SOREL: Efficient and Secure ORE-based Range Query over Outsourced Data

Songnian Zhang, Suprio Ray, *Member, IEEE,* Rongxing Lu, *Fellow, IEEE*

Abstract—Outsourcing data to the cloud has become popular due to the big data challenges. However, security concerns compel the outsourced data to be encrypted before sending them to the cloud, which lowers their utility and efficiency. The range query plays a significant role in common queries. Consequently, how to efficiently support the range query over encrypted data has become an important challenge. Previously reported schemes either achieve efficiency only in a specific operation or have severe defects in scalability. To address these limitations, we propose a framework, called SOREL, which simultaneously considers security, efficiency and scalability. Specifically, we first propose a new efficient and Secure Order Revealing Encryption (SORE) scheme, which is more secure than bit-based ORE schemes. Then, by employing the proposed SORE scheme, we design a novel index within our framework SOREL to support efficient updating and query operations over encrypted data. Detailed security analysis shows that our SOREL achieves the desirable security requirements. Additionally, results from extensive evaluations indicate that i) SORE outperforms other alternative schemes by at least $8\times$; and ii) SOREL is at least $3\times$ faster than the comparative schemes with range query operation in the best case while ensuring the competitiveness with insertion operation.

Index Terms—Secure ORE, Efficient Range Query, Outsourced Data, Index Structure.

1 Introduction

With the rapidly growing data volume, outsourcing data to the cloud has become a popular option that many individuals and companies are tending to consider today. In such a scenario, it is crucial to ensure the security of outsourced data since they could be either stolen by external attackers or snooped by semi-trusted cloud providers. To address these security issues, an intuitive approach is to encrypt all data before outsourcing them to the cloud. However, traditional encryption schemes prevent the cloud from directly processing queries over the encrypted data, which obviously lowers the data utility. Therefore, ensuring the utility of encrypted outsourced data is of great importance to the success of outsourcing data to the cloud.

Although the point query can be easily dealt with using deterministic encryption schemes, it is still challenging to deal with the range query using mainstream symmetric or asymmetric encryption schemes. The main reason is that it is quite difficult to determine the order relation only by comparing ciphertexts. However, the support of ciphertext-based range query is indispensable, because if the range query over encrypted data (henceforth *range query* will refer to *range query over encrypted data*) can be properly resolved, then other complex comparison-based operations, such as sort and clustering, can also be resolved. A plausible approach is to adopt Homomorphic Encryption (HE) [1] to deal with the challenge. However, the HE-based solutions are impractical due to the prohibitive computational overhead. Similarly, Garbled Circuit (GC) based range query approaches [2] suffer from performance issues and are inefficient in insertions. However, it is challenging to adopt these solutions in real-world databases, because efficient search and insertion operations are critical to populating databases. Considering practicability, it is becoming urgent to ensure efficiency for the encrypted databases, since they inevitably incur more computation compared with original databases.

Searchable Symmetric Encryption (SSE) [3], [4] has been utilized to support efficient range query and insertion, which can take full advantage of hash tables by scanning all values in the query range. However, these schemes are quite limited due to the fact that they can only support enumerable fields.

Property preserving encryption (PPE) [5] is another approach proposed to enhance the efficiency of searching over encrypted data. It can enable the ciphertexts to maintain a particular property on their underlying plaintexts. For instance, order preserving encryption (OPE) [6] is a typical cryptographic primitive for PPE and preserves the order relations of plaintexts. Although such encryption schemes reveal some properties of plaintexts, they are still attractive due to the high efficiency and practicability. Recently, the Microsoft researchers released a secure cloud database solution [7], in which the order relations are revealed for efficiency considerations. Since the OPE ciphertexts can directly reveal the order relations of plaintexts, several range query schemes were proposed [8], [9] based on OPE. Even some secure databases, e.g., CryptDB [10], also integrated OPE as the cryptographic primitive into their systems to support the range query. However, Boldyreva et al. [11] proved that there does not exist an efficient OPE scheme that satisfies the security notion of IND-OCPA (*indistinguishability under ordered chosen-plaintext attack*). Additionally, only supporting the numeric type range queries is another weakness of OPE. To address these limitations, order revealing encryption (ORE) was presented in [12]. As the name suggests, this encryption primitive reveals order relations in a similar way as OPE. The key difference is that, for ORE, the order prop-

• *S. Zhang, S. Ray, and R. Lu are with the Faculty of Computer Science, University of New Brunswick, Fredericton, NB E3B 5A3, Canada (e-mail: szhang17@unb.ca, sray@unb.ca, rlu1@unb.ca).*

erty is exposed by a public comparison function, while the ciphertext of OPE can reveal the plaintext order directly. The existing ORE schemes can support any data type (since it is designed for the bit sequence) and are more efficient than OPE. However, when it comes to security, to the best of our knowledge, none of the ORE schemes satisfies IND-OCPA, because they leak additional information in the plaintext. The schemes proposed in [13], [14] will disclose the most significant differing bit or block, and the schemes presented in [15], [16] will leak "equality pattern" of the most significant differing bit. Since the most significant differing bit reveals the difference among plaintexts, it will facilitate the adversary to infer the underlying plaintexts from ORE-based ciphertexts. Thus, although the ORE schemes are efficient enough, they leak too much information.

In this paper, we first propose a novel ORE-based encryption scheme, called Secure Order Revealing Encryption (SORE), to address the security deficiencies of the existing ORE schemes. In this way, we can inherit the high efficiency of ORE schemes while enhancing their security. With SORE, the adversaries are unable to determine the real first differing bit through comparing ciphertexts. Our scheme first encodes and expands the plaintext bit sequence into a new sequence, which is called *order-preserving bit sequence*. Afterward, the new sequence is encrypted bit by bit and form the final ciphertext. The details of SORE are presented in Section 3. For most security enhancement schemes, the challenging part is improving security while still ensuring efficiency. Note that our proposed SORE scheme can deal with the challenge, i.e., experimental evaluation results indicate SORE can hide the most significant differing bit only at the cost of an extra 10% overhead when the plaintext is greater than 24 bits.

Then, we propose a SORE-enabled framework, called SOREL, to support the efficient point and range query over encrypted data (details are in Section 4). For the point query, the hash table based index (SOREL$_{eq}$ in Fig. 4) can be employed to achieve $O(1)$ efficiency. To support range query efficiently, a number of ordered indices were proposed, such as B+tree [17], Masstree [18] and Adaptive Radix Tree [19]. However, the time complexity of these schemes is $O(\log N)$, where N is the number of elements in the dataset. In this paper, we introduce a novel index structure to efficiently support the range query over SORE based ciphertexts. At first, the idea of locality preserving hashing [20] is combined with the hash table to form a two-level range index component (SOREL$_{rq}$ in Fig. 4), where the first level is a hash table, and the second level consists of several blocks generated by aggregating the dataset. Then, to further improve the performance, we adopt some techniques to optimize the prototype index, which enables the worst-case time for looking up a predecessor or successor to take only $O(1) + O(\log B)$, where B is the block size in the two-level range index component. Additionally, to tackle security and efficiency issues incurred by non-uniform data distribution, we propose a step function based distribution suppression scheme to hide the data distribution. Note that some existing indexing approaches for encrypted data are only efficient in either insertion [9] or query [21], whereas our SOREL framework achieves high efficiency in both query and insertion. In summary, the main contributions of this paper are as follows.

- First, we propose a secure ORE scheme, called SORE, to enhance the security of original ORE by addressing the first differing bit leakage. This encryption scheme is employed as the cryptographic primitive in our SOREL to encrypt the outsourced data.
- Second, we design an efficient two-level index structure to support the range query within our SOREL framework. In particular, we introduce the concept of *order aggregating hash function* to bridge the gap between the range query and hash tables. With this function, a dataset can be divided into multiple blocks, and then the order aggregating hash is used to quickly locate one block for further searching.
- Third, we propose a *distribution suppression scheme* to hide the data distribution of a dataset, which can further improve the security and efficiency of our SOREL framework.
- Finally, we implement our SOREL framework and experimentally evaluate it with different datasets. The results show that i) the proposed SORE scheme outperforms alternative schemes by at least 8$\times$ in terms of the average execution time, and ii) our index structure over encrypted data is faster than B+tree-based ORE protocol [22] and Logarithmic-BRC [3] by up to 7$\times$ and 3$\times$, in terms of range query execution time, respectively.

The remainder of this paper is organized as follows. We start with an overview of threat model and SOREL architecture in Section 2. Section 3 introduces our proposed SORE scheme. In Section 4, we describe our novel index structure together with the details of our SOREL framework. Then, we analyze SOREL in terms of security and performance in Section 5. Experimental evaluation is presented in Section 6. Finally, we discuss some related works in Section 7 and draw our conclusion in Section 8.

2 Overview

In this paper, we consider a typical cloud-based *3-role* model, including a data provider, a cloud server and data users, in which the data provider is responsible for uploading his/her data to the cloud, while the utilization is initiated by the data users (authorized with secret keys by the data provider). A typical application scenario for this model is that some enterprises release their datasets to the cloud for their users to query. In such a scenario, the queries are high frequent and happen in real-time, while the insertions are infrequent after the first data release. In this work, we target this type of scenario. Based on the above model and scenario, we will formalize the threat model and the overview of SOREL in this section.

2.1 Threat model

In our threat model, the cloud server is assumed semi-honest, which means it will faithfully follow the protocols but may be curious about the outsourced data and query values. On the other hand, the data provider and the data users are assumed to be trusted. There are two adversaries considered: i) external attackers; ii) the semi-honest cloud

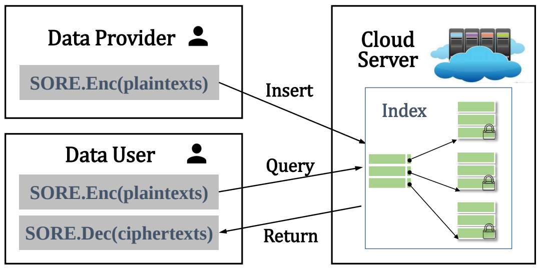

Fig. 1: Overview of SOREL

server. External attackers may break the defense system and pry into sensitive data on the transmission or storage stage, while the cloud server attempts to obtain and infer the value of data items. To defend against these adversaries, a straightforward solution is to apply cryptographic primitives to the outsourced data, so that only the data provider and the authorized users can decrypt it. However, it can entail two issues. First, the cloud server needs to support basic operations over encrypted data. Second, the employed cryptographic primitive may lead to performance overhead. Besides, the generated ciphertexts may not work with the existing index schemes, ensuing lower efficiency when using data.

2.2 Overview of SOREL

In our SOREL framework, we adopt cryptographic techniques to deal with the security issues for outsourced data and take into account high efficiency. In particular, we introduce our secure order revealing encryption, called SORE, which is employed as the cryptographic primitive to encrypt the outsourced data. To improve efficiency, we design a novel index structure to support insertion, query, and deletion operations over these encrypted data. Hence, there are two core components in SOREL: namely (i) SORE encryption scheme (Section 3) and (ii) Index structure (Section 4). As shown in Fig. 1, the data provider employs the SORE encryption module for encrypting the outsourcing data, while the data user holds SORE encryption and decryption modules for encrypting query values and recovering plaintexts, respectively. In SOREL, the index structure can be built and updated by the cloud server over the encrypted data, which can avoid extra communication. Furthermore, our devised index structure considers the non-uniform data distribution. By integrating a distribution suppression scheme, our SOREL framework can securely and efficiently deal with skewed data. It is worth noting that all data transferred to the cloud server and responded to the data user are in ciphertexts.

3 SORE: Secure ORE-based Encryption

In this section, we review the essential concepts of ORE and then introduce our security enhanced ORE scheme, SORE.

Preliminaries. For $n \in \mathbb{N}^{+}$, we denote M=$\{m_i\}_{i=0}^{n}$ as a set of plaintexts, and C=$\{c_i\}_{i=0}^{n}$ as the corresponding ciphertexts, where each c_i=$Enc(m_i)$. We say $Compare(\cdot)$ is a common comparison function that can reveal the order relation of two input parameters, and $\mathsf{len}()$ is a function to return the length of a bit sequence. We also define that *prefix* and *suffix* represent two fixed bit sequences, and $\mathsf{len}(prefix) = \mathsf{len}(suffix) = \ell$.

3.1 ORE

The motivation behind ORE is to remedy the inherent defects of OPE in terms of performance and scalability. Unlike OPE, the ORE ciphertext does not reveal the order of underlying plaintext directly. Only by comparing two ciphertexts with a publicly computable comparison function can we reveal the order of their corresponding plaintexts. Typically, an order revealing encryption is a tuple $\prod_{ORE} = \{ORE.Setup, ORE.Encrypt, ORE.Compare\}$ [13].

- *ORE.Setup*(1^{λ}). The algorithm is responsible for generating the secret key sk for the encryption algorithm.
- *ORE.Encrypt(sk, m)*. Using the secret key sk, this algorithm encrypts a message m_i into a ciphertext, c_i=$ORE.Encrypt(sk, m_i)$ (i=$0, 1, 2 \ldots n$).
- *ORE.Compare(c_i, c_j)*. Generally, the algorithm takes two ciphertexts, e.g., $(c_i, c_j) \in C$, as input, and outputs the order relation of $(m_i, m_j) \in M$.

Here we do not describe the *ORE.Decrypt(sk, c)* algorithm, as it is relatively simple to deduce *ORE.Decrypt(sk, c)* from *ORE.Encrypt(sk, m)*. For simplicity, henceforth, we ignore the secret key in *ORE.Encrypt*(sk, m).

Note that, for any ORE scheme, the following correctness equation must be satisfied:

$$ORE.Compare(c_i, c_j) = Compare(m_i, m_j) \tag{1}$$

where $c_i = ORE.Encrypt(m_i)$ and $c_j = ORE.Encrypt(m_j)$. Currently, there are several approaches to design ORE schemes. The most practical one, called bit-based ORE, is to convert a message into a bit sequence, and encrypt these bits one by one. Specifically, give a message m, the ORE scheme first converts it into a bit sequence $(b_1, b_2, \cdots, b_\omega)$. Then, the bit sequence is encoded into the ORE ciphertext $(u_1, u_2, \cdots, u_\omega)$, where $u_i, i \in [1, \omega]$ is related to b_i. Generally, the bit length of u_i is no less than 2. However, this approach has a severe security issue, i.e., it will leak the most significant differing bit or block [13], [14].

3.2 Description of SORE

Our proposed SORE scheme can enhance the security of bit-based ORE. Specifically, before encrypting a message m with the bit-based ORE, we expand m into an order preserving bit sequence by substituting the selected bits with random bit sequences. With the help of this idea, SORE can counteract against the leakage of bit-based ORE, i.e., the most significant differing bit or blocks, while inheriting the features of ORE. There are five algorithms involving SORE, and they can be described as: $\prod_{SORE}$=$\{SORE.Setup, SORE.REE, SORE.Encrypt, SORE.Compare, SORE.Decrypt\}$.

- *SORE.Setup*$(1^{\lambda}, k)$. The setup algorithm takes a security parameter λ and an expansion coefficient k as inputs and generates seven parameters used in *SORE.REE*(), *SORE.Encrypt*(), and *SORE.Decrypt()*. We list all parameters in Table 1 and will elaborate them later in this section.
- *SORE.REE*$(N_b, loc[N_b], rs[N_b], m)$. Using the first three inputs, this algorithm expands and encodes a message m into an order-preserving bit sequence, namely m_{opbs}=$SORE.REE(m)$. See details in Algorithm 1. An important property of the algorithm is that: given two messages (m_i, m_j), we have:

$$Compare(m_{iopbs}, m_{jopbs}) = Compare(m_i, m_j) \tag{2}$$

TABLE 1: *SORE.Setup* Parameters

Notation	Description	Where to use
sk	Secret key	*SORE.Encrypt()*, *SORE.Decrypt()*
N_b	Number of selected bits	*SORE.REE()*
$loc[N_b]$	A set to indicate the location of selected bits	*SORE.REE()*, *SORE.Decrypt()*
$allo[N_b]$	A set to indicate the sub-bit sequence length	*SORE.Setup()*
$rs[N_b]$	A set to indicate the random sub-bit sequence	*SORE.REE()*, *SORE.Decrypt()*
rbs	A random bit sequence used as prefix and suffix	*SORE.Setup()*
N	Target length of the order-preserving bit sequence	*SORE.Setup()*

where m_{iopbs}=*SORE.REE*(m_i), m_{jopbs} = *SORE.REE*(m_j).

- *SORE.Encrypt*(sk, m_{opbs}). The algorithm encrypts m_{opbs} using sk and outputs a ciphertext c. In SORE, we employ *ORE.Encrypt*() in bit-based ORE as *SORE.Encrypt*().
- *SORE.Compare*(c_i, c_j). The algorithm should be consistent with *SORE.Encrypt*(). If we adopt a bit-based *ORE.Encrypt*() in SORE, we should also adopt the corresponding bit-based comparison function *ORE.Compare*() as *SORE.Compare*().
- *SORE.Decrypt*$(sk, loc[N_b], rs[N_b], c)$. The decryption algorithm consists of two sub algorithms. One is *ORE.Decrypt*(), and the other is *SORE.DeREE*() (it is formally depicted in Algorithm 2). To decrypt the ciphertext c, first, *ORE.Decrypt*() is applied to recover m_{opbs} using sk. Then, *SORE.DeREE*() is used to recover m from m_{opbs}.

Obviously, *SORE.Setup()* and *SORE.REE()* are two core algorithms in SORE. To better understand why we need to initialize those parameters listed in Table 1, we first discuss *SORE.REE()* then describe *SORE.Setup()*.

SORE.REE(). We divide this algorithm into three steps:

Step-1. Add fixed *prefix* and *suffix*. The last argument in *SORE.REE*() is the message m to be expanded and encoded. To support multiple types of messages, SORE uniformly converts a message m into a bit sequence $\{0,1\}^n$ before processing it, where $n = \mathsf{len}(m)$. Afterward, *prefix*:$\{0,1\}^\ell$ and *suffix*:$\{0,1\}^\ell$ are added to the original bit sequence, and we denote the new bit sequence as m_{ps}:$\{0,1\}^{\ell+n+\ell}$.

Step-2. Expand. The expansion step refers to the selected bits in m_{ps} being replaced with random sub-bit sequences $rs[N_b]$ generated in *SORE.Setup*(). The i-th selected bit is expanded to $rs[i]$ $(i = 0, 1 \dots N_b - 1)$ random bit sequence.

Step-3. Encode. This step forms the order-preserving bit sequence m_{opbs}:$\{0,1\}^N$, which keeps the order of original messages. We propose a simple but effective strategy to maintain the order of original messages. For unselected bits, they remain unchanged. For the i-th selected bit, if it is 0, we replace it with the corresponding random sub-bit sequence $rs[i]$. Otherwise, we replace it with $rs[i]$+1. See line 9-13 in Algorithm 1 for details.

Now, we prove that the order-preserving bit sequences preserve the order of original messages, i.e., Eq. (2) holds.

Proof. Given two plaintexts m_i and m_j, suppose $m_i < m_j$. After adding *prefix* and *suffix* to each plaintext, we have $m_{ips} < m_{jps}$. In *Step-2*, the same bit positions in m_{ips} and m_{jps} are selected to expand. If the first differing bit between m_{ips} and m_{jps} is not selected, the bit can guarantee that the expanded sequences have $m_{iopbs} < m_{jopbs}$. Otherwise, we suppose the first differing bit is the i-th selected, and the bit in m_{ips} will be replaced with $rs[i]$, while it is $rs[i]$ + 1 for m_{jps}. Therefore, $m_{iopbs} < m_{jopbs}$ holds in this case. Hence, we have *Compare*(m_i, m_j)=*Compare*(m_{iopbs}, m_{jopbs}).

SORE.Setup(). In Table 1, the bit length of the secret key sk is determined by the security parameter λ, namely, $\mathsf{len}(sk) = \lambda$. Besides sk, there are six parameters, and we explain them as follows.

(1) N_b is a random number. We limit it to be greater than $n/2$ where n=$\mathsf{len}(m)$, and generally $n \geq 8$. This is because, if N_b is small, it means that few bits are selected, which lowers the security of SORE and deviates from our security goal.

(2) $loc[N_b]$ is a set with size N_b. It is generated by a piecewise sampling algorithm and imposed on *prefix* position set $\{0, 1, \dots .\ell - 1\}$, message position set $\{\ell, \ell+1, \dots \ell+n-1\}$ and *suffix* position set $\{\ell+n, \ell+n+1, \ell+n+\ell-1\}$, respectively. Accordingly, $0 \leq loc[i] < \ell+n+\ell$ where $i = \{0, 1 \dots . N_b - 1\}$. In the worst case, we hope to guarantee that at least one bit is selected in *prefix* and *suffix*.

(3) $allo[N_b]$ can be regarded as a set from an allocation algorithm that randomly assigns the total extension length (*TEL*) to each selected bit. Regarding *TEL*, we have the following equation:

$$TEL = N - (\ell + n + \ell - N_b) = \sum_{i=0}^{N_b-1} allo[i] \qquad (3)$$

The assigned value $allo[i]$ represents the length of the random sub-bit sequence, which replaces the selected bit in the expand step of *SORE.REE()*. Note that, it is required that each item in $allo[N_b]$ must be no less than 2. That is because SORE protects the index of the most significant differing bit by randomly adding the sub-bit sequences to the original message. If the length of the sub-bit sequence is 1, the security improvement of our scheme may be weakened, especially in some extreme cases, e.g., the length of all sub-bit sequences is 1.

(4) $rs[N_b]$ is a random sub-bit sequence set. First, we generate N_b random strings, and then these strings are truncated by specified length, such that, $rs[i] = \{0,1\}^{allo[i]} (i = 0, 1, 2 \dots N_b - 1)$. Before finalizing $rs[N_b]$, we design a verification algorithm to check whether $rs[i]$ is all 1-bit sequence, e.g., '111', '11111'. From the encode step of *SORE.REE*(), we know that if the selected bit is 1, the substitution sequence would be $rs[i]$+1. Hence, to avoid the overflow, $rs[N_b]$ needs to exclude all 1-bit sequence.

(5) rbs is a random bit sequence with N bits: $rbs = \{0,1\}^N$, where N will be introduced in the next part. Since ℓ is public and $\ell < N$, we can obtain *prefix* and *suffix* by truncating rbs, i.e., *prefix*=*suffix*=$\{0,1\}^\ell$.

(6) N is the target length of the order-preserving bit sequence $m_{opbs} : \{0,1\}^N$, which is produced by *SORE.REE*(). It can be calculated by the following equation:

$$N = k \times \mathsf{len}(\textit{prefix} \,||\, m \,||\, \textit{suffix}) = k \times (\ell + n + \ell) \qquad (4)$$

where ℓ=len(*prefix*)=len(*suffix*), n=$\mathsf{len}(m)$ and k is an expansion coefficient, and $k > 1$.

The proof of SORE correctness. As a secure ORE scheme, SORE must satisfy the ORE correctness definition,

Algorithm 1 *SORE.REE*()

```
Input: Number of selected bit, $N_b$; The set of location for selected bit, $loc[N_b]$; The set of sub-bit sequence, $rs[N_b]$; Message $m$; Note that prefix and suffix are fixed, len(prefix)=len(suffix)=$\ell$.
Output: Order-preserving bit sequence, $m_{opbs} : \{0,1\}^N$;
$i, index \leftarrow 0$
$m : \{0,1\}^n \leftarrow \mathsf{ConvertToBits}(m)$
$m_{ps} : \{0,1\}^{\ell+n+\ell} \leftarrow \mathsf{Add}(prefix, suffix, m)$
$\mathsf{Sort}(loc[N_b])$
while $i < (\ell + n + \ell)$ do
    $currentBit \leftarrow \mathsf{getBit}(m_{ps})$
    if $i = loc[index]$ then
        $subBitSeq \leftarrow rs[index]$
        if $currentBit = 0$ then
            $m_{opbs} \leftarrow m_{opbs} \,||\, subBitSeq$
        else
            $m_{opbs} \leftarrow m_{opbs} \,||\, (subBitSeq + 1)$
        end if
        $index \leftarrow index + 1$
    else
        $m_{opbs} \leftarrow m_{opbs} \,||\, currentBit$
    end if
    $i \leftarrow i + 1$
end while
return $m_{opbs}$
```

i.e., *SORE.Compare*(c_i, c_j)=*Compare*(m_i, m_j). The proof:

$$\begin{aligned}
&SORE.Compare(c_i, c_j) = ORE.Compare(c_i, c_j)\\
&= ORE.Compare(SORE.Encrypt(m_{iopbs}),\\
&\quad SORE.Encrypt(m_{jopbs}))\\
&= ORE.Compare(ORE.Encrypt(m_{iopbs}),\\
&\quad ORE.Encrypt(m_{jopbs})) \xrightarrow{\text{from Eq. (1)}}\\
&= Compare(m_{iopbs}, m_{jopbs}) \xrightarrow{\text{from Eq. (2)}} = Compare(m_i, m_j).
\end{aligned}$$

Example of SORE. As shown in Fig. 2, we give an example to illustrate SORE. In the example, the message is an integer 89, and *prefix*=*suffix*='0101'. Additionally, we set n=len(89)=8 and k=2. In *SORE.Setup*(), we obtain the following parameters:

$$N_b = 6, \quad \ell = \mathsf{len}(prefix) = \mathsf{len}(suffix) = 4$$
$$N = k \times (\ell + n + \ell) = 32$$

N_b is randomly generated and greater than $n/2$=4. Using the piecewise sampling algorithm, we get the selected bit set $loc[6]$=$\{0, 3, 5, 9, 11, 14\}$, which indicates the position of m_{ps}:$\{0,1\}^{16}$. Simultaneously, we have the allocation set $allo[6]$=$\{4, 2, 5, 3, 3, 5\}$, and we can check the correctness of this set using Eq. (3). In Fig. 2, there are six sub-bit sequences $rs[6] = \{$'1000', '01', '11110', '101', '011', '00111'$\}$. The length of each item is $\{4, 2, 5, 3, 3, 5\}$.

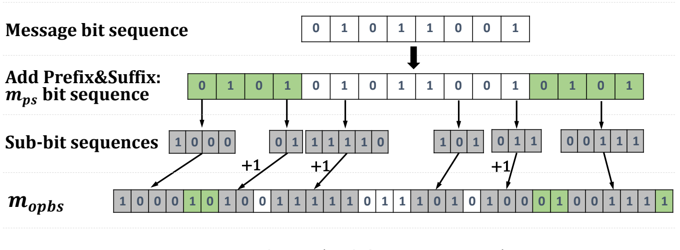

Fig. 2: A Simple SORE Example

In *SORE.REE*(), first, we add *prefix* and *suffix* ('0101') to bit sequence m, then we expand selected bits with corresponding sub-bit sequences. Finally, we encode m_{ps}:$\{0,1\}^{16}$ and form output m_{opbs}:$\{0,1\}^{32}$. In Fig. 2, grey, green and white boxes represent sub-bit sequences, *prefix*/*suffix* and the original message, respectively.

Algorithm 2 *SORE.DeREE*()

```
Input: An order-preserving bit sequence, $m_{opbs}$; The set of location for selected bit, $loc[N_b]$; The set of sub-bit sequence, $rs[N_b]$.
Output: A plaintext, $m$;
$N \leftarrow \mathsf{len}(m_{opbs})$
$i, anchor, index \leftarrow 0$
while $anchor < N$ do
    if $i = loc[index]$ then
        $n \leftarrow \mathsf{len}(rs[index])$
        $bitSeq \leftarrow \mathsf{getBits}(m_{opbs}, anchor, anchor + n)$
        if $bitSeq = rs[index]$ then
            $m \leftarrow m \,||\,$ '0'
        else
            $m \leftarrow m \,||\,$ '1'
        end if
        $index \leftarrow index + 1$
        $anchor \leftarrow anchor + n$
    else
        $m \leftarrow m \,||\, m_{opbs}[anchor]$
        $anchor \leftarrow anchor + 1$
    end if
    $i \leftarrow i + 1$
end while
$m \leftarrow \mathsf{prunePreSuffix}(m, \ell)$
return $m$
```

4 SOREL: Index Structure and Basic Operations Processing

To guarantee the security of outsourced data, we employ SORE as a cryptographic primitive in SOREL, which however incurs a formidable challenge in usability and efficiency. Hence, an efficient index is required to support basic operations (query, insert and delete) over SORE produced ciphertexts. Unfortunately, most of the existing index schemes cannot satisfy the requirements. For instance, hash tables can perform the point query with $O(1)$ time, but they cannot directly support the range query. Ordered indices, such as B+tree, can perform range query well, but many of them have time complexity with $O(\log N)$, where N is the number of records or size of the dataset, which are not sufficiently efficient. In view of the reality, we propose a novel index structure, which can perform query and insert operations over SORE ciphertexts and attain better efficiency. With our index, the worst-case time complexity of range query is $O(1) + O(\log B)$, where B is the size of the block in our index, and the time complexity of insertion is similar to that in the range query. Besides, we apply several optimization techniques to further improve the efficiency of our index. Next, we describe (i) our index structure and (ii) processing basic operations in detail.

4.1 Our index structure

In this section, we first introduce the key ideas of our index structure. Then, we present a distribution suppression scheme that can address the security and efficiency issues when using the index structure for non-uniform datasets. Finally, we describe how to build our index structure.

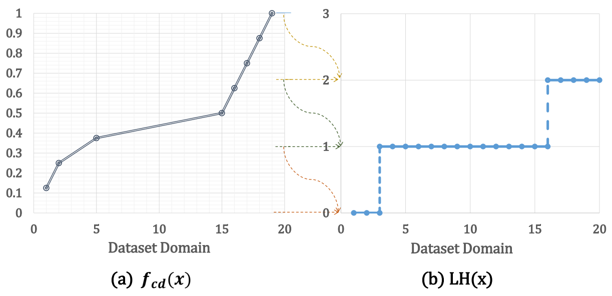

Fig. 3: Distribution suppression scheme

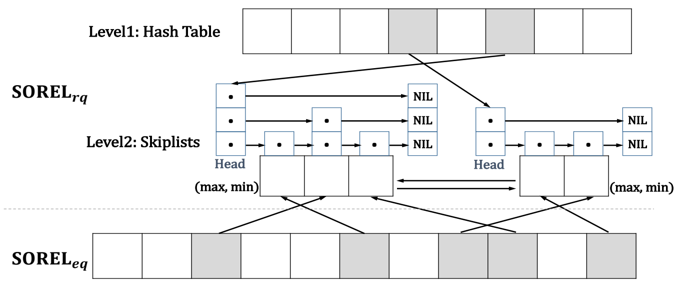

Fig. 4: Our Index Structure

4.1.1 Key ideas

The motivation behind the design of our index structure is to support fast query (including point and range query) operations in a secure manner. Our performance aim is to drive the cost of query processing close to $O(1)$. Since a hash table can perform the point query fast, we first utilize the hash table based data structure to deal with the point query. The remaining and challenging issue is how to make hash tables support the range query. To tackle it, we introduce the idea of locality preserving hashing [20] and propose a notion of **order aggregating hash function** to make it possible to use hash tables for the range query.

Order aggregating hash function. The definition of order aggregating hash function $H_{oa}(x)$ is as follows:

Definition 1. An order aggregating hash function is defined as $H_{oa}(x) = Hash(LH(x), sk)$, where $LH(x)$ is a many-to-one function that can map a set into one point while preserving the order relation of sets, $Hash(\cdot)$ is a regular hash function with the secret key sk.

Take a sorted dataset: D=$\{1, 2, 5, 15, 16, 17, 18, 19\}$ as an example. First, with the help of $LH(x)$, we can divide the dataset into several sets and map them into points. Also, $LH(x)$ should keep the order relation of these sets. A simple example is $LH(x)$= $\lfloor \frac{x}{a} \rfloor$, where x is an element of the dataset, and a is a constant, e.g., a=5. In this case, D can be divided into three sets: $\{1, 2\}$, $\{5\}$ and $\{15, 16, 17, 18, 19\}$. Then, we link these sets to a hash table using $Hash(\cdot)$, which means the key is $H_{oa}(d_i)$, where $d_i \in D$, and the value is the pointer to a set. When launching a range query, we can first calculate hash values for the query's boundary by $H_{oa}(x)$, and then identify in which set the boundary value should be located. Afterward, a special algorithm, e.g., binary search, is applied on the set to lookup the boundary value. To ensure efficiency in both insertions and range queries, we design a search algorithm based on skiplist. Since locating the range query's boundary values on a set only involves the comparison operation, and SORE ciphertexts can reveal the order relation of underlying plaintexts by a public comparison algorithm, searching on a set can be performed over a SORE encrypted dataset. Therefore, for a range query, given boundary values' hash values and SORE ciphertexts, we can answer the query using the index structure built by such an idea.

4.1.2 Distribution suppression scheme

Although the proposed $H_{oa}(x)$ can bridge the gap between the range query and hash tables, there is still a critical issue, i.e., the sets pointed by the hash table will reveal the distribution of the underlying dataset (notably, in our index structure, all items in a dataset are encrypted by SORE). Accordingly, an attacker can easily analyze the size of each set and infer the distribution of the original dataset. In addition to the security issue, there is also an efficiency issue, i.e., a non-uniformly distributed dataset tends to result in a mass of elements concentrating in one set, which may skew and slow down the execution of the range query and insertion. In such a case, the benefits of hash tables will vanish, and the efficiency will degenerate to a binary search. To alleviate these issues, we put forth a distribution suppression scheme to hide data distribution. First of all, we summarize the design goals for the scheme as follows.

- Design goals: Given a sorted dataset, (1) hide the data distribution; (2) limit the size of each subset to no more than a threshold; (3) locate a value falling in which set quickly.

A potential approach is to combine the cumulative distribution function $f_{cd}(x)$ and $LH(x)$ function in Definition 1. We observe that equalizing $f_{cd}(x)$ over the y-range can ensure the sets have the same approximate size. Therefore, integrating $f_{cd}(x)$ into $LH(x)$ can form a step function and can be formally expressed as $LH(x) = \lfloor f_{cd}(x) \cdot \frac{Count(D)}{B_n} \rfloor$, where *Count(D)* is the number of the distinct value in the given dataset, and B_n is the expected size of a set. Taking the aforementioned dataset D as an example, the function $f_{cd}(x)$ and refined $LH(x)$ are shown in Fig. 3a and Fig. 3b, respectively. In this example, if we suppose $B_n = 3$, there will be three sets, i.e., $\{1, 2, \}$,$\{5, 15, 16\}$ and $\{17, 18, 19\}$. However, it is hard and inefficient to fit the function $f_{cd}(x)$ if the data volume is large. Our distribution suppression scheme still uses the idea that $LH(x)$ is a step function, which has a mapping relation in Fig. 3b. But, in a practical way, the following expression will be adopted to calculate the value of $LH(x)$:

$$LH(x) = \sum_{i=1}^{B_c-1} \begin{cases} 1 & x - x_i \geq 0 \\ 0 & else \end{cases} \tag{5}$$

where B_c= $\lceil \frac{Count(D)}{B_n} \rceil$ is the number of sets, and x_i is the value of split point of the step function. In the above example, we can treat 5 and 17 as the split points in dataset D. When performing a range query, the data user can compute $LH(x)$ in parallel using Eq. (5). Therefore, he/she can quickly determine which set the query value falls in. Additionally, reducing the number of split points can further improve the efficiency of computing $LH(x)$, for example, using the idea of the learned index introduced in [23]. However, since we focus on the response time of the range query taking on the cloud server, these optimization

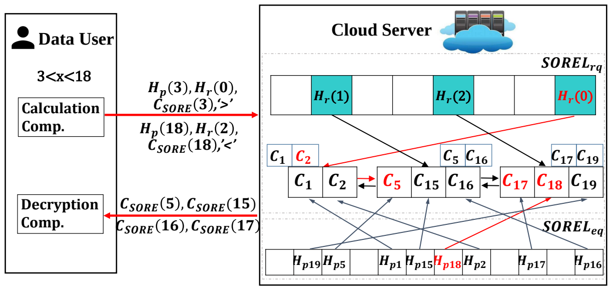
Data User
3<x<18
Calculation Comp.
Decryption Comp.
$H_p(3), H_r(0)$, $C_{SORE}(3)$,'>'
$H_p(18), H_r(2)$, $C_{SORE}(18)$,'<'
$C_{SORE}(5), C_{SORE}(15)$
$C_{SORE}(16), C_{SORE}(17)$
Cloud Server
$SOREL_{rq}$
$H_r(1)$
$H_r(2)$
$H_r(0)$
C_1 C_2
C_5 C_{16}
C_{17} C_{19}
C_1 C_2 C_5 C_{15} C_{16} C_{17} C_{18} C_{19}
$SOREL_{eq}$
H_{p19} H_{p5} H_{p1} H_{p15} H_{p18} H_{p2} H_{p17} H_{p16}

Fig. 5: Range query example

techniques are beyond the scope of this paper and will be discussed in our future work.

4.1.3 *Index structure building*

Based on the discussion in Section 4.1.1, one could employ two relatively independent components for the proposed index structure: one is for point query, denoted as $SOREL_{eq}$, and the other is a two-level data structure for range query, called $SOREL_{rq}$, as shown in Fig. 4. For ease of description, we first introduce $SOREL_{rq}$ and then discuss $SOREL_{eq}$.

$SOREL_{rq}$. It is a two-level data structure. The first level is a hash table linked to distinct skiplists that comprise the second level. Prior to constructing $SOREL_{rq}$, the data provider needs to find the split points for the given dataset D and uses them to form $LH(x)$ shown in Eq. (5). Afterward, there are three steps for building $SOREL_{rq}$ as follows.

Step-1. Preparation. Sort the dataset D, and then calculate the tuple $\langle H_{oa}(d_i), C_{sore}(d_i)\rangle$ for each item $d_i \in D$, where $H_{oa}(d_i) = Hash(LH(d_i), sk)$ and $C_{sore}(d_i) =$ *SORE.Encrypt(SORE.REE*(d_i)*)*.

Step-2. Build the first level. Initialize a hash table, and limit the size of each bucket to 1. Let the hash key be $H_{oa}(d_i)$, and the value be a pointer to the block that contains the corresponding ciphertext $C_{sore}(d_i)$.

Step-3. Build the second level. After all items in D are assigned to blocks, we can build skiplists for each block over the included $C_{sore}(d_i)$. It is noteworthy that we utilize a few optimization techniques for this level. First, maximum and minimum values are stored in each block to enhance efficiency. Then, a doubly linked list is built for these blocks.

$SOREL_{eq}$. A hash table can be directly employed here for the point query. We define $H_p(x)$=$Hash(x, sk_p)$ as the hash function for the hash table, where sk_p is the other secret key. To maximize the benefits of the hash table, we modify it and link each record in $SOREL_{eq}$ to $SOREL_{rq}$, as shown in Fig. 4. Accordingly, the cost of range query can be further reduced by searching in $SOREL_{eq}$ first (see details in Section 4.2.1).

Algorithm 3 formally outlines the entire index structure building process. Notably, in reality, since the index building is based on the encrypted data, the process can be outsourced to the cloud, and the data provider only needs to transfer the hash values and encrypted data of each item to the cloud server.

4.2 Processing basic operations

In this section, we will present the detail of how our SOREL framework deals with the basic operations, i.e., query, insert, and delete, over encrypted data. All of these operations

Algorithm 3 Building Index Structure

```
Input: A dataset, $D$=[$d_1, d_2 ... d_n$]; Two secret keys, $sk$ and $sk_p$; Size of block, $B_n$.
Output: The proposed index structure, $\Gamma$;
 1: Sort($D$)
 2: $hashTableRQ \leftarrow$ initRQHashTable()
 3: $hashTableEQ \leftarrow$ initEQHashTable()
 4: $LH(x), B_c \leftarrow$ generateLH($D, B_n$)
 5: for $i = 0 \rightarrow n-1$ do
 6:     $h_p \leftarrow$ Hash($d_i, sk_p$)
 7:     $h_{oa} \leftarrow$ Hash($LH(d_i), sk$)
 8:     $c_{sore} \leftarrow$ SORE.Encrypt(SORE.REE($d_i$))
 9:     if isExistBlock($h_{oa}, hashTableRQ$) = $True$ then
10:         $block \leftarrow$ HashTableSearch($h_{oa}, hashTableRQ$)
11:     else
12:         $block \leftarrow$ InitSkiplist()
13:         HashTableWrite($\Gamma, h_{oa}, block, hashTableRQ$)
14:     end if
15:     $address \leftarrow$ blockAddNode($\Gamma, block, c_{sore}$)
16:     setMinMax($\Gamma, block, c_{sore}$)
17:     HashTableWrite($\Gamma, h_p, address, hashTableEQ$)
18: end for
19: buildDoublyLinkedList($\Gamma, B_c, sk, hashTableRQ$)
20: return $\Gamma$
```

are executed on the proposed index structure for efficiency consideration.

4.2.1 *Query*

Although our index structure can support the point and range query simultaneously, for simplicity, we describe these two queries separately.

Point query. This operation is straightforward. The data user calculates the hash value $H_p(d_p)$ for a query value d_p, i.e., $H_p(d_p)$=$Hash(d_p, sk_p)$, where the secret key sk_p is authorized by the data provider. Then, $H_p(d_p)$ is sent to the cloud server. Upon receiving it, the cloud can lookup the search key in $SOREL_{eq}$ with $O(1)$.

Range query. In our scheme, as the dataset is sorted, and items are bidirectionally connected, the range query can be decomposed as locating upper bound and lower bound values and collecting all items between them. When performing a range query, the cloud server can first use $SOREL_{eq}$ to locate the boundary values. If the value exists in the hash table, it is quite easy for our index structure to locate the corresponding ciphertext. Otherwise, $SOREL_{rq}$ will be used to find the predecessor (<) or successor (>) for the boundary value. In $SOREL_{rq}$, the first level (a hash table) can be used to quickly locate the skiplist block that the boundary value should fall in, and then searching is conducted on the corresponding skiplist to find the predecessor or successor.

To illustrate range query processing, let us suppose that the data user launches a query $3 < x < 18$ over the encrypted dataset D, as shown in Fig. 5 (to save space, the skiplists in level 2 are not shown in the figure). First, the data user needs to calculate two hash values, $H_p(x)$ and $H_{oa}(x)$, and a ciphertext $C_{sore}(x)$ for each boundary value, where $H_p(x) = Hash(x, sk_p)$, $H_{oa}(x) = Hash(LH(x), sk)$, and $C_{sore}(x) =$ *SORE.Encrypt(SORE.REE*(x)*)*. In this case, x=3 or 18. Since $LH(3) = 0$ and $LH(18) = 2$ (see details in Section 4.1.2), we denote $H_{oa}(3) = Hash(0, sk)$ and $H_{oa}(18) = Hash(2, sk)$ as $H_r(0)$ and $H_r(2)$, respectively. Upon receiving these values, the cloud first checks whether $H_p(x)$ is in $SOREL_{eq}$ or not. For $H_p(18)$, it is easy to locate the position of $C_{sore}(18)$ by $SOREL_{eq}$ and find that

$C_{sore}(17)$ is predecessor. For $H_p(3)$, since there does not exist the corresponding hash value in *SOREL*$_{eq}$, the cloud server needs to perform a search on *SOREL*$_{rq}$. By using $H_r(0)$, the cloud can quickly locate in which block $C_{sore}(3)$ should be. The time complexity is only $O(1)$. Then, $C_{sore}(3)$ will be used to lookup the successor $C_{sore}(5)$ on skiplist with $O(\log B)$, where B is the size of the skiplist block. In fact, with the help of the maximum value $C_{sore}(2)$ in the first block and the minimum value $C_{sore}(5)$ in the second block, the cloud can find the successor for $C_{sore}(3)$ quickly. Finally, the cloud server responds to the range query $3 < x < 18$ by returning all items between $C_{sore}(5)$ and $C_{sore}(17)$ (including them).

After obtaining the desired ciphertexts, the data users can use the sk, $loc[N_b]$, and $rs[N_b]$, which are authorized by the data provider, to recover the corresponding plaintexts.

4.2.2 Insert

Similar to the range query launched by the data user, when inserting, the data provider also needs to calculate three values $H_p(d_i)$, $H_{oa}(d_i)$ and $C_{sore}(d_i)$ for the inserting data d_i, and forward them to the cloud server. Then, on the cloud side, $H_{oa}(d_i)$ will be used to insert $C_{sore}(d_i)$ in *SOREL*$_{rq}$, and $H_p(d_i)$ is for insertion in *SOREL*$_{eq}$.

For inserting a ciphertext, the cloud server needs to lookup the predecessor of the insertion value first, which is analogous to the query operation. Then, some updating operations have to be conducted, such as updating the doubly linked list and the tuple of $\langle max, min\rangle$. However, if the new items are continuously inserted, there may be excessive items in one block. To tackle this case, we set a threshold for the block size, denoted as B_{th}, and define an index refactoring mechanism.

Index refactoring. Suppose there are B_c blocks in our index structure, the dataset has the domain A_n. Therefore, B_{th} = $\lceil \frac{A_n}{B_c} \rceil$. Since $A_n >$ *Count*(D), B_{th} will be greater than B_n. The data provider will hold a table to record the real-time size for each block. In the table, there are B_c items, and the key is $LH(x)$ that can uniquely identify a block. Before forwarding a new item to the cloud server, the data provider will check if the block size to which the inserting value belongs is larger than B_{th}. If so, the data provider will launch the index building as discussed in Section 4.1.3. Since the data provider only needs to check if the size of the block to be inserted is greater than B_{th} or not, the time complexity of this operation is $O(1)$. Actually, in our target scenario, which is read frequently and has fewer insertions, the index refactoring does not happen often.

Notably, in our scheme, the duplicate items are ignored, since they can be organized under the same node of the skiplist. When performing a query, all of the values in one node will be returned if hit.

4.2.3 Delete

For deleting, the data provider only needs to compute and send one hash value $H_p(d_e)$ to the cloud server, where d_e is the data to be deleted. If the value exists, the cloud can find it quickly both in *SOREL*$_{eq}$ and *SOREL*$_{rq}$ due to the link between them. The remaining tasks are to update the *SOREL*$_{rq}$, which are similar to insertion. For the data provider, the other operation is to update the block size to which the deletion item belongs.

5 Analysis

In this section, we conduct a formal analysis of SOREL in terms of security and efficiency.

5.1 Security analysis

As the confidentiality of the outsourced data in SOREL is mainly guaranteed by SORE, in this section, we will first explore the leakage profile of SORE, and then analyze the security of the proposed index structure.

Security of SORE. For efficiency and practicality, we choose the bit-based ORE as the basic cryptographic primitive. However, these encryption schemes leak more information, namely, the most significant differing bit, except for order relations. Therefore, the security of these encryption schemes is much weaker than the ideal security (IND-OCPA) of ORE, which requires that the encrypted data reveal nothing except for order relations. To enhance the security, we propose SORE to hide the most significant differing bit, and our aim is to bring the existing bit-based ORE schemes closer to IND-OCPA. The key idea is that the cloud server still knows the first differing bit through a comparison function, but it is not the real first differing bit in the original plaintext sequence. Suppose $m \in \{0,1\}^n$ is a plaintext with n bits. By applying *SORE.REE*$()$, it turns into $m_{opbs} \in \{0,1\}^{k(\ell+n+\ell)}$, where k is an expansion coefficient and ℓ is the length of fixed *prefix* and *suffix*. The cloud server can learn the first bit that is different in two encoded sequences $m_{opbs} \in \{0,1\}^{k(\ell+n+\ell)}$ using the comparison function *SORE.Compare*$()$. However, the first differing bit in the original plaintext $m \in \{0,1\}^n$ is still unknown. To learn the real first differing bit in plaintext $m \in \{0,1\}^n$, the cloud server needs to identify which bit of $m \in \{0,1\}^n$ is mapped to the leaked bit in $m_{opbs} \in \{0,1\}^{k(\ell+n+\ell)}$. Next, we will prove that it is hard for the cloud to infer the real first differing bit from the SORE ciphertexts.

To facilitate security analysis, we introduce a leakage function to depict the security level. For IND-OCPA, we have the following leakage function:

$$\mathcal{L}(c_i, c_j) = \{Pr(m_i < m_j) = 1 \mid c_i < c_j\}$$

It means the order relation for the plaintexts m_i and m_j can be leaked through comparing the corresponding ciphertexts. Whilst, for the bit-based ORE, the leakage function can be defined as follows [13]:

$$\begin{aligned}\mathcal{L}_{ore}(c_i, c_j) = \{&Pr(\mathit{diff}(m_i, m_j)) = 1,\\ &Pr(m_i \le m_j) = 1 \mid c_i \le c_j\}\end{aligned}$$

where *diff*(m_i, m_j) gives the index of the first bit that differs between two plaintexts: m_i and m_j. Here, we define the lower order as little endian and encode the index from 1. For instance, comparing 0101 and 0111, the index of the first differing bit is 2.

Now, we explore the leakage function for SORE. First, we denote p as the index of the most significant differing bit for SORE ciphertexts and say x is the index of the most significant differing bit for the corresponding plaintexts.

Then, using *SORE.Compare*() function, the cloud can get p for any given two ciphertexts. Afterward, it is easy for the cloud to guess the upper bound of x, i.e., $x < p - \ell$, in which ℓ is public. Further, the lower bound of x can be revealed by the number of distinct ciphertexts whose values lie inside the range of these two compared ciphertexts, and we denote it as n_c. As these ciphertexts will occupy at least $\log_2 n_c$ bits, the cloud server can guess $x > \log_2 n_c$. Consequently, the real index should be guessed in a range: $\log_2 n_c < x < p - \ell$.

To prove the correctness of above inequality, we assume that there are two integers $\{m_i, m_j\}$ and $m_i > m_j$. It is clear that there are $m_i - m_j - 1$ distinct integers between them. From the definition of n_c, we have $m_i - m_j - 1 \geq n_c$. If let p' indicate the index of the first differing bit between m_i and m_j, we have $2^{p'} > m_i - m_j - 1 \geq n_c$. Recalling Section 3.2, our SORE scheme adds the suffix bit sequence $\{0,1\}^\ell$ to the message bit sequence and expands it into a new bit sequence with $k(\ell + n + \ell)$ length. Therefore, we have $p - \ell > p'$. Hence, $p - \ell > \log_2 n_c$. Accordingly, the leakage profile for SORE is as follow:

$$\mathcal{L}_{sore}(c_i, c_j) = \{Pr(\mathit{diff}(m_i, m_j)) = \frac{1}{(p-\ell) - \log_2 n_c},$$
$$Pr(m_i \leq m_j) = 1 \mid c_i \leq c_j\}$$

However, since we guarantee that at least one bit is selected in *suffix*, and the bit length of the substitution sequence is larger than 2, we have $(p - \ell) - \log_2 n_c \geq 2$. Besides, there is no advantage for the cloud to determine whether the guessed index is correct. Therefore, the cloud server cannot determine the most significant differing bit of the underlying plaintexts from SORE ciphertexts, and our SORE scheme indeed improves the security of bit-based ORE by hiding the most significant differing bit.

For OPE and ORE based encryption schemes, they preserve the order relations in their ciphertexts to maximize efficiency when performing range queries. Since the order relations may incur inference attacks [24], we suggest applying SORE in a large domain with low density due to security considerations. Besides, to ensure security, our SORE scheme should be applied to one column, i.e., each column of a database should have its own secret key, and SORE encrypts the database by column.

Security of Index structure. In our threat model, we consider the cloud server is curious about the outsourced data and operation values. At first, our SORE scheme guarantees the cloud server cannot obtain the underlying plaintexts from the encrypted outsourced data without sk, $loc[N_b]$, and $rs[N_b]$ (note that we apply SORE in a large domain with low density). Second, when performing range queries and insertions on the index structure, the cloud server can receive the corresponding operation values, i.e., $H_p(x)$, $H_{oa}(x)$, and $C_{sore}(x)$. However, none of these values leaks the original plaintext. Regarding point queries and deletions, only $H_p(x)$ is sent to the cloud server. Therefore, it is secure on the operation values when performing queries, insertions, and deletions. Another security concern on the index structure is that it may potentially disclose the data distribution of the original dataset. We design a distribution suppression scheme to hide this information. Since each block in level 2 has the approximately same number of data items, the cloud server learns nothing about the original data distribution. Hence, the proposed index structure is secure for outsourced data and operation values.

Note that, in our scheme, the data user should offer identity-related information to demonstrate his/her legitimacy, which can be achieved by the authentication schemes in [25], [26].

5.2 Efficiency analysis

A significant goal of our system is to make the proposed schemes as efficient as possible while enhancing the security of the outsourced data. In this section, we analyze the efficiency of SORE and operations over our index structure.

SORE. The efficiency of SORE depends on the security enhancement algorithm *SORE.REE*() and the encryption algorithm *SORE.Encrypt*(). First, *SORE.REE*() algorithm only has a time complexity $O(\ell + n + \ell)$, where n and ℓ are the bit lengths of plaintext sequence and *prefix*/*suffix*, respectively, and often neither n nor ℓ is large. Second, *SORE.Encrypt*() algorithm has a time complexity $O(k(\ell + n + \ell))$, while the underlying bit-based ORE has a baseline with $O(n)$.

Index structure. We now analyze the efficiency of query and insert operations over the designed index structure.

• *Query*. Since the time complexity of the point query in our scheme is straightforward, i.e., $O(1)$, here we will only discuss the efficiency of the range query. Without loss of generality, when performing a range query, our scheme has a time complexity of $O(1) + O(\log B)$, where B is the size of the skiplist block, as shown in Fig. 4. For a fixed dataset, the smaller the B is, the closer the efficiency of our scheme is to that of hash table with $O(1)$. Conversely, as the value of B gets larger, the efficiency deteriorates as the complexity approaches towards that of the binary tree with $O(\log N)$. Since we can make $B \ll N$, (concrete examples are in Section 6.3), our scheme is always much better than the binary tree. Furthermore, we enhance range query efficiency by adopting optimization techniques: (i) we store maximum and minimum values at each block. It can accelerate the range query performance over original skiplist, since the values outside the minimum and maximum boundaries can perform the range query with $O(1)$ rather than $O(\log B)$; (ii) we link $SOREL_{eq}$ to $SOREL_{rq}$ and search on $SOREL_{eq}$ first, which can improve the efficiency at the structural level compared with tree-based indexes.

• *Insertion*. On the cloud side, for $SOREL_{eq}$, the insertion has the same time complexity as hash tables. For $SOREL_{rq}$, the insertion complexity is $O(1) + O(\log B)$, which is similar to the range query. Meanwhile, stored maximum and minimum values can speed up the operation. Therefore, our scheme is competitive for the insertion on the cloud side. On the data provider side, if the current block size is not greater than the threshold B_{th}, the time complexity is $O(1)$. Otherwise, we need to rebuild the index structure. However, for a stable dataset with a given domain, the rebuilding is infrequent in the target scenario.

6 Evaluation

In this section, we evaluate the proposed SORE and data operations (range query and insert) on our index structure. Both of them are evaluated in terms of experimentally analyzing the impact factors and comparing with prior works.

6.1 Experimental setup

Implementation. Our implementation is entirely written in C and is evaluated on Ubuntu 16.04 OS with 16 GB memory and 3.4 GHz Intel(R) Core(TM) i7-3770 processors. To implement SORE, we choose the most practical ORE construction scheme [13], henceforth referred to **PORE**, as the security enhancement object, which uses SHA-256 as the pseudo-random function and AES-128 as the encryption algorithm. For the index structure, we adopt an extendible hashing technique [27] in level 1 of $SOREL_{rq}$ and limit the bucket to 1, which allows us to automatically expand the size of the hash table when conflicts occur. In $SOREL_{eq}$, the cuckoo hashing technique [28] is used to attain $O(1)$ search time complexity. As the main memory capacity is growing, in-memory databases [29] are becoming popular. Hence, we evaluate all the schemes with in-memory versions. Note that all of the schemes are evaluated on integer datasets. It is reasonable since we can easily convert non-integer data, such as floats, into integers.

Datasets. As mentioned in Section 5.2, the time complexity of SORE is related to the maximum bit length of plaintexts. To evaluate it, we synthesized a dataset with six attributes, and each of them contains 4 million integers that are randomly generated in different domains from 2^8 to 2^{24}.

To evaluate the performance of the range query and insertion, we experiment with five datasets, including two real-world datasets and three synthetic datasets. The first real-world dataset is from the "total pay and benefits" column of California public employees salaries [30], hereafter called *Salary* dataset. We convert the dataset into positive integers with domain A_n={0,770000}. The second real-world dataset is from the "location id" column of the Gowalla geo-social network [31], called *LocId* dataset, which has a domain A_n={0,6000000} with 1,280,969 distinct items. For the synthetic datasets, we adopt the benchmark *Sanzu* [32] to generate one normal-distribution and two uniform-distribution datasets. We list the detailed information of these datasets in Table 2, where *Nor24*, *Uni24*, and *Uni48* are shorthand for these three synthetic datasets, respectively, and the column $Density = \frac{Dataset\ size}{Domain\ size}$.

TABLE 2: Datasets for testing rang query and insertion

Name	Datasize	Domain	Density
Salary	86412	[0,770000]	11%
LocId	1280969	[0,6000000]	21%
Nor24	2410000	[0,6000000]	40.2%
Uni24	2420000	[0,6000000]	40.3%
Uni48	4760000	[0,6000000]	79.5%

6.2 Performance of SORE

In our experiments, we use execution time as the cost metric of the encryption process of SORE. We first measure the average execution time for each component in SORE and then compare it with other related cryptographic primitives.

6.2.1 Performance of the components in SORE

Recall from Section 5.2, for encryption, the execution time of SORE is mainly due to the algorithms of $SORE.REE()$ and $SORE.Encrypt()$. Since we adopt PORE as the underlying encryption scheme, here the $SORE.Encrypt()$=$PORE.Encrypt()$. The evaluation results for various bit lengths of plaintext are shown in Fig. 6. It is evident that SORE is sensitive to the bit length of plaintexts. And the longer the length, the more time it will take. In these experiments, we set the expansion coefficient k as 1.5 and 2, respectively, and the length of *prefix* and *suffix* ℓ as 2. The cost of PORE is a linear function for bit length n and is the major time-consuming component for SORE. Note that PORE encrypts the plaintext directly, while *SORE.Encrypt()* encrypts the expanded plaintext. From Fig. 6, we can see that most of the execution time of SORE is contributed by the underlying ORE scheme, while the core algorithm $SORE.REE()$ only accounts for a small proportion. This trend is more obvious as the bit length of plaintext gets longer, which demonstrates that we improved the security of ORE at a little cost.

6.2.2 Comparison with other encryption primitives

To date, there are two major cryptographic primitives, OPE and ORE, which reveal the order property of plaintext. To be fair, we select the most efficient scheme (with a similar security level) in each primitive as competitors. For OPE, we choose a practical OPE scheme, denoted as **POPE**, to compare against. This scheme has been integrated into CryptDB [10] for supporting the range query over encrypted data. For ORE, we choose an efficient bit-based ORE scheme [14], denoted as **NORE**, which can also protect the first bit from being leaked. Fig. 7 illustrates the average execution time of these schemes at different plaintext lengths. We evaluate SORE with k=1.5 and k=2.0, and the results show that even at k=2.0, SORE is at least 8× faster than NORE and at least 25× faster than POPE when the plaintext length is greater than 8 bits. It is worth noting that we do not expand the ciphertext of POPE to a larger domain in the experiment. Otherwise this encryption will take more time.

6.3 Performance of index structure

In this section, we evaluate the performance of our proposed index structure. Since we adopt the hash table technique to resolve the point query, it is unfair to measure it. Hence, for the query processing, we only evaluate and compare the range query operation. Similar to Section 6.2, we first experimentally analyze how the relevant factors affect the efficiency of building, insertion and range query, then we compare our scheme with other efficient schemes that can support these operations over encrypted data.

6.3.1 Performance of building, insertion and range query

For a given dataset, generally, the index structure is built intensively, while it is scattered in time for insertion and range query. Consequently, we will evaluate the total execution time of index building and the average execution time of insertion and range query.

Index Building. For most index structures, intuitively, the dataset size can affect the performance of index building. For our scheme, in addition to this factor, the other factor is the block size, B_n, which is determined by the data provider. Fig. 8 shows the execution time to build our index structure on these five datasets. In each dataset, we vary B_n from 16 to 256, and the results show that the larger B_n is, the more time it will take. It is reasonable since we build the index structure by inserting items (see details in

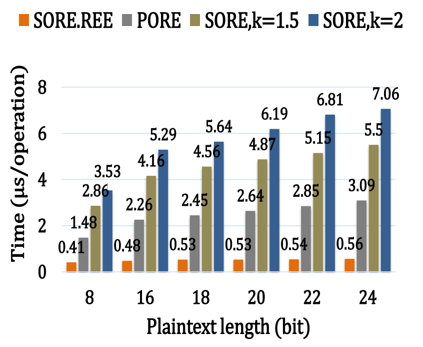

Fig. 6: Evaluating SORE components

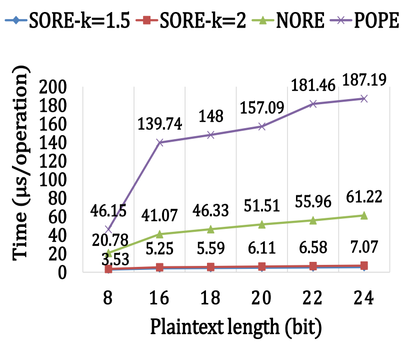

Fig. 7: Encryption primitives

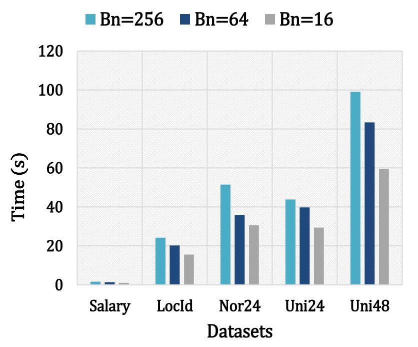

Fig. 8: Total time of index building

Algorithm 3). Another interesting point is that the index building overhead for *Nor24* and *Uni24* is close. In fact, for the datasets with different data distributions, we apply the distribution suppression scheme to compress the datasets into equal-size blocks. Therefore, the index building time is not impacted by the dataset distribution.

Insertion. As mentioned in Section 5.2 , the time complexity of insertion is $O(1)+O(\log B)$, where B is the size of the block. In our experiments: $B = B_n$. Besides, we remove the repeated data. Hence, the average insertion time is only related to the block size. Fig. 9 shows the evaluation results on different datasets, which verifies our analysis.

Range query. Similar to insertion, the range query on our index structure has a basic time complexity $O(1) + O(\log B)$. It means that B_n will also affect the performance of the range query. Additionally, the density of the dataset will have a significant impact on the average execution time of the range query. Specifically, the denser the dataset, the lower the average execution time. That is because, on a fixed domain, we can locate the predecessor or successor by $SOREL_{eq}$ rather than $SOREL_{rq}$ with a higher probability when the data volume is large, which can dramatically reduce the range query complexity from $O(1) + O(\log B)$ to $O(1)$. Fig. 10 shows the average query time with these five datasets. Obviously, the average query time with *Uni48* is the lowest, which benefits from its high density of around 80%. For *Nor24* dataset, it keeps the same level, even lower, performance compared to that of *Uni24* dataset, which suggests that our *distribution suppression scheme* can prevent range query performance from deteriorating on non-uniform distribution. Also, although the datasets *Nor24* and *Uni24* have a close density value, the average query time of *Uni24* is higher than that of *Nor24* slightly. In fact, the larger density can only guarantee that there is a higher probability, not absolute, to hit. Therefore, the performance of the range query is also related to the specific dataset and query values.

6.3.2 Comparison with other index structures

Since OPE is not efficient, the combination of OPE and traditional indexes is not competitive in terms of performance. Roche et al. [9] proposed an approach to support range query and insert operations using OPE, which is fairly fast with insertion operation. However, it sacrifices the performance of query operations and has a large number of interactive communications. ORE is more practical than OPE. It can be expected that building index structures on ORE can have better performance than on OPE. Therefore, the ORE-based protocol from [22] is adopted as a competitor, which is built with a B+tree working on top of a practical ORE scheme. To be fair, in implementing the comparison scheme, we use the most practical ORE [13] for the competitor, and the STX B+tree [33], a highly optimized and mature implementation, is adopted.

TABLE 3: SSE-based range query schemes
n: dataset size, r: result size, m: domain size, R: query range size

Schemes	Interaction	Search time	Storage
PBtree [4]	No	$\Omega(\log n \log R + r)$	$O(n \log n \log m)$
Quadratic	No	$O(r)$	$O(nm^2)$
Constant-BRC	No	$O(R + r)$	$O(n)$
Logarithmic-BRC	No	$O(\log R + r)$	$O(n \log m)$
Logarithmic-SRC	No	$O(n)$	$O(n \log m)$
Logarithmic-SRC-i	Yes	$O(R + r)$	$O(n \log m)$

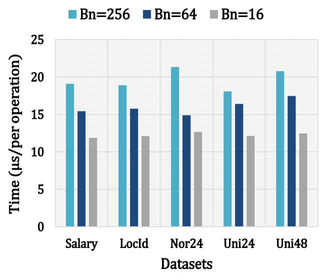

Fig. 9: Average time of insertion

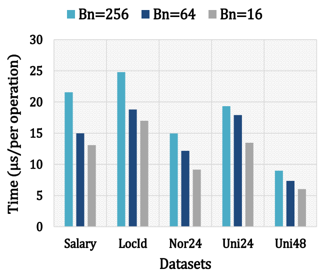

Fig. 10: Average time of range query

A different approach is to perform the range query and insertion using SSE [3]. Such schemes allow direct use of hash tables for the range query and achieve efficient performance on both operations. In Table 3, we compare the properties of these schemes and select one of the efficient SSE-based scheme logarithmic-BRC, abbreviated as *LogBRC*, as the comparison scheme. In Table 3, we can see that the fastest scheme is *Quadratic* that has the search time complexity with $O(r)$ (note that $n \gg R$ and r in the big data scenario). However, this scheme consumes a lot of storage overhead, which makes it impractical. Therefore, we choose *LogBRC* as the comparison scheme for the overall performance considerations.

Note that, both the B+tree based scheme and our scheme only locate the predecessor and successor when performing a range query. Thus, for *LogBRC*, we just locate the query results without collecting them in our implementation.

Since our index structure is built by inserting data items, the average insertion time can also reflect the performance of the index structure building. Hence, we will compare our scheme with other schemes only in the average execution time of insertion and range query. Besides, since the *Uni48* dataset has the largest data size (4.76 million) among these

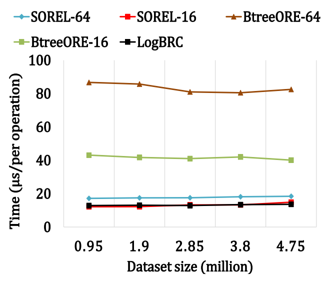

Fig. 11: Insert comparison

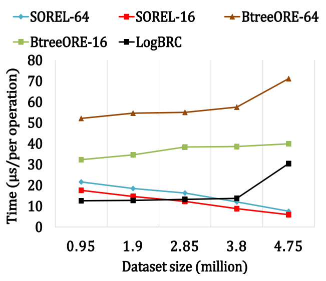

Fig. 12: Range query comparison

five datasets, it is selected to evaluate each scheme varying with dataset size, and the data volume ranged from 0.95 million to 4.75 million that are randomly sampled.

Comparison on insertion. Fig. 11 compares our scheme with other schemes in the average execution time of insertion. We provide two versions for the B+tree-based scheme. One has 64 slots denoted as *BtreeORE-64*, the other has 16 slots represented as *BtreeORE-16*. Correspondingly, our scheme also offers the same settings, choosing B_n=64 and B_n=16 as comparison schemes. In Fig. 11, both our schemes have better performance than BtreeORE based schemes. Despite that the insertion performance of *SOREL-64* is not as good as *LogBRC*, it is competitive when B_n=16. It is because the insertion time of our index is affected by the block size, and the smaller block size will make our index more efficient in insertions.

Comparison on range query. Fig. 12 shows the average execution time of the range query for each scheme. As the data volume grows, the average query time of our schemes decreases, while it increases for others. This shows the superiority of our scheme in the given domain scenarios. When the data volume is large, our schemes outperform the corresponding BtreeORE based schemes by 9× and 7×, respectively. From Fig. 12, we can see that *SOREL-16* performs better than *LogBRC* when the dataset size is greater than around 2.5 million, while it is about 3.5 million for *SOREL-64*. In our scheme, when the data volume grows large, the search key would hit the existing values with a higher probability, which approaches the range query complexity to $O(1)$. For *LogBRC*, more items will be retrieved from the inverted table when the data volume is larger. Thus, our index has better query performance with large data volumes. When the dataset size is 4.75 million, our scheme improves the average execution time of *LogBRC* by up to around 3×. Actually, by making full use of the hash table techniques, the *LogBRC* scheme is already efficient enough in performing the insertion and range query. However, such schemes are limited to enumerable attributes. By contrast, our scheme can support continuous attributes while guaranteeing better performance in range query operations. Therefore, our scheme has better scalability in data volume and data type.

7 Related Work

Encryption primitives. To utilize outsourced data in a secure way, researchers have presented several cryptographic primitives to support the secure query and insertion. One of the most prominent schemes is property preserving encryption (PPE) [5], which allows the ciphertexts to reveal a particular property of their underlying plaintexts. For instance, deterministic encryption (DET) leaks the equality property, while order preserving encryption (OPE) [6] is proposed to keep the order property of plaintexts. In [11], an efficient OPE scheme was proposed, and the authors proved that the ideal security level of OPE is IND-OCPA. To improve security, some optimized OPE schemes [34], [35] were proposed based on the OPE definition, that is, if $p_1 < p_2$, then $Enc(p_1) < Enc(p_2)$. In [34], before applying OPE, a modular shift is applied in plaintext to enhance its security. In [35], Kerschbaum proposed a scheme in which the basic idea is to randomize the ciphertexts to hide the frequency of the queries. However, OPE schemes are only able to support numeric value, and Boldyreva et al. [11] showed that IND-OCPA is unachievable for all practical OPE schemes. An alternative order persevering scheme called ORE (Order-Revealing Encryption) [12] was proposed by Dan Boneh et al. This approach can support any data type. Although the encryption scheme appears to resemble semantic security, a special public comparison function is designed to reveal order by comparing two ciphertexts. Unfortunately, in [13], [14], the constructed ORE schemes have extra leakage for the most significant differing bit or block. In addition, [15], [16] were proposed with the leakage of "equality pattern" of the most significant differing bits.

In addition to PPE, several cryptographic primitives have been proposed for secure outsourced data, which have a higher security level and do not expose any plaintext properties. Among them, a popular scheme is Homomorphic Encryption (HE) [1], which allows basic operations on encrypted data. Although HE can guarantee both security and utility, its significant performance overhead makes it hard to be a practical scheme. Another popular cryptographic primitive is the garbled circuit [36]. It can be used to compare data without leaking their information. However, the garbled circuit suffers from similar performance issues and is inefficient with updates. A very different primitive is Oblivious RAM (ORAM), which was proposed by [37]. Similarly, this approach also has performance problems.

Query over encrypted data. The issue of supporting query processing over encrypted data has attracted a lot of attention due to the important applications involving outsourced data in the cloud. Most of the ciphertext-based range query protocols were built on existing cryptographic primitives that we introduced earlier. Following the idea of OPE, MOPE [8] and POPE [9] were designed to support range query while improving security. Both protocols are claimed to be able to achieve IND-OCPA. However, excessive communications among the entities make them less practical. Based on the garbled circuit, the authors in [2] presented an index, ArxRange, for range and order-by-limit queries, which can avoid interactions and achieve higher security compared with OPE-based protocols. As mentioned earlier, updating circuits may seriously affect the efficiency of ArxRange. A number of ORAM-based range query schemes [38], [39] have been proposed. However, due to the limitations of the underlying primitive, they are less competitive in terms of efficiency. Following the notion of Searchable Symmetric Encryption (SSE), Demertzis et al. [3] proposed several schemes with realistic security and efficiency trade-offs. The main idea behind these schemes is to convert a range query into the multi-keyword search and then apply SSE schemes. However, these range query protocols only support enumerable fields. In [21], a scheme to support range query over encrypted data for multiple dimensions was proposed. In the approach, multi-attributes in a dataset are divided into small cubes, and queries are executed on them. However, this scheme incurs false positives, and updates are performed inefficiently.

8 CONCLUSIONS

In this paper, we first propose a secure ORE-based encryption scheme, SORE, which can enhance the security of existing bit-based ORE schemes. Then, a novel index structure is presented to support query and insertion operations efficiently. Moreover, a distribution suppression scheme is introduced to guarantee the security and efficiency of the proposed index structure. By combining these schemes, we present a practical framework, SOREL, which can be applied to secure outsourced data. Detailed experimental evaluations are conducted, and the results show that our schemes outperform existing approaches. As future work, we expect to enhance the security of SORE when applying it to the small domain or high-density datasets and further improve the efficiency of our ORE-based range query scheme.

REFERENCES

[1] I. Damgard, M. Geisler, and M. Kroigard, “Homomorphic encryption and secure comparison,” *IJACT*, vol. 1, no. 1, pp. 22–31, 2008.

[2] R. Poddar, T. Boelter, and R. A. Popa, “Arx: an encrypted database using semantically secure encryption,” *VLDB*, pp. 1664–1678, 2019.

[3] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. Garofalakis, “Practical private range search revisited,” in *SIGMOD*, 2016, pp. 185–198.

[4] R. Li, A. X. Liu, A. L. Wang, and B. Bruhadeshwar, “Fast range query processing with strong privacy protection for cloud computing,” *VLDB*, pp. 1953–1964, 2014.

[5] O. Pandey and Y. Rouselakis, “Property preserving symmetric encryption,” in *EUROCRYPT*. Springer, 2012, pp. 375–391.

[6] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, “Order preserving encryption for numeric data,” in *SIGMOD*, 2004, pp. 563–574.

[7] P. Antonopoulos, A. Arasu, K. D. Singh, K. Eguro, N. Gupta, R. Jain, R. Kaushik, H. Kodavalla, D. Kossmann, N. Ogg *et al.*, “Azure sql database always encrypted,” in *SIGMOD*, 2020, pp. 1511–1525.

[8] R. A. Popa, F. H. Li, and N. Zeldovich, “An ideal-security protocol for order-preserving encoding,” in *S&P*, 2013, pp. 463–477.

[9] D. S. Roche, D. Apon, S. G. Choi, and A. Yerukhimovich, “Pope: Partial order preserving encoding,” in *SIGSAC*, 2016, pp. 1131–1142.

[10] R. A. Popa, C. M. Redfield, N. Zeldovich, and H. Balakrishnan, “CryptDB: protecting confidentiality with encrypted query processing,” in *SOSP*, 2011, pp. 85–100.

[11] A. Boldyreva, N. Chenette, Y. Lee, and A. O’neill, “Order-preserving symmetric encryption,” in *EUROCRYPT*, 2009, pp. 224–241.

[12] D. Boneh, K. Lewi, M. Raykova, A. Sahai, M. Zhandry, and J. Zimmerman, “Semantically secure order-revealing encryption: Multi-input functional encryption without obfuscation,” in *EUROCRYPT*, 2015, pp. 563–594.

[13] N. Chenette, K. Lewi, S. A. Weis, and D. J. Wu, “Practical order-revealing encryption with limited leakage,” in *FSE*, 2016, pp. 474–493.

[14] K. Lewi and D. J. Wu, “Order-revealing encryption: New constructions, applications, and lower bounds,” in *SIGSAC*, 2016, pp. 1167–1178.

[15] D. Cash, F.-H. Liu, A. O’Neill, and C. Zhang, “Reducing the leakage in practical order-revealing encryption.” *IACR Cryptology ePrint Archive*, vol. 2016, p. 661, 2016.

[16] D. Cash, F.-H. Liu, A. O’Neill, M. Zhandry, and C. Zhang, “Parameter-hiding order revealing encryption,” in *ASIACRYPT*, 2018, pp. 181–210.

[17] D. Comer, “Ubiquitous B-tree,” *CSUR*, pp. 121–137, 1979.

[18] Y. Mao, E. Kohler, and R. T. Morris, “Cache craftiness for fast multicore key-value storage,” in *EuroSys*, 2012, pp. 183–196.

[19] V. Leis, A. Kemper, and T. Neumann, “The adaptive radix tree: Artful indexing for main-memory databases,” in *ICDE*, 2013, pp. 38–49.

[20] P. Indyk, R. Motwani, P. Raghavan, and S. Vempala, “Locality-preserving hashing in multidimensional spaces,” in *STOC*, 1997, pp. 618–625.

[21] S. Wu, Q. Li, G. Li, D. Yuan, X. Yuan, and C. Wang, “ServeDB: secure, verifiable, and efficient range queries on outsourced database,” in *ICDE*, 2019, pp. 626–637.

[22] D. Bogatov, G. Kollios, and L. Reyzin, “A comparative evaluation of order-revealing encryption schemes and secure range-query protocols,” *VLDB*, vol. 12, no. 8, pp. 933–947, 2019.

[23] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska, “Fiting-tree: A data-aware index structure,” in *SIGMOD*, 2019, pp. 1189–1206.

[24] M. Naveed, S. Kamara, and C. V. Wright, “Inference attacks on property-preserving encrypted databases,” in *SIGSAC*, 2015, pp. 644–655.

[25] S. Qiu, D. Wang, G. Xu, and S. Kumari, “Practical and provably secure three-factor authentication protocol based on extended chaotic-maps for mobile lightweight devices,” *IEEE Transactions on Dependable and Secure Computing*, 2020.

[26] D. Wang and P. Wang, “Two birds with one stone: Two-factor authentication with security beyond conventional bound,” *IEEE transactions on dependable and secure computing*, pp. 708–722, 2016.

[27] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, “Extendible hashing—a fast access method for dynamic files,” *TODS*, vol. 4, no. 3, pp. 315–344, 1979.

[28] R. Pagh and F. F. Rodler, “Cuckoo hashing,” in *ESA*, 2001, pp. 121–133.

[29] J. Lee, Y. S. Kwon, F. Färber, M. Muehle, C. Lee, C. Bensberg, J. Y. Lee, A. H. Lee, and W. Lehner, “SAP HANA distributed in-memory database system: Transaction, session, and metadata management,” in *ICDE*. IEEE, 2013, pp. 1165–1173.

[30] “2019 salaries for state of california,” *https://transparentcalifornia.com/salaries/2019/state-of-california/*.

[31] J. Leskovec, “Gowalla geo-social network dataset,” *http://snap.stanford.edu/data/loc-gowalla.html*, 2011.

[32] A. Watson, D. S. V. Babu, and S. Ray, “Sanzu: A data science benchmark,” in *Big Data*. IEEE, 2017, pp. 263–272.

[33] T. Bingmann, “STX B+ Tree C++ Template Classes,” *https://panthema.net/2007/stx-btree/*, 2013.

[34] A. Boldyreva, N. Chenette, and A. O’Neill, “Order-preserving encryption revisited: Improved security analysis and alternative solutions,” in *CRYPTO*, 2011, pp. 578–595.

[35] F. Kerschbaum, “Frequency-hiding order-preserving encryption,” in *SIGSAC*, 2015, pp. 656–667.

[36] A. C. Yao, “Protocols for secure computations,” in *SFCS*, 1982, pp. 160–164.

[37] O. Goldreich, “Towards a theory of software protection and simulation by oblivious rams,” in *STOC*, 1987, pp. 182–194.

[38] E. Stefanov and E. Shi, “Oblivistore: High performance oblivious cloud storage,” in *S&P*. IEEE, 2013, pp. 253–267.

[39] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, “Path ORAM: an extremely simple oblivious RAM protocol,” in *SIGSAC*, 2013, pp. 299–310.

Songnian Zhang received his M.S. degree from Xidian University, China, in 2016 and he is currently pursuing his Ph.D. degree in the Faculty of Computer Science, University of New Brunswick, Canada. His research interest includes cloud computing security, big data query and query privacy.

Suprio Ray is an Associate Professor with the Faculty of Computer Science, University of New Brunswick, Fredericton, Canada. He received a Ph.D. degree from the Department of Computer Science, University of Toronto, Canada. His research interests include big data and database management systems, run-time systems for scalable data science, provenance and privacy issues in big data and data management for the Internet of Things. E-mail: sray@unb.ca

Rongxing Lu (S'09-M'11-SM'15-F'21) is an associate professor at the Faculty of Computer Science (FCS), University of New Brunswick (UNB), Canada. Before that, he worked as an assistant professor at the School of Electrical and Electronic Engineering, Nanyang Technological University (NTU), Singapore from April 2013 to August 2016. Rongxing Lu worked as a Postdoctoral Fellow at the University of Waterloo from May 2012 to April 2013. He was awarded the most prestigious "Governor General's Gold Medal", when he received his PhD degree from the Department of Electrical & Computer Engineering, University of Waterloo, Canada, in 2012; and won the 8th IEEE Communications Society (ComSoc) Asia Pacific (AP) Outstanding Young Researcher Award, in 2013. Dr. Lu is an IEEE Fellow. His research interests include applied cryptography, privacy enhancing technologies, and IoT-Big Data security and privacy. Currently, Dr. Lu serves as the Vice-Chair (Conferences) of IEEE ComSoc CIS-TC (Communications and Information Security Technical Committee).