithm to generate $params$, $msk_{\mathcal{DO}}$ and gives $params$ to $\mathcal{A}$.
- **Phase 1**: $\mathcal{A}$ adaptively issues queries, where each query is of one of three types:
  - Token query. On the ith query, a range $\mathcal{Q}_i = [q_s, q_e]$ is submitted satisfying the condition that, for any previous ciphertext query of z_i, either $(t_0 < \mathcal{Q}_i) \wedge (t_1 < \mathcal{Q}_i) \wedge z_i \geq q_s$, or $(t_0 > \mathcal{Q}_i) \wedge (t_1 > \mathcal{Q}_i) \wedge z_i \leq q_e$ or $(t_0 \in \mathcal{Q}_i) \wedge (t_1 \in \mathcal{Q}_i)$. The challenger responds with **LSED_ExtractToken(**$params, msk_{\mathcal{DO}}, \mathcal{Q}_i$**)**.
  - Key query. On the ith query, a range $\mathcal{Q}_i$ is submitted satisfying the condition that, for any previous ciphertext query of z_i, either $(t_0 \in \mathcal{Q}_i) \wedge (t_1 \in \mathcal{Q}_i) \wedge z_i \notin \mathcal{Q}_i$ or $(t_0 \notin \mathcal{Q}_i) \wedge (t_1 \notin \mathcal{Q}_i)$. The challenger responds with **LSED_ExtractKey(**$params, msk_{\mathcal{DO}}, \mathcal{Q}_i$**)**.
  - Ciphertext query. On the ith query, a value z_i and a message m_i is submitted. For any previous token query of $\mathcal{Q}_i = [q_s, q_e]$ such that $t_0 < \mathcal{Q}_i \wedge t_1 < \mathcal{Q}_i$, it is required that $z_i \geq q_s$. For any previous token query of $\mathcal{Q}_i = [q_s, q_e]$ such that $t_0 > \mathcal{Q}_i \wedge t_1 > \mathcal{Q}_i$, it is required that $z_i \leq q_e$. For any previous key query of $\mathcal{Q}_i$ such that $t_0 \in \mathcal{Q}_i \wedge t_1 \in \mathcal{Q}_i$, it is required that $z_i \notin \mathcal{Q}_i$. The challenger responds with **LSED_Encrypt(**$params, msk_{\mathcal{DO}}, z_i, m_i$**)**.
- **Challenge**: $\mathcal{A}$ outputs two equal-length messages m_0, m_1. If there is an i, in key query, for which $(t_0 \in \mathcal{Q}_i) \wedge (t_1 \in \mathcal{Q}_i)$, then it is required $m_0 = m_1$. The challenger flips a random coin, b, and responds **LSED_Encrypt**$(params, msk_{\mathcal{DO}}, t_b, m_b)$ to the adversary.
- **Phase 2**: The adversary may continue to issue queries, subject to the same restrictions as in Phase 1.
- **Guess**: The adversary outputs a guess b' of b.

The advantage of the adversary in the above game is defined as $\mathbf{Adv}_{\mathcal{A}} = |\mathsf{Pr}[b' = b] - \frac{1}{2}|$. Compared to the plaintext privacy of range predicate encryption (Def. 2), the above definition does not allow the case where $t_0 < Q_i$ and $t_1 > Q_i$ during the token query because adversary can immediately tell b by executing **LSED_Test** over the challenge ciphertext. The above definition also captures the case where $\mathcal{U}$ and $\mathcal{S}$ collude because the adversary is allowed to issue both token and key query.

The following definition describes token privacy, with which, $\mathcal{U}$ query privacy is guaranteed against a malicious $\mathcal{S}$.

Definition 6. *A LSED scheme has* selectively secure token privacy *if all polynomial-time adversaries have at most a negligible advantage in the selective security game defined below:*

- **Init**: $\mathcal{A}$ submits two ranges $\mathcal{Q}_0, \mathcal{Q}_1$ where it wishes to be challenged.
- **Setup**: The challenger runs the **LSED_Setup**$(1^k, [0, T-1])$ algorithm to generate $msk_{\mathcal{DO}}$.
- **Phase 1**: The adversary issues queries, where each query is of one of three types:
  - Token query. On the ith query, a range $\mathcal{Q}_i$ is submitted and the challenger responds with **LSED_ExtractToken(**$params, msk_{\mathcal{DO}}, \mathcal{Q}_i$**)**.
  - Key query. On the ith query, a range $\mathcal{Q}_i$ is submitted and the challenger responds with **LSED_ExtractKey(**$params, msk_{\mathcal{DO}}, \mathcal{Q}_i$**)**.
  - Ciphertext query. On the ith query, a value z_i and a message m_i is submitted. For any token query of $\mathcal{Q}_i$, it is required $z_i \in \mathcal{Q}_i$. For any key query of $\mathcal{Q}_i$, it is required $z_i \notin \mathcal{Q}_i$. The challenger responds with **LSED_Encrypt(**$params, msk_{\mathcal{DO}}, z_i, m_i$**)**
- **Challenge**: $\mathcal{A}$ submits two ranges $\mathcal{Q}_0$ and $\mathcal{Q}_1$. The challenger flips a random coin, b, and responds **LSED_ExtractToken**$(params, msk_{\mathcal{DO}}, \mathcal{Q}_b)$ to the adversary.
- **Phase 2**: The adversary may continue to issue queries, subject to the same restrictions as in Phase 1.
- **Guess**: The adversary outputs a guess b' of b.

The advantage of the adversary in the above game is defined as $\mathbf{Adv}_{\mathcal{A}} = |\mathsf{Pr}[b' = b] - \frac{1}{2}|$.

5 LSED Construction

In this section, we show how to instantiate each algorithm defined in Def. 4 based on range predicate(-only) encryption scheme proposed in Sec. 3.5.

LSED_Setup$(1^k, [0, T-1])$: on input a security parameter 1^k and a range $[0, T-1]$, it runs $SK_1 \leftarrow$ **RPE_Setup**$^1(1^k, [0, T-1])$ and $SK_2 \leftarrow$**RPE_Setup**$^2(1^k, [0, T-1])$. Then it outputs $\mathcal{DO}$'s master private key $msk_{\mathcal{DO}} \leftarrow \{SK_1, SK_2\}$.

LSED_Encrypt$(msk_{\mathcal{DO}}, t, m)$: $\mathcal{DO}$, on input master key $msk_{\mathcal{DO}}$, a value point t and a record m, generates a random 128-bit session key k. Then it runs $c_1 \leftarrow$**RPE_Encrypt**$^1(SK_1, t)$, $c_2 \leftarrow$ **RPE_Encrypt**$^2(SK_2, t, k)$ and $c_3 \leftarrow \mathsf{SEncrypt_k(m)}$. Last, it outputs $C \leftarrow \{c_1, c_2, c_3\}$. If the input record m is empty, i.e. only encrypting the key value in a B^+-tree node, only c_1 is generated. In the security definition, we assume m is nonempty.

LSED_ExtractToken$(msk_{\mathcal{DO}}, \mathcal{Q})$: $\mathcal{DO}$, on input a master key $msk_{\mathcal{DO}}$ and a search range $\mathcal{Q} = [q_s, q_e]$, constructs two seperate ranges $\mathcal{Q}^- = [0, q_s - 1]$ and $\mathcal{Q}^+ = [q_e + 1, T - 1]$ and runs $tk_{\mathcal{Q}^-} \leftarrow$ **RPE_ExtractKey**$^1(SK_1, \mathcal{Q}^-)$, $tk_{\mathcal{Q}^+} \leftarrow$ **RPE_ExtractKey**$^1(SK_1, \mathcal{Q}^+)$. Then it outputs a search token $tk_{\mathcal{Q}} \leftarrow \{tk_{\mathcal{Q}^-}, tk_{\mathcal{Q}^+}\}$.

LSED_ExtractKey$(msk_{\mathcal{DO}}, \mathcal{Q})$: $\mathcal{DO}$, on input a master key $msk_{\mathcal{DO}}$ and a search range $\mathcal{Q}$, runs $sk_{\mathcal{Q}} \leftarrow$ **RPE_ExtractKey**$^2(SK_2, \mathcal{Q})$ and outputs a decryption key $sk_{\mathcal{Q}}$.

LSED_Test$(tk_{\mathcal{Q}}, C)$: $\mathcal{S}$, on input a search token $tk_{\mathcal{Q}} = \{tk_{\mathcal{Q}^-}, tk_{\mathcal{Q}^+}\}$ and a ciphertext $C = (c_1, c_2, c_3)$, outputs "$<$" if **RPE_Decrypt**$^1(tk_{\mathcal{Q}^-}, c_1)$=1 and outputs "$>$" if **RPE_Decrypt**$^1(tk_{\mathcal{Q}^+}, c_1)$=1. Otherwise it outputs "$=$".

LSED_Decrypt$(sk_{\mathcal{Q}}, C)$): $\mathcal{U}$, on input a decryption key $sk_{\mathcal{Q}}$ and a ciphertext $C = (c_1, c_2, c_3)$, runs $k \leftarrow$**RPE_Decrypt**$^2(sk_{\mathcal{Q}}, c_2)$ and outputs $m \leftarrow \mathsf{SDecrypt}_k(c_3)$.

Note that, in the ciphertext $C = (c_1, c_2, c_3)$, we employ range predicate-only encryption for c_1, range predicate encryption for c_2 and symmetric encryption for c_3. We use c_1 for search purpose in **LSED_Test** and use c_2, c_3 for decryption purpose in **LSED_Decrypt**. Since **RPE_Encrypt** can only encrypt short-length messages, we use it to encrypt a random 128-bit session key as c_2 and use that key to further encrypt the real message as c_3. In **LSED_ExtractToken**, two range query tokens are extracted – one for ranges smaller than $\mathcal{Q}$ and one for ranges larger than $\mathcal{Q}$. Then, in **LSED_Test**, we can know whether the key embedded in a given ciphertext is smaller or larger than $\mathcal{Q}$ by running **RPE_Decrypt** over these two tokens.

We have following theorems regarding security of our LSED system and their proofs are provided in Appx. C.1 and C.2 respectively.

Theorem 4. *If range predicate-only and predicate encryption has selectively secure plaintext privacy, then our LSED scheme has selectively secure plaintext privacy.*

Theorem 5. *If range predicate-only encryption has selectively secure predicate privacy, then our LSED scheme has selectively secure token privacy.*

6 Query Authentication

Since $\mathcal{S}$ is untrusted, $\mathcal{U}$ can not simply believe the result from $\mathcal{S}$. Instead, $\mathcal{U}$ wants a proof that the result is indeed authentic, complete and fresh; this is called query authentication.

In order to achieve query authentication, we modify our B^+-tree to allow a Merkle tree-like hash in each node. Each leaf node is associated with a hash which is computed over the concatenation of the hash values of encrypted records pointed to by that node and each non-leaf node is associated with one hash which is computed over the concatenation of the hash values of its children. For example, in Fig. 3, the top node has f keys and $f+1$ pointers. Its associated hash value is computed over the concatenation of hash values of its $f+1$ children nodes. Note the way we embed Merkle tree into B^+-tree is different from MB-tree presented in [17] where one hash is associated with every pointer, instead of every node. The reason why we choose to associate one hash with every node is to allow authenticated update (Sec.7).

During the encryption phase, $\mathcal{DO}$ also computes the hash values for each node of the B^+-tree. When the encrypted Merkle B^+-tree is fully constructed, $\mathcal{DO}$ stores a copy of the root node's hash. When $\mathcal{U}$ issues a query, $\mathcal{DO}$ gives $\mathcal{U}$ the root hash value in addition to the search token and decryption key. On input $\mathcal{U}$'s search token, $\mathcal{S}$ searches for all records whose key falls within the search range and constructs a proof for the result. In detail, $\mathcal{S}$ includes in the proof one encrypted record to the immediate left and one encryped record to the immediate right of the lower-bound and upper-bound of the query result respectively. $\mathcal{S}$ also includes additional hash values necessary to help compute the root's hash, i.e. hashes of all left sibling nodes and right sibling nodes of B^+-tree left boundary traversal path and right boundary traversal path respectively. When $\mathcal{U}$ receives the proof and the query results, it first ensures that the encrypted record to the immediate left of the lower-bound is smaller than the query range and the encrypted record to the immediate right of the upper-bound is larger than the query range by running **LSED_Test** with the help of the search token. Then $\mathcal{U}$ recomputes the root hash in a bottom-up manner based on all the query result and all additional sibling hash

values. Finally, $\mathcal{U}$ compares the computed root hash to the one received from $\mathcal{DO}$. If they are the same, then the query result is authentic and fresh.

Note that we do not employ the common mechanism that requires $\mathcal{DO}$ to sign the hash root and $\mathcal{U}$ verify the signature. Instead, we let $\mathcal{U}$ fetch the latest root hash from $\mathcal{DO}$ in each query. This is because the former mechanism cannot guarantee query result freshness. If $\mathcal{DO}$ does some update to the database and $\mathcal{S}$ still keeps the old copy, $\mathcal{U}$ cannot detect that. However, our mechanism can guarantee query result freshness without introducing additional cryptographic operation.

7 Provable Data Update

7.1 Data Insertion

Suppose $\mathcal{DO}$ wants to insert data record m^* with attribute k^*. First, $\mathcal{DO}$ generates a search token tk^* for k^* and encrypts m^* as c^*. Next, $\mathcal{DO}$ constructs an insertion request message $Insert(tk^*, c^*)$ and sends it to $\mathcal{S}$. Upon receiving the insertion request, $\mathcal{S}$ first does a B^+-tree traversal to locate the leaf node where insertion should be executed based on the search token. During the traversal, $\mathcal{S}$ records information of all nodes $\mathcal{P}$ that are on the traversal path. $\mathcal{S}$ also records all hash values $\mathcal{H}$ of those sibling nodes of $\mathcal{P}$. Next $\mathcal{S}$ performs B^+-tree insertion operation, which may cause several nodes on the traversal path to split. Then $\mathcal{S}$ updates all affected nodes' hash value in a bottom-up manner until it generates the new root hash value h'_r. Finally, $\mathcal{S}$ responds to $\mathcal{DO}$ with the proof message, $ProofInsert(\mathcal{P}, \mathcal{H}, h'_r)$.

After receiving the proof from $\mathcal{S}$, $\mathcal{DO}$ generates the old root hash value h_r based on $(\mathcal{P}, \mathcal{H})$, and authenticates it by comparing it to the stored root hash value. If h_r is authentic, it means $(\mathcal{P}, \mathcal{H})$ are authentic as well. Then $\mathcal{DO}$ can verify whether $\mathcal{S}$ has performed the insertion correctly by simulating the insertion, regenerating the new root hash value using $(\mathcal{P}, \mathcal{H})$ and comparing it to h'_r. If h'_r is computed correctly, $\mathcal{DO}$ stores a copy of h'_r and finishes this operation.

Fig. 4 shows an example of B^+-tree insertion. The node associated with hash values h_r, h_3, h_1 are returned from $\mathcal{S}$ to $\mathcal{DO}$. The sibling hash values h_0, h_4 are also returned. Based on these nodes on the traversal path, $\mathcal{DO}$ can simulate the insertion operation and further compute the new hash values for h_1, h_2, h_3, h_r.

7.2 Data Deletion

Suppose $\mathcal{DO}$ wants to delete data record m^* with attribute k^*. First, $\mathcal{DO}$ generates a search token tk^* for k^*. Next, $\mathcal{DO}$ constructs a deletion request message $Delete(tk^*)$ and sends it to $\mathcal{S}$. Upon receiving the deletion request, $\mathcal{S}$ first does a B^+-tree traversal to locate the leaf node where deletion should be executed based on the search token. During the traversal, $\mathcal{S}$ records information of all nodes $\mathcal{P}$ that are on the traversal path. In addition, since B^+-tree deletion involves key redistribution and merging between immediate sibling nodes. Therefore, those affected sibling nodes information, $\mathcal{B}$, is recorded as well. $\mathcal{S}$ also records all hash values $\mathcal{H}$ of those sibling nodes of $\mathcal{P}$. Next $\mathcal{S}$ performs B^+-tree deletion operation. Then $\mathcal{S}$ updates all affected nodes' hash value in a bottom-up manner until it generates the new root hash value h'_r. Finally, $\mathcal{S}$ responds to $\mathcal{DO}$ with the proof message, $ProofDelete(\mathcal{P}, \mathcal{B}, \mathcal{H}, h'_r)$.

After receiving the proof from $\mathcal{S}$, $\mathcal{DO}$, based on $(\mathcal{P}, \mathcal{B}, \mathcal{H})$, generates the old root hash value h_r and authenticates it by comparing it to the stored root hash value. If h_r is authentic, it means $(\mathcal{P}, \mathcal{B}, \mathcal{H})$ are authentic as well. Then $\mathcal{DO}$ can verify whether $\mathcal{S}$ has performed the deletion correctly by simulating the deletion, regenerating the new root hash value using $(\mathcal{P}, \mathcal{B}, \mathcal{H})$ and comparing it to h'_r. If h'_r is computed correctly, $\mathcal{DO}$ stores a copy of h'_r and finishes this operation.

Fig. 5 shows an example of B^+-tree deletion. The node associated with hash values h_r, h_3, h_1, h_0 are returned from $\mathcal{S}$ to $\mathcal{DO}$. The sibling hash values h_2, h_4 are also returned. Based on returned nodes information, $\mathcal{DO}$ can simulate the deletion operation and further compute the new hash values for h_1, h_3, h_r.

7.3 Data Modification

The data modification is just a combination of deletion and insertion, i.e. deletion of old value and insertion of new value. Thus we omit the detail here.

8 Extension

8.1 Real-Values Attribute

Our scheme so far only supports integer attribute. In order to support real-value attribute, we need to find a way to transform them into integer values. An IEEE 754 single precision floating point number is represented in 32 bits. For a system only dealing with positive floating point numbers, simply using 32-bit integers to represent them preserves the order. Then, LSED can be directly used for encrypting positive floating point values.

For a system involving both positive and negative floating point values, however, direct interpretation as integers yields an inverse order for negative floating point values. In order to preserve the order, we subtract negative values from the largest negative (2^{31}) and add 2^{31} to each positive floating numbers. Then, LSED can be used for encrypting

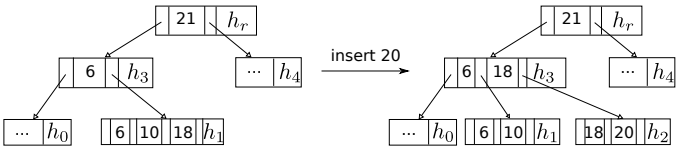


Figure 4. B^+-tree insertion example

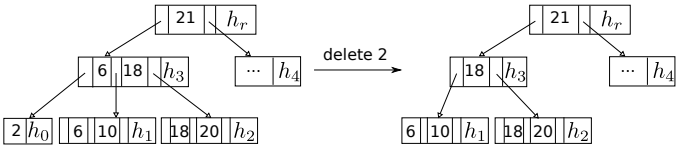


Figure 5. B^+-tree deletion example

both positive and negative floating point values. The same adjustment is needed for $\mathcal{U}$ queries as well.

The same idea applies to encrypting 64-bit double precision floating point values.

8.2 String Attribute

Since one ASCII character takes 7 bits. Any length-l ASCII strings can be encoded using integers $[0, 2^{7 \cdot l} - 1]$. After encoding, LSED can be used for encrypting string attribute.

9 Limitation

There are several limitations with our LSED system.

First, if the distribution of domain is known, a malicious cloud server can guess with high probability each plaintext value since ciphertexts are sorted in B^+-tree. This is an inherent issue with any sorted encrypted database.

Second, the cloud server will learn the access patterns of ciphertexts (i.e., which ciphertexts are more frequently queried). However we don't think leak of access pattern of encrypted database is as serious as that of plaintext database.

Third, all database update operation and query authorization relies on the database owner which becomes a single point of failure. One option is to let database owner store its master key in a smartcard. The smartcard can be encoded in a way that it only allows certain queries. Then database owner can safely hand the smartcard to users who later interact with the smartcard to get search tokens and decryption keys.

10 Performance Evaluation

We implemented our LSED system in C using PBC (ver. 0.57) [18] library. The following benchmark refers to executions on an Intel Harpertown server with Xeon E5420 CPU (2.5 GHz, 12MB L2 Cache) and 8GB RAM inside. Each data point is averaged over 10 runs.

First, we show the comparison of asymptotic performance of different range predicate encryption schemes in Table 2. As we can see, the strawman scheme has linear performance with respect to domain limit (T) in all operations. Even its ciphertext size is linear to T. SBCSP07 [22] has $O(\log T)$ performance in all operations and ciphertext size. However, it has less strong security model compared to the strawman and our scheme. Our scheme is a trade-off between the strawman and SBCSP07. It has $O(\log T)$ performance in Encrypt operation and ciphertext size. And it has $O(\log^2 T)$ performance in Decrypt and ExtractKey operations.

Next, we benchmark each algorithm of our symmetric-key range predicate encryption scheme to see its real performance. The result is shown in Fig. 6. The **RPE_Encrypt** algorithm takes a 128-bit session key as its message input. As we can see, one encryption for 32-bit domain takes less than half second. When benchmarking the decryption, we try all the $2(\log T - 1)$ keys extracted by **RPE_ExtractKey**, which is the worst-case performance. As we can see, one decryption for 32-bit domain takes less than one second. In practice, we expect average cost to be half of that. The most expensive operation comes from **RPE_ExtractKey** which needs around 20s for 32-bit domain. We will discuss how to improve that later.

Then, we benchmark each algorithm of LSED system. Fig. 7 shows the performance of each algorithm with respect to different $\log T$. As we can see, the cost of **LSED_Encrypt** is two times as expensive as that of **RPE_Encrypt**. Again, we use 128-bit session key as its input. The cost of **LSED_ExtractToken** also doubles that of **RPE_ExtractKey**. **LSED_ExtractKey** and **RPE_ExtractKey** are equally expensive. When benchmarking **LSED_Test**, we use encryption of uniformly sampled attributes and tokens for small ranges as input. It turns

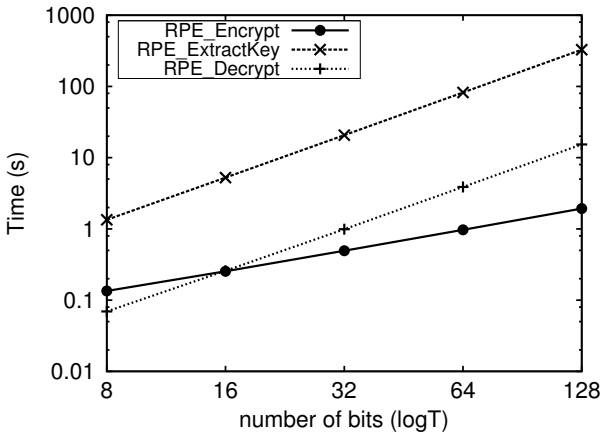


Figure 6. Performance of each algorithm in symmetric-key range predicate-only encryption scheme.

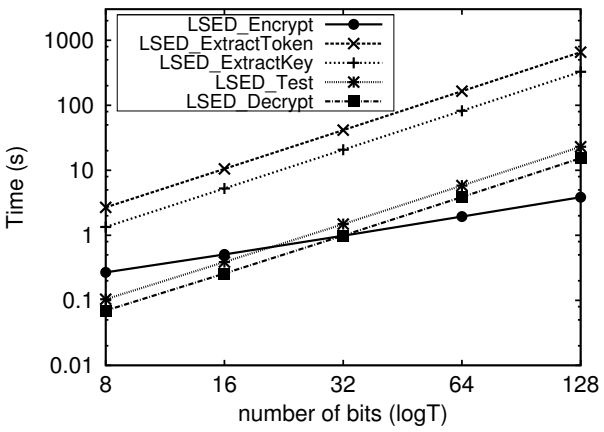


Figure 7. Performance of each algorithm in LSED system

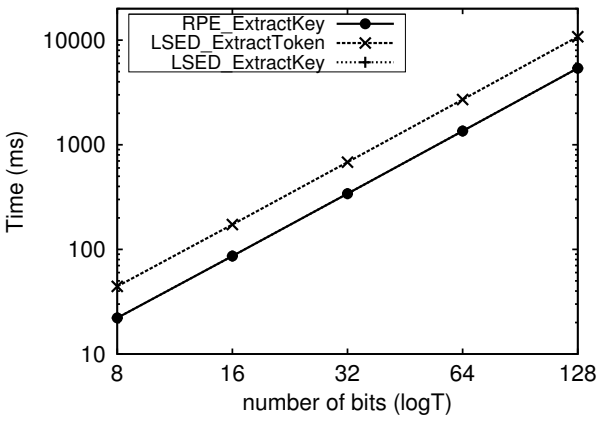


Figure 8. Performance of extraction algorithms after hardware acceleration.

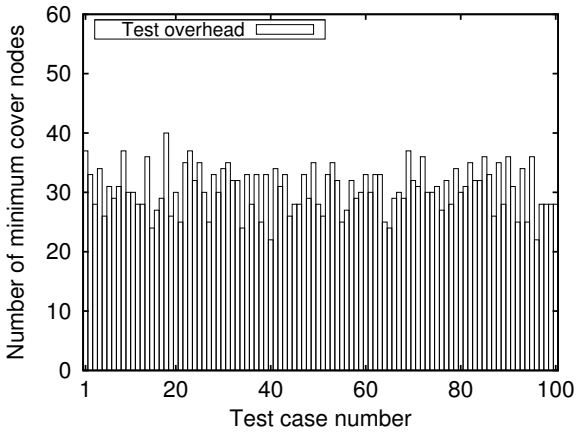


Figure 9. Number of nodes in $\mathcal{MCS}$

out that **LSED_Test** is around 1.5 times as expensive as **RPE_RangeDecrypt**. **LSED_Decrypt** is as expensive as **RPE_Decrypt**, which again shows the worst-case scenario.

From the benchmark of range predicate encryption and LSED system, we can see that the extraction of token and key is quite expensive. Since the extraction algorithm mainly consists of exponentiation operation, we can employ the accelerator chip supporting elliptic curve cryptography to improve its performance. We use the results presented in [1] to estimate the cost of our schemes. The cost of exponentiation reduces from $2.48ms$[1] to $30\mu s$. Fig. 8 shows the performance of algorithm **RPE_ExtractKey**, **LSED_ExtractKey** and **LSED_ExtractToken** after the improvement. As we can see, for $\log T = 32$, the cost of **LSED_ExtractToken** can reduce from $41s$ to $681ms$. The cost of **RPE_ExtractKey** and **LSED_ExtractKey** can reduce from $21s$ to $34ms$.

Extraction algorithm can be further improved through precomputation. Recall that, in **RPE_ExtractKey**, only those $u(v)$ where $v \in \mathcal{MCS}(\mathcal{Q})$ are useful. The random integers appended to U are for confusion purpose only. Therefore, we can precompute those keys corresponding to those appended random integers. To see how much percentage of performance we can gain, in Fig. 9, we plot the number of nodes in $\mathcal{MCS}(\mathcal{Q})$ for random chosen ranges $\mathcal{Q}$ in the full 32-bit domain. As we can see, compared to $|U|$, the aver-

[1]We consider Type A pairing family in [18] with a base field size of 512 bits here.

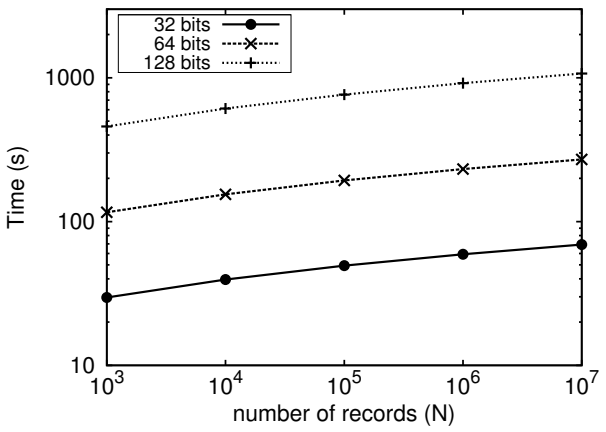


Figure 10. Performance of search.

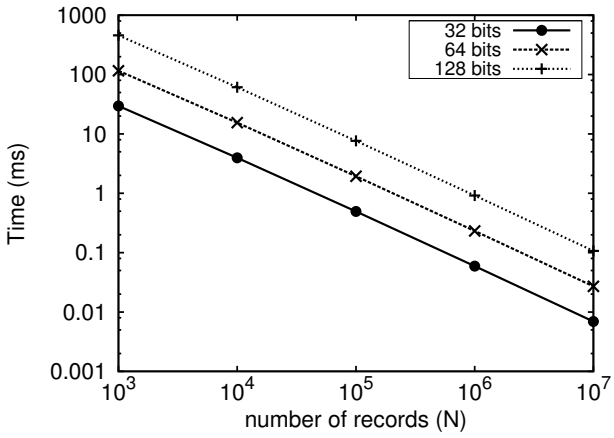


Figure 11. Performance of search time per record.

Scheme	Security Model	Encrypt Cost	Decrypt Cost	ExtractKey Cost	Ciphertext Size
Strawman (Sec. 3.3)	Match Concealing	$O(T)$	$O(T)$	$O(T)$	$O(T)$
SBCSP07 ([22])	Match Revealing	$O(\log T)$	$O(\log T)$	$O(\log T)$	$O(\log T)$
Our scheme (Sec. 3.5)	Match Concealing	$O(\log T)$	$O(\log^2 T)$	$O(\log^2 T)$	$O(\log T)$

Table 2. Asymptotic performance of different range predicate encryption.

age $\mathcal{MCS}(\mathcal{Q})$ goes around $\log T - 1$ which is half of $|U|$. This means that roughly half of the keys can be precomputed, which implies the cost shown in Fig. 8 can be further halved. In case of single value search, only one node is useful for **LSED_ExtractKey** and thus it can take as less as $5ms$ through precomputation.

In order to benchmark LSED search performance, we first build an encrypted B^+-tree with different number of records. Then we let user issue queries for arbitrary ranges. We measure the time it takes cloud server to find the correct range of nodes in the encrypted B^+-tree. Fig. 10 shows the cost of a range search with respect to the number of records in the database. As we can see, the cost increases sublinearly with the number of records. Recall that a range search involves two B^+-tree traversals, one for the left boundary and the other for the right boundary. Therefore, a single value search takes half of the time shown in Fig. 10. We further show the total search time divided by the number of records in Fig. 11. The search time per record decreases below $0.1ms$ for all three different domain sizes when total number of records in B^+-tree reaches 10 million. In other words, a linear search mechanism needs to spend less than $0.1ms$ on each record in order to beat our scheme when there are 10 million records. Further increasing the number of records can further reduce search time per record.

11 Conclusion

In this paper, we proposed a cloud storage LSED system comprising three entities – data owner, data user and cloud server. Data owner stores its data in encrypted form at cloud server. Data user gets query authorization from data owner through search token and decryption key. Cloud server can use the search token to do logarithmic search to locate matching data. During the query, cloud server learns nothing about plaintext data, nor about user's query content. User learns nothing more than what it is entitled to. In order to build LSED system, we proposed a range predicate encryption scheme that is provably secure with regard to plaintext privacy and predicate privacy. Based on that scheme, we built and proved the security of LSED system. Further, we extend LSED system to support query authentication and provable data update. Experiments show that our construction is efficient in supporting range queries over encrypted data.

References

[1] The Elliptic Semiconductor CLP-17 high performance elliptic curve cryptography point multiplier core: Product brief. `http://www.ellipticsemi.com/pdf/CLP-17_High_Performance_ECC_Point_Multiplier_Core_Rev1_0.pdf`.

[2] The federal health insurance portability and accountability act. Public Law. 104-191, 1996.

[3] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 563–574, New York, NY, USA, 2004. ACM.

[4] N. Attrapadung and B. Libert. Functional encryption for inner product: Achieving constant-size ciphertexts with adaptive security or support for negation. In P. Nguyen and D. Pointcheval, editors, *Public Key Cryptography PKC 2010*, volume 6056 of *Lecture Notes in Computer Science*, pages 384–402. Springer Berlin / Heidelberg, 2010.

[5] M. Bellare, A. Boldyreva, and A. ONeill. Deterministic and efficiently searchable encryption. In A. Menezes, editor, *Advances in Cryptology - CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 535–552. Springer Berlin / Heidelberg, 2007.

[6] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18:509–517, September 1975.

[7] J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-policy attribute-based encryption. In *Security and Privacy, 2007. SP '07. IEEE Symposium on*, pages 321 –334, May 2007.

[8] C. Blundo, V. Iovino, and G. Persiano. Private-key hidden vector encryption with key confidentiality. In J. Garay, A. Miyaji, and A. Otsuka, editors, *Cryptology and Network Security*, volume 5888 of *Lecture Notes in Computer Science*, pages 259–277. Springer Berlin / Heidelberg, 2009.

[9] A. Boldyreva, N. Chenette, Y. Lee, and A. ONeill. Order-preserving symmetric encryption. In A. Joux, editor, *Advances in Cryptology - EUROCRYPT 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 224–241. Springer Berlin / Heidelberg, 2009.

[10] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In C. Cachin and J. Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 506–522. Springer Berlin / Heidelberg, 2004.

[11] D. Boneh and B. Waters. Conjunctive, subset, and range queries on encrypted data. In S. Vadhan, editor, *Theory of Cryptography*, volume 4392 of *Lecture Notes in Computer Science*, pages 535–554. Springer Berlin / Heidelberg, 2007.

[12] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer Berlin / Heidelberg, 1997.

[13] M. T. Goodrich and M. Mitzenmacher. Mapreduce parallel cuckoo hashing and oblivious ram simulation. *CoRR*, abs/1007.1259, 2010.

[14] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, pages 89–98, New York, NY, USA, 2006. ACM.

[15] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *SIGMOD'02*, 2002.

[16] J. Katz, A. Sahai, and B. Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In *Proceedings of the theory and applications of cryptographic techniques 27th annual international conference on Advances in cryptology*, EUROCRYPT'08, pages 146–162, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 121–132, New York, NY, USA, 2006. ACM.

[18] B. Lynn. PBC: The Pairing-Based Cryptography Library. `http://crypto.stanford.edu/pbc/`.

[19] R. Ostrovsky, A. Sahai, and B. Waters. Attribute-based encryption with non-monotonic access structures. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 195–203, New York, NY, USA, 2007. ACM.

[20] A. Sahai and B. Waters. Fuzzy identity-based encryption. In R. Cramer, editor, *Advances in Cryptology EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 557–557. Springer Berlin / Heidelberg, 2005.

[21] E. Shen, E. Shi, and B. Waters. Predicate privacy in encryption systems. In O. Reingold, editor, *Theory of Cryptography*, volume 5444 of *Lecture Notes in Computer Science*, pages 457–473. Springer Berlin / Heidelberg, 2009.

[22] E. Shi, J. Bethencourt, T.-H. Chan, D. Song, and A. Perrig. Multi-dimensional range query over encrypted data. In *Security and Privacy, 2007. SP '07. IEEE Symposium on*, pages 350 –364, May 2007.

[23] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 44–, Washington, DC, USA, 2000. IEEE Computer Society.

A Proof of theorem 1

Proof. We use $\mathcal{MCS}^*_{\mathcal{T}}$ to denote the $\mathcal{MCS}(\mathcal{Q})$ with maximum cardinality for a tree $\mathcal{T}$ with height $h = \log T$. We use $root_l$ and $root_r$ to denote the left and right child of the root. $\mathcal{T}_l$ and $\mathcal{T}_r$ refers to the left and right subtree of the root. We prove $|\mathcal{MCS}^*_{\mathcal{T}}| = 2 \cdot (h-1)$ for $h \geq 2$ below.

First, we show that it is possible to construct a $\mathcal{Q}$ such that $|\mathcal{MCS}(\mathcal{Q})| = 2 \cdot (h-1)$. This $\mathcal{Q}$ can be the range accross all leaf nodes except the left most and right most leaf. The corresponding $\mathcal{MCS}(\mathcal{Q})$ can be constructed as follows: We select nodes into $\mathcal{MCS}(\mathcal{Q})$ in a top-down manner. For nodes at depth 2, we put the two most inner nodes into $\mathcal{MCS}(\mathcal{Q})$ and then remove the two subtrees rooted at these two nodes. Now, there are only 4 nodes left at depth 3. For example, in Fig.2(a), nodes with binary label '01', '10' are selected and nodes left at depth 3 are '000', '001', '110', '111'. Repeat the above procedure for depth $3, \ldots, h$. It is obvious that $\mathcal{MCS}(\mathcal{Q})$ is computed correctly.

Next, we show that, from each depth of the tree, at most two nodes are in $\mathcal{MCS}^*_{\mathcal{T}}$. At each depth, for each pair

of closest nodes in $\mathcal{MCS}^*(\mathcal{T})$ (not necessary to be neighbors), the left one must be its parent node's right side child. Otherwise, instead of this left node, its parent should be in $\mathcal{MCS}^*_{\mathcal{T}}$. This is because using parent node instead of its left node and some nodes in the right subtree can reduce the size of $\mathcal{MCS}^*_{\mathcal{T}}$ without affecting coverage. The same reason explains why the right node of the closest pair must be its parent node's left side child. If there are more than two nodes at the same depth, there must be a pair of closest nodes such that either left node is left child or the right node is right child, which is impossible.

Last we argue that the two nodes at depth 1, $root_l, root_r$, are not in $\mathcal{MCS}^*_{\mathcal{T}}$ if $h > 2$. It is obvious that they cannot be both in $\mathcal{MCS}^*_{\mathcal{T}}$. Without loss of generality, we assume $root_l$ is in $\mathcal{MCS}^*_{\mathcal{T}}$ and $root_r$ is not. Now we consider two cases: (1) $\mathcal{MCS}^*_{\mathcal{T}}$ contains nodes from both $\mathcal{T}_l$ and $\mathcal{T}_r$. Note that, in $\mathcal{MCS}^*_{\mathcal{T}}$, $root_l$ is the only node from $\mathcal{T}_l$. Assume that the number of nodes in $\mathcal{MCS}^*_{\mathcal{T}}$ from $\mathcal{T}_r$ is larger than 1. Then by symmetric mapping, the same number of nodes can appear in $\mathcal{MCS}^*_{\mathcal{T}}$ from $\mathcal{T}_l$, which means, by ruling out $root_l$, $|\mathcal{MCS}^*_{\mathcal{T}}|$ can be larger. This is contradictory to the assumption that $|\mathcal{MCS}^*_{\mathcal{T}}|$ is already maximum. Note that if the number of nodes in $\mathcal{MCS}^*_{\mathcal{T}}$ from $\mathcal{T}_r$ is equal to 1, $root_l$ can still be in $\mathcal{MCS}^*_{\mathcal{T}}$, which is only possible when $h = 2$. (2) $\mathcal{MCS}^*_{\mathcal{T}}$ contains only $root_l$, which is only possible when $h = 1$.

To sum up, from each depth of the tree except the first depth, at most two nodes are in $\mathcal{MCS}^*_{\mathcal{T}}$, which means $|\mathcal{MCS}^*_{\mathcal{T}}| \leq 2 \cdot (h-1)$. Considering there exists $|\mathcal{MCS}(\mathcal{Q})| = 2{\cdot}(h{-}1)$, we have $|\mathcal{MCS}^*_{\mathcal{T}}| = 2{\cdot}(h{-}1)$. □

B Security proof of Range Predicate Encryption

B.1 Proof of Theorem 2

Proof. First we change the game described in Def. 2 and show that the view of $\mathcal{A}$ in the original game and the new game is the same. The new game is as follows: The challenger constructs three vectors $\vec{v}_0 = \{a_0, \ldots, a_h\}$, $\vec{v}_1 = \{b_0, \ldots, b_h\}$ and $\vec{c} = \{c_0, \ldots, c_h\}$ such that $\langle \vec{a}, \vec{c} \rangle = 0$ and $\langle \vec{b}, \vec{c} \rangle = 0$. In phase 1 and 2, if $\mathcal{A}$ submits $\mathcal{Q}_i$ such that $t_0 \notin \mathcal{Q}_i \wedge t_1 \notin \mathcal{Q}_i$, the challenger still follows original game. If $t_0 \in \mathcal{Q}_i \wedge t_1 \in \mathcal{Q}_i$, the challenger constructs a set Y with $\vec{c}$ and $2\log T - 3$ random vectors inside. Then challenger shuffles set Y. Next, for each $\vec{y}_i \in Y$, the challenger runs **SSW_ExtractKey** and returns results $\{sk_i\}_{1 \leq i \leq 2(\log T - 1)}$ to $\mathcal{A}$. In the challenge phase, challenger flips a coin b and runs **SSW_Encrypt** over $\vec{v}_b$ and returns result e_b to $\mathcal{A}$.

We argue that $\mathcal{A}$ cannot computatinally differentiate the view in the original game and the new game. What $\mathcal{A}$ gets at the end is e_b and $\{sk_i\}_{1 \leq i \leq 2(\log T - 1)}$ one of which decrypts e_b. Since we have predicate privacy from SSW, $\{sk_i\}_{1 \leq i \leq 2(\log T - 1)}$ in the new game are indistinguishable from those in the original game. Due to plaintext privacy from SSW, $\mathcal{A}$ cannot see the difference between e_b in the new game and that in the original game.

Now we start to prove the new game is still secure based on SSW plaintext privacy. Suppose an adversary $\mathcal{A}$ wins the new game for ciphertext challenge with advantage ϵ. We can define an adversary $\mathcal{B}$ that wins the selective single challenge security game for SSW scheme with advantage ϵ as follows. When $\mathcal{A}$ makes a key query for $\mathcal{Q}_i$, $\mathcal{B}$ constructs vectors $\{\vec{y}_i\}_{1 \leq i \leq 2(\log T - 1)}$ according to the new game. Then, for each $\vec{y}_i$, $\mathcal{B}$ submits it to $\mathcal{B}$'s challenger as key query and responds to $\mathcal{A}$ with the keys it receives. Note that, if $t_0 \in \mathcal{Q}_i \wedge t_1 \in \mathcal{Q}_i$, $\vec{c}$ is always submitted, which guarantees returned key matches both $\vec{v}_0$ and $\vec{v}_1$. In the challenge phase, $\mathcal{B}$ outputs $\vec{v}_0$ and $\vec{v}_1$ to its challenger and responds to $\mathcal{A}$ with the answer it receives. $\mathcal{B}$ outputs the same guess b' as $\mathcal{A}$ does. It is clear that $\mathcal{B}$ wins SSW single challenge security game with the same advantage ϵ with which $\mathcal{A}$ wins the single ciphertext challenge selective security game. □

B.2 Proof of Theorem 3

Proof. Let $U_0 = \{u(v)\}_{v \in \mathcal{MCS}(\mathcal{Q}_0)}$ and $U_1 = \{u(v)\}_{v \in \mathcal{MCS}(\mathcal{Q}_1)}$ If U_0 or U_1 size is smaller than $n' = 2(\log T - 1)$, append enough random numbers bigger than $2 \cdot T$ to that size. As a result, $U_0 = \{u_{0,0}, u_{0,1}, \ldots, u_{0,n'-1}\}$, and $U_1 = \{u_{1,0}, u_{1,1}, \ldots, u_{1,n'-1}\}$.

To show that our symmetric-key range predicate-only scheme is secure, we make a hybrid argument. We define a series of games $Game_0, \cdots Game_i, \cdots, Game_{n'}$ with $Game_i$ defined as follows: During query phase, challenger honestly answers $\mathcal{A}$'s queries. During challenge phase, the challenger constructs n' decryption keys as follows. For $0 \leq j < i$, challenger constructs $\vec{y}_j = \{u^0_{1,j}, \ldots, u^h_{1,j}\}$ and, for $i \leq j < n'$, challenger constructs $\vec{y}_j = \{u^0_{0,j}, \ldots, u^h_{0,j}\}$. Then, for $0 \leq j < n'$, challenger calls $sk_j \leftarrow$ **SSW_ExtractKey**$(SK, \vec{y}_j)$. It is obvious that $Game_0$ is equivalent to the case where challenger answers **RPE_ExtractKey**$^1(SK, \mathcal{Q}_0)$ and $Game_{n'}$ is equivalent to the case where challenger answers **RPE_ExtractKey**$^1(SK, \mathcal{Q}_1)$. Suppose the negligible advantage of an adversary in SSW game is ϵ_{ssw}.

For each $0 \leq i < n'$, we construct a simulator $\mathcal{B}_i$ that reduces $Game_i$ to $Game_{i+1}$ as follows: During query phase, $\mathcal{B}_i$ forwards both key and ciphertext queries to SSW oracles and returns answers to $\mathcal{A}$. During the challenge phase, $\mathcal{B}_i$ receives $\mathcal{Q}_0, \mathcal{Q}_1$ from $\mathcal{A}$. Then $\mathcal{B}_i$ constructs $U_0 = \{u(v)\}_{v \in \mathcal{MCS}(\mathcal{Q}_0)} = \{u_{0,0}, u_{0,1}, \ldots, u_{0,n'-1}\}$ and $U_1 = \{u(v)\}_{v \in \mathcal{MCS}(\mathcal{Q}_1)} = \{u_{1,0}, u_{1,1}, \ldots, u_{1,n'-1}\}$. For

$j < i$, $\mathcal{B}_i$ queries SSW predicate oracle for vector $\vec{y}_j = \{u^0_{1,j}, \ldots, u^h_{1,j}\}$ and, for $j > i$, $\mathcal{B}_i$ queries SSW predicate oracle for vector $\vec{y}_j = \{u^0_{0,j}, \ldots, u^h_{0,j}\}$. As a result, $\mathcal{B}_i$ gets back $sk_0, \cdots, sk_{i-1}, sk_{i+1}, \cdots, sk_{n'-1}$. Then $\mathcal{B}$ outputs $\vec{x}_0 = \{u^0_{0,i}, \ldots, u^h_{0,i}\}$ and $\vec{x}_1 = \{u^0_{1,i}, \ldots, u^h_{1,i}\}$ to the SSW challenger in the challenge phase and gets back sk_i. After that, $\mathcal{B}$ outputs $\{sk_0, \cdots, sk_{n'-1}\}$ to $\mathcal{A}$. Last $\mathcal{B}$ outputs the b' that $\mathcal{A}$ outputs. When the SSW challenger chooses $b = 0$, the view of $\mathcal{A}$ is equivalent to the view in $Game_i$. When the SSW challenger chooses $b = 1$, the view of $\mathcal{A}$ is equivalent to the view in $Game_{i+1}$. Therefore the advantage of $\mathcal{A}$ differentiating the view between $Game_i$ and $Game_{i+1}$ is ϵ_{ssw}. By induction, the advantage of $\mathcal{A}$ in differentiating the view between $Game_0$ and $Game_{n'}$ is $2 \cdot (\log T - 1) \cdot \epsilon_{ssw}$ which is negligible. □

C Security Proof of LSED System

C.1 Proof of Theorem 4

Proof. Let's call the original game $\mathbb{G}_0$. We construct a game $\mathbb{G}_1$ which is the same as $\mathbb{G}_0$ except that, in challenge phase, challenger responds $C = (c_1, c_2)$ where $c_1 = \mathbf{RPE\_Encrypt}^1(SK_1, (t_1 + t_2)/2)$ and $c_2 = \mathbf{RPE\_Encrypt}^2(SK_2, t_b, m_b)$. We show a simulator $\mathcal{B}$ which reduces breaking range predicate-only encryption to distinguishing between $\mathbb{G}_0$ and $\mathbb{G}_1$. In the setup phase, $\mathcal{B}$ generates SK_2. In phase 1 and 2, $\mathcal{B}$ delegates the queries to corresponding oracles. On input t_0, t_1, in the challenge phase, $\mathcal{B}$ flips a random coin b and creates $c_2 = \mathbf{RPE\_Encrypt}^2(SK_2, t_b, m_b)$. Then $\mathcal{B}$ submits t_{b} and $(t_1 + t_2)/2$ to its challenger which flips another coin b. If $b = 0$, the challenger encrypts $(t_1 + t_2)/2$ and otherwise it encrypts t_{b}. Then it returns the result to $\mathcal{B}$ as c_1. Finally, $\mathcal{B}$ responds $C = (c_1, c_2)$ to $\mathcal{A}$. It is easy to see that, when $b = 1$, the view of $\mathcal{A}$ is the same as that in $\mathbb{G}_0$. When $b = 0$, the view of $\mathcal{A}$ is the same as that in $\mathbb{G}_1$. Therefore the probability of differentiating $\mathbb{G}_0$ from $\mathbb{G}_1$ is negligible.

Now we construct another simulator $\mathcal{B}_2$ to reduce range predicate encryption plaintext privacy game to $\mathbb{G}_1$. In the setup phase, $\mathcal{B}_2$ creates SK_1. In phase 1 and phase 2, $\mathcal{B}_2$ honestly answers all token queries and forwards all key queries to its challenger. In the challenge phase, $\mathcal{B}$ constructs $c_1 = \mathbf{RPE\_Encrypt}^1(SK_1, (t_1 + t_2)/2)$ and forwards (t_0, t_1) to its challenger which outputs c_2. $\mathcal{B}$ responds $C = (c_1, c_2)$ to $\mathcal{A}$ and outputs the bit $\mathcal{A}$ outputs. It is obvious the advantage of $\mathcal{A}$ in $\mathbb{G}_1$ is the same as that in range predicate game, which is negligible. Therefore, the advantage of $\mathcal{A}$ in $\mathbb{G}_0$ is also negligible. □

C.2 Proof of Theorem 5

Proof. Let's call the original game $\mathbb{G}_0$. Let $\mathcal{Q}_0 = [q_{0,s}, q_{0,e}]$ and $\mathcal{Q}_1 = [q_{1,s}, q_{1,e}]$. Let Q' denote $[(q_{0,s} + q_{1,s})/2, (q_{0,e} + q_{1,e})/2]$. We construct a game $\mathbb{G}_1$ which is the same as $\mathbb{G}_0$ except that, in challenge phase, $(tk_{\mathcal{Q}'^-}, tk_{\mathcal{Q}_b^+})$ instead of $(tk_{\mathcal{Q}_b^-}, tk_{\mathcal{Q}_b^+})$ is returned. We show a simulator $\mathcal{B}_1$ which reduces breaking range predicate-only encryption to distinguishing between $\mathbb{G}_0$ and $\mathbb{G}_1$. In phase 1 and 2, $\mathcal{B}_1$ delegates the queries to corresponding oracles. On input $\mathcal{Q}_0, \mathcal{Q}_1$ in the challenge phase, $\mathcal{B}_1$ flips a random coin b and poses $\mathcal{Q}_{\mathsf{b}}^+$ to range predicate-only token oracle which returns $tk_{\mathcal{Q}_{\mathsf{b}}^+}$. Then $\mathcal{B}$ sends $(\mathcal{Q}'^-, \mathcal{Q}_{\mathsf{b}}^-)$ to its challenger which flips another coin b and responds with $\mathcal{Q}_b^-$ where $\mathcal{Q}_0^- = \mathcal{Q}'^-$ and $\mathcal{Q}_1^- = \mathcal{Q}_{\mathsf{b}}^-$. $\mathcal{B}$ outputs $(\mathcal{Q}_b^-, \mathcal{Q}_{\mathsf{b}}^+)$ to $\mathcal{A}$. When $\mathcal{A}$ outputs its guess b', $\mathcal{B}$ outputs b' as well. It is easy to see that, when $b = 1$, the view of $\mathcal{A}$ is the same as that in $\mathbb{G}_0$. When $b = 0$, the view of $\mathcal{A}$ is the same as that in $\mathbb{G}_1$. Therefore the probability of differentiating $\mathbb{G}_0$ from $\mathbb{G}_1$ is negligible.

Similarly, we can construct another game $\mathbb{G}_2$ which, in the challenge phase, returns $(tk_{\mathcal{Q}'^-}, tk_{\mathcal{Q}'^+})$ and we can prove $\mathbb{G}_2$ is indistinguishable from $\mathbb{G}_1$. It is easy to see, in $\mathbb{G}_2$, $\mathcal{A}$'s advantage is negligible. Therefore, $\mathcal{A}$'s advantage in $\mathbb{G}_0$ is also negligible. □

D Extension to Multi-dimensional Query

For a database with multiple searchable attributes, it is easy to reuse the same algorithm shown in Sec. 5 as long as the user is only doing search over one specific attribute. An independent encrypted B^+-tree needs to be constructed for each attribute. Each internal node of B^+-tree of a specific attribute is encrypted under that attribute.

When the user wants to do a disjunctive query over multiple attributes, the database owner can simply invoke $\mathbf{LSED\_ExtractToken}$ and $\mathbf{LSED\_ExtractKey}$ algorithm to generate tokens and keys for each attribute of the query. Then the user can let cloud server try searching in each attribute of the query. Whenever there is a match, the user can use the matched attribute's key to decrypt.

When it comes to conjunctive query, logarithmic search is still possible if a k-d tree [6] is employed. Before encryption, the whole database's records are organized into a k-d tree. Recall that, in a k-d tree, each node is a k-dimensional point that divides the space into two parts through one of the k dimensions. Each dimension corresponds to each attribute of the database table. Then we encrypt each internal node of the k-d tree under every attribute and outsources the encrypted k-d tree to the cloud. When the user poses a conjunctive query, the database owner can generate the tokens and decryption keys for each attribute of the query. Then

cloud server can use search tokens to go through the k-d tree and efficiently locate the matching records whose attribute values fall in the ranges of every attribute of the query. The user can use the decryption keys for any attribute to decrypt the matching records.

However, the above two methods pose some privacy leaks. For disjunctive queries, the user learns which attribute of the query matches the result. For conjunctive queries, if the user and the cloud server collude, then the cloud server can have decryption keys for each attribute of the query and can decrypt records that match any attribute in the query. In other words, the cloud server and the user together learn more than what the user is entitled to.

E Inner-Product Predicate Encryption Scheme (SSW)

We review the construction of SSW [21] symmetric-key predicate-only encryption scheme for inner product queries.

Let $\mathcal{G}$ denote a group generator algorithm for a bilinear group whose order is the product of four distinct primes.

SSW_Setup(1^k): The setup algorithm runs $\mathcal{G}(1^k)$, where k is a security parameter, to obtain $(p, q, r, s, \mathbb{G}, \mathbb{G}_T, e)$ with $\mathbb{G} = \mathbb{G}_p \times \mathbb{G}_q \times \mathbb{G}_r \times \mathbb{G}_s$. Next it picks generators g_p, g_q, g_r, g_s of $\mathbb{G}_p, \mathbb{G}_q, \mathbb{G}_r, \mathbb{G}_s$, respectively. It chooses $(h_{1,i}, h_{2,i}, u_{1,i}, u_{2,i}) \in (\mathbb{G}_p)^4$ uniformly at random for $i = 1$ to n. The secret key is

$$SK = (g_p, g_q, g_r, g_s, \{h_{1,i}, h_{2,i}, u_{1,i}, u_{2,i}\}_{i=1}^n).$$

SSW_Encrypt$(SK, \vec{x})$: Let $N = pqrs$. Let $\vec{x} = (x_1, \ldots, x_n) \in \mathbb{Z}_N^n$. The encryption algorithm chooses random $(y, z, \alpha, \beta) \in \mathbb{Z}_N$, random $(S, S_0) \in (\mathbb{G}_s)^2$, and random $(R_{1,i}, R_{2,i}) \in (\mathbb{G}_r)^2$ for $i = 1$ to n. It outputs the ciphertext

$$CT = \begin{pmatrix} C = S \cdot g_p^y, \\ C_0 = S_0 \cdot g_p^z, \\ \{C_{1,i} = h_{1,i}^y \cdot u_{1,i}^z \cdot g_q^{\alpha x_i} \cdot R_{1,i}, \\ C_{2,i} = h_{2,i}^y \cdot u_{2,i}^z \cdot g_q^{\beta x_i} \cdot R_{2,i}\}_{i=1}^n \end{pmatrix}$$

SSW_ExtractKey$(SK, \vec{v})$: Let $\vec{v} = (v_1, \ldots, v_n) \in (\mathbb{Z}_N)^n$. The token generation algorithm chooses random $f_1, f_2 \in \mathbb{Z}_N$, random $(r_{1,i}, r_{2,i}) \in (\mathbb{Z}_N)^2$, random $(R, R_0) \in (\mathbb{Z}_r)^2$, and random $(S_{1,i}, S_{2,i}) \in (\mathbb{Z}_r)^2$ for $i = 1$ to n. It outputs the token

$$SK_{\vec{v}} = \begin{pmatrix} K = R \cdot \prod_{i=1}^n h_{1,i}^{-r_{1,i}} \cdot h_{2,i}^{-r_{2,i}}, \\ K_0 = R_0 \cdot \prod_{i=1}^n u_{1,i}^{-r_{1,i}} \cdot u_{2,i}^{-r_{2,i}}, \\ \{K_{1,i} = g_p^{r_{1,i}} \cdot g_q^{f_1 v_i} \cdot S_{1,i}, \\ K_{2,i} = g_p^{r_{2,i}} \cdot g_q^{f_2 v_i} \cdot S_{2,i}\}_{i=1}^n \end{pmatrix}$$

SSW_Decrypt$(SK_{\vec{v}}, C)$:

Let $CT = (C, C_0, \{C_{1,i}, C_{2,i}\}_{i=1}^n)$ and $SK_{\vec{v}} = (K, K_0, \{K_{1,i}, K_{2,i}\}_{i=1}^n)$ as above. The query algorithm outputs 1 iff

$$\hat{e}(C, K) \cdot \hat{e}(C_0, K_0) \cdot \prod_{i=1}^n \hat{e}(C_{1,i}, K_{1,i}) \cdot \hat{e}(C_{2,i}, K_{2,i}) \stackrel{?}{=} 1,$$

which is only possible when $\langle \vec{x}, \vec{y} \rangle = 0 \mod N$.