2017 IEEE 33rd International Conference on Data Engineering

Adaptively Secure Conjunctive Query Processing over Encrypted Data for Cloud Computing

Rui Li Alex X. Liu
College of Computer Science and Network Security, Dongguan University of Technology
Dongguan 523808, Guangdong, China
Email: ruili@dgut.edu.cn, alexliuxy@yahoo.com

***Abstract*—This paper concerns the fundamental problem of processing conjunctive queries that contain both keyword conditions and range conditions on public clouds in a privacy preserving manner. No prior Searchable Symmetric Encryption (SSE) based privacy-preserving conjunctive query processing scheme satisfies the three requirements of adaptive security, efficient query processing, and scalable index size. In this paper, we propose the first privacy preserving conjunctive query processing scheme that satisfies the above requirements. To achieve adaptive security, we propose an Indistinguishable Bloom Filter (IBF) data structure for indexing. To achieve efficient query processing and structure indistinguishability, we propose a highly balanced binary tree data structure called Indistinguishable Binary Tree (IBtree). To optimize searching efficiency, we propose a traversal width minimization algorithm and a traversal depth minimization algorithm. To achieve scalable and compact index size, we propose an IBtree space compression algorithm to remove redundant information in IBFs. We formally prove that our scheme is adaptive secure using a random oracle model. The key contribution of this paper is on achieving conjunctive query processing with both strong privacy guarantee and practical efficiency in terms of both speed and space. We implemented our scheme in C++, evaluated and compared its performance with KRB [24] for keyword queries and PBtree [32] for range queries on two real-world data sets. Experimental results show that our scheme is fast and scalable (in milliseconds).**

I. Introduction

A. Motivation and Problem Statement

Both enterprises and end users have been increasingly outsourcing their data and computing services to public clouds such as Amazon EC2 and S3, Microsoft Azure, and Google App Engine for lower cost, higher reliability, better performance, and faster deployment. However, privacy has become the key concern as data owners may not fully trust public clouds. First, clouds may have corrupt employees. For example, in 2010, a Google engineer broke into the Gmail and Google Voice accounts of several children. Second, clouds may be hacked and customers may not be informed. Third, some cloud facilities may be operated in some foreign countries where privacy regulations are difficult to enforce.

This paper focuses on the popular cloud computing paradigm where a data owner stores data on a cloud and multiple data users query the data. Figure 1 shows these three parties: a data owner, a cloud, and multiple data users. Among the three parties, the data owner and data users are trusted, but the cloud is not fully trusted.

In this paper, we consider the fundamental problem of processing conjunctive queries that contain both keyword conditions and range conditions on public clouds in a privacy preserving manner. The query conditions in the *where* clauses of SQLs are often conjunctive conditions of keywords and ranges. A disjunctive query can be easily converted into multiple conjunctive queries. For example, in the SQL query `select * from patient where NAME="John" and age` ≤ 30, the `where` clause contains a *keyword* condition `NAME="John"` and a *less-than* condition `age < 30`, which can be converted into range condition `age` $\in [0, 30]$. In particular, we consider Searchable Symmetric Encryption (SSE) schemes as symmetric encryption based privacy preserving schemes are significantly more efficient than asymmetric ones. In SSE, the data owner builds a *secure index* `I` for a data set D, and encrypts each data item $d_i \in D$ into $(d_i)_K$ using a secret key K that is shared between the data owner and data users. Then, the data owner outsources the secure index `I` along with the set of encrypted data $\{(d_1)_K, (d_2)_K, \cdots, (d_n)_K\}$ to the cloud. Given a conjunctive query q, the data user generates a *trapdoor* t_q for q, and sends t_q to the cloud. Base on t_q and the secure index `I`, the cloud can determine which encrypted data items satisfy q without knowing the content of the data and query. Yet, in this process, the cloud should not be able to infer privacy information about the data items and queries such as data content, query content, and the statistical properties of attribute values. The index `I` should leak no information about the data items in D. Note that a data item d_i could be a record in a rational database table or a text document in a document set.

Fig. 1: Cloud computing model

B. Threat Model

We assume that the cloud is *semi-honest* (also called *honest-but-curious*), which was proposed by Canetti *et al.* in [9] and has been widely adopted by prior privacy preserving keyword and range query work [4], [6], [7], [10]–[12], [16], [18], [25], [31], [32], [40]. A cloud is semi-honest means that it follows required communication protocols and execute required algorithms correctly, but it may attempt to obtain information about data items and the content of user queries. The data owner and data users are trusted.

C. Security Model

In this paper, we adopt the *adaptive IND-CKA* security model proposed in [16], which is by far the strongest security model for SSE based keyword query schemes. Here IND-CKA means that the index is *indistinguishable (IND)* under *chosen keyword attack (CKA)*. Non-adaptive and adaptive IND-CKA

2375-026X/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDE.2017.122

models differ in the view offered to the adversary. The IND-CKA model is based on two facts. First, the outputs of a truly random function contains no meaningful information. Second, the outputs of a pseudo-random function are not distinguishable from the outcomes of a truly random function by any probabilistic polynomial-time adversary. An index I constructed by a pseudo-random function is $\mathcal{L}$-secure under the IND-CKA model if and only if there exists a simulator $\mathcal{S}$ that is constructed from the information leaked by leakage function $\mathcal{L}$ using a truly random function and can simulate I with the queries chosen by a polynomial-time adversary $\mathcal{A}$ so that to $\mathcal{A}$, the query results from I are not distinguishable from the outputs of $\mathcal{S}$ (*i.e.*, $\mathcal{A}$ can correctly guess whether the query result is obtained from I or $\mathcal{S}$ with a negligible probability over $\frac{1}{2}$). The leakage function $\mathcal{L}$ usually includes the information such as input queries (search pattern), output query results (access pattern), the index size, and the data size.

In the non-adaptive IND-CKA model, the adversary is not allowed to interact with the secure index prior to being challenged. The adversary $\mathcal{A}$ chooses a data set D and sends to a challenger $\mathcal{C}$. The challenger $\mathcal{C}$ builds a secure index I for D and then encrypts D. Now, the adversary chooses a set of queries Q and sends them to the challenger $\mathcal{C}$. The challenger $\mathcal{C}$ generates trapdoor t_q for each query $q \in Q$, processes queries using t_q and I, and sends back the query results to the adversary $\mathcal{A}$. The index I is secure in this non-adaptive IND-CKA model if and only if (1) there exists a simulator $\mathcal{S}$ that can simulate I by returning query results for the queries in Q and (2) $\mathcal{A}$ cannot distinguish the results returned by I from the outputs returned by $\mathcal{S}$.

In the adaptive IND-CKA model, the adversary $\mathcal{A}$ adaptively chooses each future query based on the previously chosen queries along with their trapdoors and query results. The index construction I is similar to the previous model. The adversary begins the query process as follows. First, $\mathcal{A}$ chooses a query q_1 and sends to the challenger $\mathcal{C}$. Second, $\mathcal{C}$ computes t_{q_1} for q_1, processes the query with t_{q_1} and I, and returns the query results R_1 back to $\mathcal{A}$. Third, based on I, t_{q_1}, and R_1, $\mathcal{A}$ chooses query q_2 and repeats the above query process. This process is repeated a polynomial number of times. The index I is secure in this adaptive IND-CKA model if and only if (1) there exists a simulator $\mathcal{S}$ that can simulate I by returning query results for the queries chosen by $\mathcal{A}$ and (2) $\mathcal{A}$ cannot distinguish the results returned by I from the outputs returned by $\mathcal{S}$.

The key differences between non-adaptive IND-CKA model and adaptive IND-CKA model are three folds. First, the requirements for simulators are different. The simulator in the non-adaptive IND-CKA model only needs to simulate an index for a set of *known* queries. The simulator in the adaptive IND-CKA model needs to simulate an index for both *known* and *unknown* queries. Second, on what queries the results returned by I are undistinguishable from the outputs returned by $\mathcal{S}$ are different. For the non-adaptive IND-CKA model, only for the set of known queries, the results returned by I are guaranteed to be undistinguishable from the outputs returned by $\mathcal{S}$. For the adaptive IND-CKA model, for a polynomial number of both previously known and previously unknown queries, the results returned by I are guaranteed to be undistinguishable from the outputs returned by $\mathcal{S}$. Third, the level of security is different. A non-adaptive secure scheme only provides security if the search queries are independent of the secure index I and previous search results. An adaptive secure scheme guarantees security even the search queries are not independent of the secure index I and of previous search results [25]. A query scheme is secure under the adaptive IND-CKA model, then it is also secure under the non-adaptive IND-CKA model, but not vice versa.

D. Limitation of Prior Art

A practical privacy preserving conjunctive query scheme that supports keyword conditions, range conditions, and their conjunctions should satisfy the three requirements of *adaptive security under the IND-CKA model*, *efficient query processing*, and *scalable index size with respect to the number of data items*. However, no prior scheme satisfies all these requirements. The schemes in [5], [35] achieve adaptive security and scalable index sizes, but are extremely inefficient because they use Yao's Garbled Circuits [44] to evaluate query conditions and require multiple rounds of communication to process a query. The schemes in [11], [32] achieve efficient query processing and scalable index sizes, but are not adaptively secure; the Constant schemes proposed in [17] cannot achieve adaptive security under IND-CKA model; the scheme proposed in [27] supports incremental, query-triggered adaptive indexing over encrypted numeric data. None of the schemes in [17], [27], [32] supports conjunctive queries. The scheme in [11] cannot process range queries. The schemes in [10], [13], [24], [30], [33] and the SSE-2 scheme in [16] are adaptively secure, but they all only support keyword queries; furthermore, the schemes in [13], [24], [30], [33] have an unscalable index size of $O(mn)$, where m is the total number of all possible keywords that can be queried, which can be extremely large, and n is the total number of all documents. The CryptDB system in [37] makes use of property-preserving encryption schemes such as deterministic (DTE) and order preserving encryption (OPE) to support keyword and range queries. However, property-preserving encryption schemes have been proved to be fundamentally insecure by Naveed *et al.* in CCS 2015 [34].

E. Proposed Approach

In this paper, we propose the first SSE based conjunctive query scheme that supports both keyword conditions and range conditions and satisfies the three requirements of adaptive security, efficient query processing, and scalable index sizes. To achieve adaptive security, we propose a data structure called *Indistinguishable Bloom Filter (IBF)* for storing index elements. Each element in an IBF has two cells: one is used for storing index information and the other is for masking, which help us to achieve node indistinguishability and enable the simulator of an IBF to simulate future unknown queries in a random oracle model. The key difference between IBFs and Bloom filters is that IBFs have the ability to simulate future unknown queries while Bloom filters cannot. To achieve *efficient query processing* and *structure indistinguishability*, we propose a highly balanced binary tree structure called *Indistinguishable Binary Tree (IBtree)* whose structure solely depends on the number of index elements regardless of their values. To optimize searching efficiency, we propose a traversal width minimization algorithm and a traversal depth minimization algorithm to minimize the number of nodes need to be traversed in an IBtree for processing queries. To achieve scalable index sizes, we propose an IBtree space compression algorithm to remove redundant information in the IBFs. To

support range queries, we transform range query processing into keyword query processing by transforming range membership testing (*i.e.* testing whether a number belongs to a range), which involves less-than and bigger-than comparison, into set membership testing (*i.e.* testing whether a keyword is in a set), which only involves equality comparison. Furthermore, we formally prove that our scheme achieves adaptive security.

F. Novelty and Advantages over Prior Art

Our scheme is the first SSE based conjunctive query scheme that achieves adaptive security, efficient query processing, and scalable index sizes. The key technical novelty of this paper is five-fold. First, our IBF data structure is novel. It helps us to achieve node indistinguishability and adaptive security for the whole scheme. Second, our IBtree data structure is novel. It helps us to achieve structure indistinguishability and make sublinear query processing time possible. Third, our trapdoor generation method is novel. It overcomes the range expansion problem. Fourth, our multidimensional partition algorithm is novel. It improves searching efficiency significantly, especially in the case of large number of dimensions. Fifth, our IBtree space compression algorithm is novel. It reduces index size by orders of magnitude.

II. Related Work

Non-SSE based Privacy Preserving Query Schemes: Dan Boneh *et al.* proposed a predicate encryption named Hidden Vector Encryption (HVE) to support conjunctive, subset, and range query processing [8]. Elaine Shi *et al.* proposed a scheme named MRQED using identity based encryption to handle multidimensional range queries [39]. Bharath K. Samanthula *et al.* proposed a privacy-preserving query processing scheme based on homomorphic encryption and garbled circuit techniques that supports complex queries over encrypted data [38]. However, these three schemes have linear query processing time with respect to the total number of data records. Boyang Wang *et al.* proposed a multidimensional range query processing scheme achieving sublinear query processing time, based on HVE and R-trees [41]; however, it reveals query conditions to the cloud. Furthermore, the schemes proposed in [8], [38], [39], [41] are asymmetric cryptography based approaches, which have a high computing complexity. For example, the scheme in [38] requires several seconds to test whether a data item satisfies a query condition. The scheme in [42] adopts the ASPE approach [43] to encrypt query ranges; however, the security of ASPE is not secure against chosen plain text attack [45]. Oblivious RAMs proposed by Goldreich and Ostrovsky can be used to support complex query processing with adaptive security; however, this scheme requires multiple rounds of interaction for each read and write, which makes it extremely inefficient [19]. Arasu *et al.* described the design of Cipherbase system and presented the design of Cipherbase secure hardware and its implementation using FPGAs [2], which is different from our secure model as we do not consider to equip special hardwares in cloud servers.

SSE based Privacy Preserving Keyword Query Schemes: Besides the keyword query schemes mentioned in Section I-D, there are other SSE based privacy preserving keyword query schemes such as the ones in [12], [18], [40] and the one in [16]. In [26], the authors proposed a scheme for privacy preserving string matching. But all of them only achieve non-adaptive security. Seny Kamara *et al.* proposed a scheme to process SQL queries over encrypted data for a relational database [14]. But the scheme in [14] does not supports privacy preserving conjunctive queries that contain range conditions.

SSE based Privacy Preserving Range Query Schemes: All prior SSE based privacy preserving range query schemes, no matter one-dimensional or multidimensional, cannot achieve adaptive security. Bucketing based range query schemes [21], [22] and order preserving encryption/hash function based range query schemes [1], [6], [29], [31], [36] cannot achieve provably security, no matter under the non-adaptive or the adaptive security model.

III. Basic IBtree Algorithms

A. Index Element Encoding

There are two types of index elements for indexing data: non-numeric and numeric. Given a non-numeric index element, we encode it into a keyword by concatenating its corresponding attribute name. For example, we encode people's name ``John'' into keyword ``NAME:John'', where "NAME" is the attribute of "John". Given a numeric index element, we first adopt the prefix membership verification scheme to compute its prefix family. The prefix membership verification scheme helps us to convert the testing of whether a numeric data item falls into a range to the testing of whether two sets have common elements. Given a w-bit number x whose binary representation is $b_1 b_2 \cdots b_w$, its prefix family denoted as $F(x)$ is defined as the set of $w+1$ prefixes $\{b_1 b_2 \cdots b_w, b_1 b_2 \cdots b_{w-1}*, \cdots, b_1 * \cdots *, **...*\}$, where the i-th prefix is $b_1 b_2 \cdots b_{w-i+1} * \cdots *$. For example, the prefix family of 4-bit number 4 is $F(4)$=$F(0100)$=$\{0100, 010*, 01**, 0***, ****\}$. Each prefix denotes a range. For example, the prefix 01** denotes the range $[0100, 0111]$. Given a range $[a, b]$, we convert it into the minimum set of prefixes, denoted by $S([a, b])$, such that the union of the ranges denoted by all prefixes in $S([a, b])$ is equal to $[a, b]$. For example, $S([1, 7])$=$\{0001, 001*, 01**\}$. For any number x and range $[a, b]$, $x \in [a, b]$ if and only if there exists prefix $p \in S([a, b])$ so that $x \in R(p)$ holds, where $R(p)$ is the range denoted by p. Furthermore, for any number x and prefix p, $x \in R(p)$ if and only if $p \in F(x)$. Thus, for any number x and range $[a, b]$, $x \in [a, b]$ if and only if $F(x) \cap S([a, b]) \neq \emptyset$. From the above examples, we see that $4 \in [1, 7]$ and $F(4) \cap S([0, 8])$=$\{01**\}$. After the prefix family computation, we encode each prefix into a keyword by concatenating its corresponding attribute name. For the above example, assuming the attribute of "4" is "AGE", we encode each prefix in $F(4)$ into keywords "AGE:0100", "AGE:010*","AGE:01**", "AGE:0***", and "AGE:****".

B. IBF Construction

Bloom filters (BFs) are space-efficient probabilistic data structures for fast set membership verification. The one-wayness property of Bloom filters has been exploited for preserving data privacy in prior work [18], [32]. However, neither of these two schemes achieves adaptive security. The reason stems from the difficulty of creating Bloom filters that can simulate in advance, an index for the adversary that will be consistent with future unknown queries. In this paper, we propose a new data structure called *Indistinguishable Bloom filters (IBF)*, which enables the simulator of an IBF to simulate future unknown queries, to achieve adaptive security.

Definition 3.1 (Indistinguishable Bloom Filter): An indistinguishable bloom filter (IBF) is an array B of m twins,

k pseudo-random hash functions $h_1, h_2, \cdots, h_k$, and a hash function H. Each twin consists of two cells where each cell stores one bit and the two bits remain the opposite. For each twin, the hash function H determines which cell is chosen in a random fashion. For every twin, the chosen cell is initialized to 0 and the unchosen cell is set to 1. Given one keyword w, we hash it to k twins $B[h_1(w)], B[h_2(w)], \cdots, B[h_k(w)]$, and for each of these k twins, we set its chosen cell to 1 and set its unchosen cell to 0.

An IBF with m twins always has m 1s and the chosen cell in a twin is determined randomly. Thus, a polynomial-time adversary $\mathcal{A}$ can correctly guess which cell is chosen from a twin with a negligible probability over $\frac{1}{2}$. Figure 2 shows an example IBF, where grey cells are chosen cells.

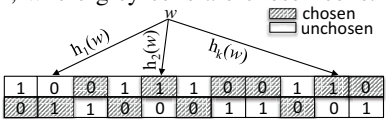


Fig. 2: An example IBF

Given a set of keywords, our IBF construction algorithm outputs the IBF B that probabilistically represents the keyword set. Let W=$\{w_1, w_2, \cdots, w_g\}$ be the given set of keywords. We assume that the data owner and the data users share $k+1$ secret keys, K_1, K_2, $\cdots$, K_k, K_{k+1}. We construct k pseudo-random hash functions $h_1, h_2, \cdots, h_k$ using the keyed-hash message authentication code HMAC, where $h_i(.)$=$\texttt{HMAC}_{K_i}(.)\%m$ for $1 \leq i \leq k$). We construct a hash function H using the pseudo-random hash function SHA1 as $H(.)$=$\texttt{SHA1}(.)\%2$. We consturct another pseudo-random hash function h_{k+1} using the keyed-hash message authentication code HMAC where $h_{k+1}(.)$=$\texttt{HMAC}_{K_{k+1}}(.)$. For each keyword w_i, we hash it to k twins $B[h_1(w_i)], B[h_2(w_i)], \cdots, B[h_k(w_i)]$. For each of these k twins $B[h_j(w_i)]$ $(1 \leq j \leq k)$, we first use the pseudo-random hash function H to determine the chosen cell location $H(h_{k+1}(h_j(w_i)) \oplus r_B)$ where r_B is a random number associated with IBF B; then, we assign its value to be 1 and the other cell's value to 0 (*i.e.*, $B[h_j(w_i)][H(h_{k+1}(h_j(w_i)) \oplus r_B)]$:=1 and $B[h_j(w_i)][1 - H(h_{k+1}(h_j(w_i)) \oplus r_B)]$:=0). For an IBF B, *the value of its i-th twin* is the value of the chosen cell.

The IBF constructed from a keyword set W preserves W's privacy for two reasons. First, through the use of secure one-way hash functions, our construction algorithm achieves one-wayness. That is, given a keyword set W, we can easily compute the IBF, but given an IBF, it is computationally infeasible to compute W. Second, through the use of the hash function $H(.)$, our construction algorithm achieves equivocation for each element in an IBF. For each twin in an IBF, the cell that we choose for storing the BF bit is totally random. Note that we associate a random number with an IBF to eliminate the correlation among different IBFs.

C. IBtree Construction

Given n data items $d_1, d_2, \cdots, d_n$, where each d_i has a keyword set $W(d_i)$, we organize the keyword sets into our *IBtree* data structure. Figure 3 shows an IBTree for the data set D=$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$.

Definition 3.2 (IBtree): An IBTree constructed from n data items is a highly balanced full binary tree with n terminal nodes and $n-1$ nonterminal nodes. All n terminal nodes form a linked list from left to right. Each terminal and nonterminal node is an IBF. Each terminal node is an IBF constructed from the keyword set of one data item, and each nonterminal node is an IBF constructed from the union of all keyword sets represented by its left and right children. For any nonterminal node, its left child has the same number of terminal nodes as its right child or its left child has one more terminal node than its right child. All IBFs in this tree have the same size.

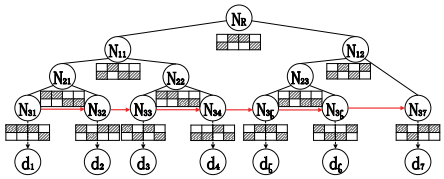


Fig. 3: IBtree Example

The IBtree data structure gives us structure indistinguishability because the structure of an IBtree solely depends on the number of data items. The key property of IBtrees is stated in Theorem 3.1, which is straightforward to prove according to the IBtree definition.

Theorem 3.1 (IBtree Structure Indistinguishability): For any two sets of data items D_1 and D_2, their IBtrees have exactly the same structure if and only if $|D_1| = |D_2|$.

Given a set of data items $D = \{d_1, d_2, \cdots, d_n\}$, we first encrypt each data item $d_i \in D$ by calling $\mathcal{E}.Enc(d_i, K_d)$, where $\mathcal{E}$ is an CPA-security encryption scheme such as AES and k_d is a secret key shared by the data owner and data users. Then we construct an IBtree in two stages. In the first stage, we build the tree structure in a top-down fashion. First, we compute the root node, which is labeled with the complete set D. Second, we partition D into two subsets D_l and D_r so that $0 \leq |D_l| - |D_r| \leq 1$. Third, we construct two child nodes for the root, where the left child is labeled with D_l and the right child is labeled with D_r. We recursively apply the above three steps to the left and right children, respectively, until each node contains only one data item. And all these nodes that contain only one data item form the leaf nodes in the IBtree. At last, we link all leaf nodes into a list.

In the second stage, we construct an IBF for each node in the IBtree in a bottom-up fashion. First, for each terminal node v labeled with a data item d, we construct an IBF from $W(d)$ using our IBF construction algorithm. Second, for each nonterminal node v whose left child and right child have constructed their IBFs, we construct v's IBF from the two IBFs of its two children using the logical OR operation. Let B_l and B_r denote the two constructed IBFs of v's left and right child, respectively. We now construct the IBF of v, denoted B_v, as follows: for each $1 \leq i \leq m$, the value of B_v's i-th twin is the logical OR of B_l's i-th twin and B_r's i-th twin. That is, $B_v[i][H(h_{k+1}(i) \oplus r_{B_v})]$:=$B_l[i][H(h_{k+1}(i) \oplus r_{B_l})]$ $\vee B_r[i][H(h_{k+1}(i) \oplus r_{B_r})]$. This process repeats until we construct the IBFs for all nonterminal nodes. Note that same $k+1$ secret keys are used for constructing all IBFs in an IBtree.

D. Trapdoor Computation

We first present our trapdoor computation algorithm for multidimensional keyword queries. Single keyword queries are special cases of multidimensional keyword queries. Given a u-dimensional keyword query $q = e_1 \wedge e_2 \cdots \wedge e_u$, the data user first encodes each index element $e_i (1 \leq i \leq u)$ into a keyword w_{e_i} by concatenating its attribute. Second, the data user computes k locations $h_1(w_{e_i})$, $h_2(w_{e_i})$, $\cdots$, $h_k(w_{e_i})$ for w_{e_i}. Third, for each location $h_j(w_{e_i})$ $(1 \leq j \leq k)$, the data user

computes hash $h_{k+1}(h_j(w_{e_i}))$. The sub-trapdoor for e_i is a k-pair of the locations and hashes: $\{(h_1(w_{e_i}), h_{k+1}(h_1(w_{e_i}))), \cdots, (h_k(w_{e_i}), h_{k+1}(h_k(w_{e_i})))\}$. Then, the $(u \times k)$-pair of locations and hashes are formed the trapdoor t_q for q.

Now we present our trapdoor computation algorithm for multidimensional range queries. One-dimensional range queries are special cases of multidimensional range queries too. Given a query q=$[a_1, b_1][a_2, b_2] \cdots [a_u, b_u]$, we first consider the case where each range $[a_i, b_i]$ $(1 \leq i \leq u)$ is a prefix range (*i.e.*, can be represented by one prefix). For such a multidimensional prefix range q, we compute its trapdoor t_q as follows. First, for each $1 \leq i \leq u$, we encode range $[a_i, b_i]$ to a keyword $A_i|[a_i, b_i]$, where A_i is the attribute of dimension i. Second, for each keyword $A_i|[a_i, b_i]$, we use the $k+1$ hash functions in IBtree construction algorithm to compute k-pair of locations and hashes. Then, for a u dimensional prefix range, we compute $u \times k$ ordered pairs as the trapdoor t_q for q.

We now consider the case where there exist $1 \leq i \leq u$ so that $[a_i, b_i]$ is not a prefix range. A straight-forward solution is to convert query q to a set of multidimensional prefix ranges whose union is q. For each multidimensional prefix range, we can compute its trapdoor as above. However, this solution suffers from the range expansion problem explained below. Let w be the number of bits for representing each a_i and b_i. In the worst case, the minimum number of prefixes to represent an arbitrary $[a_i, b_i]$ is $2w-2$ [20]. Thus, for a u-dimensional range $[a_1, b_1][a_2, b_2] \cdots [a_u, b_u]$, in the worst case, it requires a minimum of $(2w-2)^u = O(w^u)$ prefixes. To address the prefix expansion problem, we observe that there is always a minimum multidimensional prefix range, denoted by q_s, that contains the multidimensional non-prefix range $[a_1, b_1][a_2, b_2] \cdots [a_u, b_u]$. Let $q_c = q_s - [a_1, b_1][a_2, b_2] \cdots [a_u, b_u]$. We observe that q_c can be represent by $O(wu)$ multidimensional prefix ranges. Now we compute q_s and q_c. For each arbitrary range $[a_i, b_i]$, let $[\hat{a}_i, \hat{b}_i]$ be the minimum prefix range that contains $[a_i, b_i]$. The minimum multidimensional prefix range that contains $[a_1, b_1][a_2, b_2] \cdots [a_u, b_u]$ is therefore $[\hat{a}_1, \hat{b}_1][\hat{a}_2, \hat{b}_2] \cdots [\hat{a}_u, \hat{b}_u]$. For each $1 \leq i \leq u$, we use $\overline{[a_i, b_i]}$ denote $[\hat{a}_i, \hat{b}_i] - [a_i, b_i]$. Thus, $q_c = \overline{[a_1, b_1]}[\hat{a}_2, \hat{b}_2] \cdots [\hat{a}_u, \hat{b}_u] \cup [\hat{a}_1, \hat{b}_1]\overline{[a_2, b_2]}[\hat{a}_3, \hat{b}_3] \cdots [\hat{a}_u, \hat{b}_u] \cup \cdots \cup [\hat{a}_1, \hat{b}_1][\hat{a}_2, \hat{b}_2] \cdots [\hat{a}_{u-1}, \hat{b}_{u-1}]\overline{[a_u, b_u]}$. This is because for any data item $d = (d^1, d^2, \cdots, d^u)$, if there exist i so that $d^i \notin [a_i, b_i]$, then $(d^1, d^2, \cdots, d^u) \notin [a_1, b_1][a_2, b_2] \cdots [a_u, b_u]$, regardless of the values of $d^1, d^2, \cdots, d^{i-1}, d^{i+1}, \cdots, d^u$. As $\overline{[a_i, b_i]}$ can be represented by at most 2 ranges, in the worst case, the minimum number of prefixes that can represent $\overline{[a_i, b_i]}$ is $4w-4$. Thus, in the worst case, the minimum number of prefix ranges that can represent q_c is $(4w-4)u = O(wu)$. By representing query q as $q_s - q_c$, we address the range expansion problem by converting the multiplicative effect to additive effect. Our idea of processing query $q = q_s - q_c$ is to first find all data items that satisfy q_s and then prune the data items that satisfy query q_c. Thus, we call q_s the *searching component* of q and q_c the *checking component* of q.

The trapdoor t_q consists of both t_{q_s} and t_{q_c}. For q_s, which is a multidimensional prefix range, we use the above trapdoor computation method to compute $u \times k$ ordered pairs as t_{q_s}. For q_c, we only need to compute $u \times k$ ordered pairs for each $\overline{[a_i, b_i]}$ because if a data item d satisfies q_s, then we only need to check whether there exists $1 \leq i \leq u$ such that $d^i \in \overline{[a_i, b_i]}$. For each $1 \leq i \leq u$, we first convert $\overline{[a_i, b_i]}$ to a minimum set of prefix ranges, and then for each of these prefix ranges, we convert it to a keyword, and use the $k+1$ hash functions to compute k ordered pairs as t_{q_c}.

To compute a trapdoor for a conjunctive query that contains both keywords and ranges, the data user first adopts the above two trapdoor computation algorithms to compute sub-trapdoors for keywords and ranges, respectively, and then combines them together to form the trapdoor for the conjunctive query.

Discussion: A limitation of known SSE construction is that the same trapdoors are always generated for the same queries. This means that searches leak statistical information about data user search patterns. In comparison, our trapdoor computation algorithm can be amended to partially overcome this limitation. When computing t_{q_s} for query q, the data user randomly chooses $k'(k' < k)$ hash functions from $\{h_1, h_2, \cdots, h_k\}$ to compute k' location-hash pairs and pads the trapdoor to k location-hash pairs by adding k-k' location-hash pairs that do not result false negatives. We observe that for each prefix p in q_s, any prefix p' that contains p (*i.e.*, $R(p) \subset R(p')$) can be used for computing paddings, where $R(p)$ is the range denoted by p. For each $\overline{[a_i, b_i]}$ in q_c, the data user randomly inserts some location-hash pairs generated from the prefixes that contains $[\hat{a}_j, \hat{b}_j]$, where $1 \leq j \leq u$. Thus, different trapdoors can be generated for the same query. This trapdoor computation algorithm may increase false positives in query results. However, false positive is helpful for thwarting inference attacks via *access pattern*, which is described in [23]. We leave the study as our future work that combines the mitigation scheme in [23] and our amended trapdoor computation algorithm to partially hide *search patterns* and *access patterns*. Our IBtree can be combined with the mitigation scheme in [23] to prevent the inference attack in [23].

E. Query Processing

The query processing algorithm is as follows. After receiving the trapdoor t_q from a data user for query $q = q_s - q_c$, the cloud first processes query q_s using t_{q_s}. This querying process starts from the root of the IBtree. Let B_p denote the IBF of the IBtree root, $t_{qs}[i]$ denote the i-th ordered pair in t_{q_s}, $t_{q_s}[i].f$ and $t_{q_s}[i].s$ denote the first and second hash of the ordered pair $t_{q_s}[i]$, respectively. For each IBF that the cloud checks against using t_{q_s}, for each ordered pair of hashes in t_{q_s}, the first hash locates the twin that the corresponding keyword is hashed to and the second hash locates the cell in the twin. If there exists $1 \leq i \leq u \times k$ so that $B_p[t_{q_s}[i].f][H(t_{q_s}[i].s \oplus r_p)] = 0$, then t_{q_s} does not match any of the data items. If the cloud determines that t_{q_s} matches no data items, the query processing terminates. Otherwise, the cloud processes t_{q_s} against the left and right children of the root. The above process recursively applies to subtrees in the IBtree. The search process terminates when the cloud determine that t_{q_s} matches no data items or we finish searching at terminal nodes. When the search process reaches a terminal node N_j, which is associated with IBF B_j and random number r_j, and the cloud determines that t_{q_s} matches the data item associated with N_i, then the cloud processes each checking sub-trapdoor in the checking component t_{q_c} to determine whether the data item associated with N_j, denoted by d_j, should be included in the query result of t_q. If there exists one checking sub-trapdoor $t^x_{q_c} \in t_{q_c}$ such that for each $1 \leq i \leq k$, $B_j[t^x_{q_c}[i].f][H(t^x_{q_c}[i].s \oplus r_j)]$=1, then d_j is excluded from the result of t_q. Otherwise the result includes d_j.

We now analyze the time complexity of our query processing algorithm in the worst case and show that it is sub-linear in the number of data items. Let n be the number of data items indexed by the IBtree, u be the number of dimensions, q_s be the searching component of q and $q_s = q_1 \wedge q_2 \wedge \cdots q_u$, R_i be the query result of $q_i (1 \leq i \leq u)$, and R be the query result of q_s. Thus, $R = \cap_{i=1}^{u} R_i$. Let $|R_x|$ be the minimum among $|R_1|, \cdots, |R_u|$. In the worst case, there are at most $|R_x|$ internal nodes in a same layer of the IBtree whose IBFs satisfy the query q_s. The search time for each data item in R_x is $O(\log n)$. Thus, the time complexity of our query processing algorithm is $O(|R_x| \log n)$. In reality, $|R_x| \ll n$ as n is typically large and $|R_x|$ is the minimum among $R_1, \cdots, R_u$. Thus, our query processing time is sublinear.

IV. Optimized IBtree Algorithms

A. IBtree Traversal Width Minimization

Recall that in the IBtree construction algorithm in Section III-C, for a nonterminal node with data set D, we partition D into two subsets D_l and D_r so that $0 \leq |D_l| - |D_r| \leq 1$. This partition is critical for the performance of query processing on the IBtree because a sub query $q_i \subseteq W(D_l) \cap W(D_r)$ will lead to the traversal of both subtrees. Thus, in partitioning D into D_l and D_r in addition to satisfying the condition $0 \leq |D_l| - |D_r| \leq 1$, we want to minimize $|W(D_l) \cap W(D_r)|$. We call this problem *Equal Size Data Set Partition*, which is a general version of *Equal Size Prefix Family Partition* defined in [32]. According to the theory of NP-Completeness, the *Equal Size Data Set Partition* is NP-hard as its special case *Equal Size Prefix Family Partition* is NP-hard.

Next, we present our approximation algorithm to the equal size data set partition problem. For a multidimensional data set D, as $|W(D)|$ is many times larger than $|D|$, partitioning a multidimensional data set is more complicated than partitioning a 1-dimensional data set of equal size. We have the following two observations. First, a search processing for a query q_s is terminated on an internal node v of an IBtree if there exists at least one component q_s^j of q_s such that no data item in D_v satisfies q_s^j, where D_v is the data set labeled with node v in an IBtree. Second, because in order to construct an IBtree for a multidimensional data set D, we need to partition D into multiple subsets in an recursive manner till each subset only contains one data item, given a multidimensional data set D, we only need to choose one dimension in a turn to partition D, which remarkably reduces the size of the keyword set that needs to be dealt with in the partition process, and we alternatively choose different dimensions in turn to partition.

There are two types of attributes: numeric and non-numeric. For numeric attributes, we use the partition algorithm proposed in [32]. Next, we propose a graph based keyword partition algorithm to deal with non-numeric attributes.

Given a data set $D = \{d_1, d_2, \cdots, d_n\}$, suppose the x-th attribute is non-numeric, we construct a weighted undirected graph $G = (V, E)$ as follows. For each data item d_i, we create a node v_i with label $L(v_i) = \{d_i\}$. Let the *size* of a node v_i be the number of data items in v_i, *i.e.*, $|L(v_i)|$. Thus, at this initial stage, the size of each node is 1. For any two different data items d_i and d_j, we create an edge $e_{i,j}$ between v_i and v_j where the weight of $e_{i,j}$ is $|W(d_i^x) \cap W(d_j^x)|$, *i.e.*, the number of common keywords shared by d_i and d_j on x-th attribute.

After graph $G = (V, E)$ is constructed, we perform node merging as follows. Our goal is to merge the n nodes into two nodes, which represents our partition of the n data items. For any two nodes v_i and v_j, if $|L(v_i)| + |L(v_j)| \leq \lceil n/2 \rceil$, then we say v_i and v_j are *mergeable*. Among all mergeable pairs of nodes, we select the pair whose edge has the largest weight. For any two nodes v_i and v_j that we choose to merge, we merge them into a new node $v_{i,j}$ with label $L(v_{i,j}) = L(v_i) \cup L(v_j)$ and then update the weight of all edges connected with this new node. For any node v connected by an edge with node $v_{i,j}$, we update the weight of the edge to be $W(L(v)^x) \cap W(L(v_{i,j})^x)$. This merging process repeats until no two nodes can be merged.

When there are no two nodes that can be merged, the resulting graph has either exactly two nodes or exactly three nodes because of the mergeable condition $|L(v_i)| + |L(v_j)| \leq \lceil n/2 \rceil$. It cannot be merged into only one node as otherwise the node size n is larger than $\lceil n/2 \rceil$. The resulting graph will not have four or more nodes because it must be able to further merge. For example, if the resulting graph has four nodes v_1, v_2, v_3, v_4 and no two nodes can be merged, then $|L(v_1)| + |L(v_2)| > \lceil n/2 \rceil$ and $|L(v_3)| + |L(v_4)| > \lceil n/2 \rceil$; thus, we have $|L(v_1)| + |L(v_2)| + |L(v_3)| + |L(v_4)| > n$, which contracts with the fact that $\Sigma_{i=1}^{4} |L(v_i)| = n$.

Next, we discuss these two cases. If the resulting graph has exactly two nodes v_1, v_2, without loss of generality, assuming $|L(v_1)| \geq |L(v_2)|$, then the equal size partition solution is $D_l = L(v_1)$ and $D_r = L(v_2)$. Note that $0 \leq |D_l| - |D_r| \leq 1$ because $|D_l| \leq \lceil n/2 \rceil$, $|D_r| \leq \lceil n/2 \rceil$, and $|D_l| \geq |D_r|$. If the resulting graph has exactly three nodes v_1, v_2, v_3, without loss of generality, assuming $|L(v_1)| \geq |L(v_2)| \geq |L(v_3)|$, we redistribute the data items in $L(v_3)$ into nodes v_1 and v_2 as follows so that the resulting two nodes represent the equal size partition solution. For each data item d_g in $L(v_3)$, we calculate its *distance values* to v_1 and v_2, namely $|W(d_g^x) \cap |W(L(v_1)^x)|$ and $|W(d_g^x) \cap |W(L(v_2)^x)|$, respectively. Among all these distance values, we choose the highest distance value. Suppose it is the distance between data item d_g and node v_1. If $|L(v_1)| + 1 \leq \lceil n/2 \rceil$, we first redistribute d_g into node v_1 and then for each remaining data item in $L(v_3)$, we recalculate its new distance values to v_1 and v_2 and repeat this redistribution process. If $|L(v_1)| + 1 > \lceil n/2 \rceil$, then we redistribute all nodes in $L(v_3)$ to node v_2 and the redistribution process terminates.

We now analyze the worst case computational complexity of our equal size data set partition algorithm. Let n be the size of the given data set. We first need $O(n^2)$ time to build the graph G. The node merging process takes $O(|E|)$ time, where E is the set of edges in G and we have $|E| \leq n^2$. The data item redistribution process takes a maximum of $O(n)$ time. Thus, the time complexity of our algorithm is $O(n^2)$.

B. IBtree Traversal Depth Minimization

Our basic idea for minimizing IBtree traversal depth is based on the following observation: for any nonterminal node v with label $L(v) = \{d_1, d_2, \cdots, d_g\}$ that a sub query q_i of a conjunctive query q traverse, if $q_i \subseteq W(d_1) \cap W(d_2) \cap \cdots \cap W(d_g)$, then all terminal nodes of the subtree rooted at v satisfy the query; thus, instead of traversing this subtree level by level downward, we can directly jump to the left most terminal node of this subtree and use the link list pointers to collect all data items of this subtree. This is why we link all leaf nodes of an IBtree together.

The key challenge is that when we traverse to node v in processing q_i, we need to quickly determine whether $q_i \subseteq$

$W(d_1) \cap W(d_2) \cap \cdots \cap W(d_g)$. Note that the IBF of node v allows us to determine whether $q_i \subseteq W(d_1) \cup W(d_2) \cup \cdots \cup W(d_g)$. Our solution is to precompute such information when we construct the IBtree and extend the IBF data structure to store this extra information in IBF twins. Specifically, for each nonterminal node v whose IBF is denoted as B_v and for each twin $B_v[i]$, we extend the twin to store one indicator bit denoted as $B_v[i].\texttt{ind}$ whose value is 1 if and only if for every node in the subtree rooted at v, we have $B_u[i].\texttt{ind} = 1$. For any terminal node u, $B_u[i].\texttt{ind} = 1$ if and only if the value of the i-th twin $B_u[i]$ is 1. With all indicator bits in place, when we process q_i at node v, if for all the twins representing q_i, their indicator bits are all 1s, then we know that $q_i \subseteq W(d_1) \cap W(d_2) \cap \cdots \cap W(d_g)$.

In addition to helping IBtree traversal depth minimization, the indicator bits also help to improve query processing efficiency. When we process q_i at nonterminal node v and j is the location in t_{q_i}, if $B_v[i].\texttt{ind} = 1$, then when we process q_i at any node in the subtree rooted at v, we do not need to examine the value of the i-th twin of the IBF for that node because its value must be 1. This saves the operations of computing HMAC and SHA1 values.

C. IBtree Compression

When the number of dimensions and the number of bits for each dimension are constants, both the space and time complexity for constructing an IBtree over a data set D are $O(n^2)$ because the IBtree contains $2n-1$ nodes and each node contains an IBF of size $O(n)$, where n is the number of data items in D. We now present our IBtree space compression algorithm to make our IBtree data structure scalable in terms of both space and time. Our basic idea is to remove the twins whose stored information is redundant from IBFs. There are two types of information redundancy in the IBFs of an IBtree: 0-redundancy and 1-redundancy.

0-redundancy: Let B_v be the IBF of a nonterminal node v, B_l and B_r be the IBFs of v's left and right child, respectively. According to our IBtree construction algorithm, for each twin $B_v[i]$, its value is the logical OR of the corresponding twins of B_l and B_r. Thus, if the value of $B_v[i]$ is 0, then the values of both $B_l[i]$ and $B_r[i]$ are 0. Applying this analysis recursively, if the value of $B_v[i]$ is 0, then for any node u in the subtree rooted at node v, the value of $B_u[i]$ is 0. For any sub query q_i that involves checking $B_v[i]$, because the value of $B_v[i]$ is 0, which means that $q_i \not\subseteq W(L(v))$, we will not further process the query against the subtree rooted at v. Thus, for any node u in the subtree rooted at node v, the information stored in $B_u[i]$ is redundant.

1**-redundancy**: According to our IBtree traversal depth minimization algorithm, when we process sub query q_i at nonterminal node v, if $B_v[i].\texttt{ind} = 1$, then when we process q_i at any node in the subtree rooted at v, we do not need to examine the value of the i-th twin of the IBF for that node because its value must be 1. Thus, if $B_v[i].\texttt{ind} = 1$, then for any node u in the subtree rooted at v, the information stored in $B_u[i]$ is redundant.

Eliminating 0-redundancy and 1-redundancy from an IBtree can save a large amount of space because the number of elements inserted to the IBFs in an IBtree reduces exponentially on average from top down based on the definition of IBtrees (assuming that the number of keywords in each data item is similar). Space saving will result time saving as the main part time for construction an IBtree is spending on initializing IBFs.

After removing both 0-redundancy and 1-redundancy from an IBF/IBtree, we call the resulting IBF a *compressed IBF/IBtree*. It is challenging to process a query against a compressed IBF because eliminating even one twin from an IBF changes the location of each twin after that eliminated twin. Thus, processing a sub query against a compressed IBF can no longer be based on the output of the c hash functions of the IBF. Recall that it is the data user who computes the c location-selector pairs using the k hash functions and then sends the c pairs to the cloud. Note that by definition the root IBF contains neither 0-redundancy nor 1-redundancy. This means that the root IBF remains the same after IBtree compression. Thus, processing a query on the root IBF is still based on the c twin locations received from the data user.

Next, we present our fast algorithm for processing a sub query q_i on a compressed IBF B_c whose uncompressed version is denoted by B_u. Basically we need to build a one to one mapping from the k locations $l_1, l_2, \cdots, l_k$, for each $1 \leq i \leq k$) that corresponds to the uncompressed IBF B_u to k new values $l'_1, l'_2, \cdots, l'_k$ that corresponds to the compressed IBF B_c so that for each $1 \leq i \leq k$, we have $B_u[l_i] = B_c[l'_i]$. We build this one to one mapping using perfect hashing functions constructed as follows. For each l_i, we first calculate $h_{k+1}(l_i)$. To maintain structure indistinguishability, we compress all IBFs in an IBtree, except the root IBF, into IBFs of the same size. Let m' be the size of the compressed IBFs. Second, for each $1 \leq j \leq g$ where g is a constant, we calculate $\texttt{SHA1}(h_{k+1}(l_i), j)\%m'$. Here we hash g times to allow each l_i to have g mapping options. Third, we use the Hungary algorithm to construct a bipartite matching from the k locations $l_1, l_2, \cdots, l_k$ to the k new locations $l'_1, l'_2, \cdots, l'_k$ in the compressed IBF of size m'. After this bipartite matching, each l_i chooses one of the g options. Suppose l_i is mapped to $\texttt{SHA1}(h_{k+1}(l_i), j)\%m'$, then we store j as the mapping option information in the corresponding twin of the parent IBF. We allocate an auxiliary IBF B_a for an IBtree to deal with mapping fails. If a location l_i cannot find a position in corresponding child IBFs, we map l_i into B_a by hashing. Given a data set, we can construct a compressed IBtree directly, instead of constructing an uncompressed IBtree and then compress it.

To support IBtree space compression, the chosen cell of a twin needs to store more information. The information includes the value of the twin, the indicator information, and the mapping option information. In practice, we combine two bits for denoting twin value and indicator information and two bits for the mapping option information. Thus, we can allocate one byte for a twin. We randomly generate 4-bit value for the unchosen cell in a twin.

Now we analyze the space complexity and time complexity of a compressed IBtree for n data items. The worst case space complexity is calculated in a bottom-up fashion as follows. Let c be the maximum number of keywords contained in one data item and k be the number of hash functions for IBFs. In a leaf IBF, the maximum number of twins whose value is 1 is ck. Now consider a nonterminal node whose two children are terminal nodes. The case that achieves the least compression is the following: for each twin with value 1 in one child, the corresponding twin in the other child has a value of 0. Suppose that we only perform compression to remove 0-redundancy. In

this case, for the parent IBF, the number of twins of value 1 is $2ck$. Recursively applying this analysis, the number of twins of value 1 is $2ckn$ at each level. As the IBtree has $O(\log n)$ levels, the total number of twins of value 1 is $O(2ckn \log n) = O(n \log n)$ as both c and k are constants. Thus, the worst case space complexity of our compressed IBtree construction algorithm is $O(n \log n)$.

In basic IBtree construction algorithm, the main part time is consumed by initializing IBFs. We only allocate a few IBFs with $O(n)$ size, and reuse these IBFs in the whole construction of a compressed IBtree to reduce the time complexity. The time consumed in the construction of a compressed IBtree can be divided into two parts: initializing new IBFs and mapping elements by Hungary algorithm. We have analyzed that the number of cells of all IBFs in an IBtree is $O(n \log n)$. Also, there are $O(n \log n)$ keywords need to be mapped in the whole construction procedure. The time complexity of Hungary algorithm depends on the ratio n_p/n_e, where n_p and n_e denote the number of available positions for mapping and the number of elements to be mapped, respectively. When a proper value is given to the ratio such as $n_p/n_e \geq 1.2$, the most of elements can find a place in g testing without affecting the already mapped elements, here g is the number mapping options. Thus the time complexity of Hungary algorithm is almost linear. Combining the IBF initializing part and element mapping part, the complexity of our compressed IBtree construction is $O(n \log n)$.

V. Security Analysis

We now prove that our IBtree based privacy preserving keyword query scheme is secure under the adaptive IND-CKA model. We use `HMAC` and `SHA1` to implement the hash functions $h_1, h_2, \cdots, h_{k+1}$ and the hash function H, respectively. Both `HMAC` and `SHA1` are pseudo-random hash functions. A function is a pseudo-random hash function if and only if the output of this function and the output of a truly random function are not distinguishable to a probabilistic polynomial time adversary [24], [28]. Let $\mathtt{G} : \{0,1\}^e \times \{w_1, w_2, \cdots, w_m\} \rightarrow \{0,1\}^e$ be a function that takes as input a keyword and an e-bit key and maps to an e-bit string. Let $\mathtt{g} : \{w_1, w_2, \cdots, w_m\} \rightarrow \{0,1\}^e$ be a truly random function that takes as input a keyword and maps it to an e-bit string. Here `G` is a pseudo-random hash function if and only if, for a fixed value $k \in \{0,1\}^e$, $\mathtt{G}(x, k)$, where $x \in \{w_1, w_2, \cdots, w_m\}$, can be computed efficiently and a probabilistic polynomial time adversary $\mathcal{A}$ with access to r chosen evaluations of `G`, *i.e.*, $(x_i, \mathtt{G}(x_i, k))$ where $i \in [1, r]$, cannot distinguish the value $\mathtt{G}(x_{r+1}, k)$ from the output of `g`. A privacy preserving query scheme is secure if a probabilistic polynomial time adversary cannot distinguish the output of a real index, which uses pseudo-random functions, from the output of a simulated index, which uses truly random functions, with a non-negligible probability.

We view an IBtree as a set of IBFs that are organized in a tree structure where each IBF stores a set of keywords. We consider a probabilistic polynomial-time *adaptive adversary* who can view the result of past trapdoor queries and results before choosing the next query in the simulation. In essence, to prove a scheme is secure in the IND-CKA model against an adaptive adversary, we need to show the existence of a probabilistic polynomial-time simulator that has the ability to simulate future unknown queries. An adaptive adversary will not be able distinguish between the results of the real secure index and those of the simulator with a non-negligible probability. We next construct such a simulator for IBtree scheme and prove that our IBtree scheme is IND-CKA secure against an adaptive adversary. Note that an IBF can also be viewed as an IBtree that has only the root node. Thus, if our IBtree scheme is adaptive IND-CKA secure, our IBF is adaptive IND-CKA secure too. We define two leakage functions as follows:

$\mathcal{L}_1(\mathtt{I}, \mathtt{D})$*:* Given the index `I` and data set D, this function outputs the size of each IBF, the number of data items n in D, the data item identifiers $ID = \{id_1, id_2, \cdots, id_n\}$, and the size of each encrypted data item encrypted by a CPA-secure encryption scheme.

$\mathcal{L}_2(\mathtt{I}, \mathtt{D}, q_i, t)$*:* This function takes as input the index `I`, the set of data items D, and a sub query q_i as a query at time t. It outputs two types of information: the *search pattern*, which is the information about whether the same search was performed before time t or not, and the *access pattern*, which is the information about which data items contain q_i at time t.

Theorem 5.1 (Security): The optimized IBtree is IND-CKA $(\mathcal{L}_1, \mathcal{L}_2)$-secure against an adaptive adversary.

Proof: We first construct a simulator $\mathcal{S}$ that can build a simulated index based on the information revealed by the leakage function $\mathcal{L}_1(\mathtt{I}, D)$ as follows. It simulates the encrypted data items $D = \{d_1, d_2, \cdots, d_n\}$ using the simulator $\mathcal{S}_E$, which is guaranteed to exist by the CPA-security of Enc, together with the value n and the size of each encrypted date item, which are both included in the output of $\mathcal{L}_1(\mathtt{I}, D)$. To simulate IBtree index `I`, it constructs an IBtree T according to IBtree Definition 3.2 using the identifiers $ID = \{id_1, id_2, \cdots, id_n\}$ included in the leakage function. It then picks m random e-bit strings $(s_1, s_2, \cdots, s_m)$ as m keys for the m twin numbers to be used for choosing cells. Note that m is included in the leakage function $\mathcal{L}_1$.

Now for each node v in T, the simulator sets up an IBF B_v with the same size of the corresponding IBF in `I`. In the i-th twin of B_v, the simulator stores either 0 at $B_v[i][0]$ and 1 at $B_v[i][1]$, or vice verse. This is decided for each twin by flipping a coin. The simulator also store the information of coin values locally. Finally, the simulator outputs this simulated index T to the adversary.

Next, we show how the simulator simulates queries on T. Let B_p denote the IBF in the root of T. If $B_p[l_i]$=1 and for any IBF B that is in the path from the root to the leaf node for data item d, B has a location l_i^B corresponding to l_i by a random hash function and $B[l_i^B]$=1, then we say d can be reached by l_i. The data set of location l_i, denoted as $D(l_i)$, is the set of all data items that can be reached by l_i. Suppose the simulator receives a query q_i. According to the leakage function $\mathcal{L}_2(\mathtt{I}, D, q_i, t)$, the simulator knows whether this query has been performed before or not due to the revealed search pattern in $\mathcal{L}_2$. If it has been performed before, the simulator outputs the same trapdoor t_{q_i} to the adversary. Otherwise, the simulator generates a new trapdoor t_{q_i} as follows. Recall that each sub-trapdoor in a trapdoor for a query is the set of c locations determined by k' secure hashes. The simulator associates an e-bit string for each twin to simulate the secure hashes. The simulator knows the value of c from the leakage function $\mathcal{L}_2$. Suppose the trapdoors issued by the simulator have a total of p locations. Note that the simulator has stored locally the data sets of these p locations. From the access pattern revealed in

$\mathcal{L}_2$, the simulator knows the query result set R_{q_i} for each sub query q_i. If the simulator can find a new combination of c locations $\{l_1, l_2, \cdots, l_c\}$ from the p locations so that these c locations satisfy the condition $D(l_1) \cap D(l_2) \cap \cdots \cap D(l_c) = R_{q_i}$, then the simulator issues these c locations with their corresponding c random e-bit strings as the trapdoor for q_i. Otherwise, the simulator randomly chooses $h(h < c)$ locations $l_y, l_{y+1}, \cdots, l_{y+h-1}$ so that these h locations satisfy the condition $R_{q_i} \subseteq D(l_j)(1 \leq j \leq h)$. Then, the simulator chooses c-h locations $l_x, l_{x+1}, \cdots, l_{x+c-h-1}$ from the rest m-p unused locations, and associates a unique data set with each of these c-h locations so that the following condition is satisfied: $D(l_x) \cap \cdots \cap D(l_{x+c-h-1}) \cap D(l_y) \cap \cdots \cap D(l_{y+h-1}) = R(q_i)$.

The goal of associating a data set $D(l_i)$ with an unused location l_i in B_p for satisfying the above equation can be achieved by using the random oracle. The bit b that the random oracle outputs for a twin in the IBF of a tree node v is programmed by the simulator as follows: if a data item d_x needs to be included in $D(l_i)$ and v is on the path from the root to the leaf node for d_x, then the simulator let b be the bit so that $B_v[l_i^{Bv}][b]$=1 and randomly select an unused location in both its left and child IBFs to correspond to l_i^{Bv}, respectively, where B_v is the IBF of node v and l_i^{Bv} is the location in B_v that corresponds to l_i in B_p. Otherwise, if for any leaf node of data item d in $D(l_i)$ where $d \neq d_x$, v is not in the path from the root to this leaf node and the parent node of v is in that path, then the simulator let b to be the bit so that $B_v[l_i^{Bv}][b]$=0 and choose no locations in the IBFs of its left and right child nodes to correspond to l_i^{Bv}.

Now if a probabilistic polynomial time adversary issues a query, the simulator can generate a trapdoor for this query as above. The trapdoor given by the simulator and the query result produced by the simulated index T are indistinguishable to the adversary because of the pseudo-random function and CPA-secure encryption algorithm. Thus, our scheme is IND-CKA secure against an adaptive adversary. ■

VI. Experimental Evaluation

A. Experimental Methodology

The key factors that affect the performance of an IBtree are data set sizes, IBtree types, dimension numbers, and query types. Base on these factors, we generated various experimental configurations and comprehensively evaluated construction time, index sizes, and query processing time.

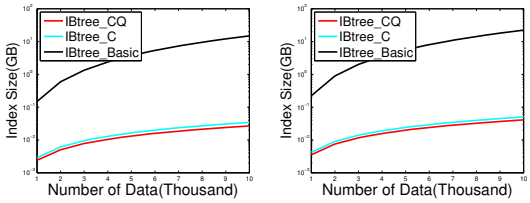
Data Sets: We chose two data sets to evaluate our schemes. One data set that we choose is the Gowlla data set [15], which is used in [32]. Gowlla data set consists of 6,442,890 check-in records of users over the period of Feb. 2009 to Oct. 2010. We extracted time stamps to form 1-dimensional data sets, time stamps and User IDs to form 2-dimensional data sets, and time stamps, User IDs, and locations to form 3-dimensional data sets. We performed a binary encoding for each attribute and then computed a keyword set for each dimension value in a data set. We generated two categories of data sets, where each category has 10 data sets. The data set sizes in Category I range from 1K to10K. We generated these 10 data sets of small sizes because our basic IBtree scheme cannot support data sets containing large number of data items. The data set sizes in Category II range from 0.5 million to 5million. Each data set was chosen randomly from the 6 million-plus total records in the Gowalla data set. We generated these 10 data sets of large sizes for testing the scalability and efficiency of our optimized IBtree scheme and for comparing our optimized IBtree scheme with $PBtree\_\texttt{WD}$, which is the optimized range query processing scheme in [32].

The other data set is the America NSF Research Award Abstract set [3], which consists of 129K abstracts describing NSF awards from 1990-2003. We extract keywords from each abstract. The number of keywords in a document ranges from 15 to 454 and the average number of keywords in a document is 99.27. We generate one category named Category III of data sets that contains 10 document sets with sizes ranging from 1K to 10K. We compare our optimized IBtree scheme with KRB scheme proposed in [24] on the data sets in this category for keyword query processing. We generate these 10 documents sets of small sizes because the KRB scheme cannot support document sets of large sizes due to the unscalable space requirement.

IBtree Types: To evaluate the effectiveness of our IBtree optimization algorithms, we compared the performance of the following three schemes: (1) `IBtree_Basic`, which is the basic IBtree scheme without any optimization, (2) `IBtree_C`, which is the IBtree with space compression, (3) `IBtree_CQ`, which is the `IBtree_C` plus traversal width and depth minimization. As `IBtree_Basic` is unscalable, we first evaluated all these three schemes on data sets in Category I, then we evaluated `IBtree_C` and `IBtree_CQ` on data sets in Category II. Note that space compression is critical for scaling our IBtree scheme to data sets of large sizes.

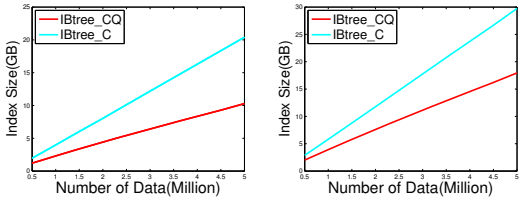
Query Types: We generated three kinds of queries: conjunctive queries, range queries, and keyword queries. We experimented two-dimensional and three dimensional conjunctive queries that contain both keyword conditions and range conditions on different IBtree schemes. For 2-dimensional queries, we generated keyword query conditions for user IDs and range query conditions for time-stamps. For 3-dimensional queries, we generated keyword query conditions for user IDs and range query conditions for both time stamps and locations. We experimented range queries on $IBtree\_\texttt{CQ}$ and $PBtree\_\texttt{WD}$ and keyword queries on $IBtree\_\texttt{CQ}$ and KRB tree. For each data set in Category I, Category II, and Category III, we generated a distinct collection of 11 query sets, where each query set contains 1000 queries, and the average query result sizes in these query sets range from 0 to 100.

Implementation Details: We conducted our experiments on a PC server running Windows server 2008 R2 Enterprise with $160GB$ memory and two *Intel(R) Xeon(R) E5-2609 2.4 GHz* processors. We used $HMAC - SHA1$ as the pseudo-random function for IBFs and our space reduction algorithm. We implemented IBtree schemes using C++. We let each dimension of each data item has 32 bits. Thus, the number of unique keyword-record pairs is up to 495 million for the 3-dimensional data set containing 5 million records. There are two parameters that need to be assigned with proper values when we conduct our experiments on data sets of large sizes. One is the IBFs' parameter m/n, where n is the total number of keywords that we need to store in the root IBF and m is the size of the root IBF, and the other is ratio n_p/n_e in Hungary algorithm mentioned in Section IV-C. We chose proper values for m/n and n_p/n_e via experiments. We let m/n vary from 2 to 11 with a scaling factor 3. Experimental results show that the index size and query processing time increase as m/n increases. No false negative is observed when $m/n \geq 2$. However, when $m/n = 2$, the average false positive



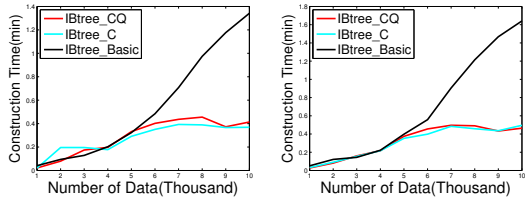
(a) 2-Dimensional Data (b) 3-Dimensional Data

Fig. 4: Index Size on Category I



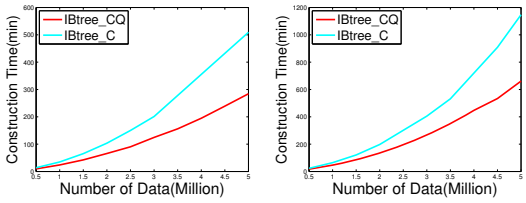
(a) 2-Dimensional Data (b) 3-Dimensional Data

Fig. 5: Index Size on Category II



(a) 2-Dimensional Data (b) 3-Dimensional Data

Fig. 6: Index Construction Time on Category I



(a) 2-Dimensional Data (b) 3-Dimensional Data

Fig. 7: Index Construction Time on Category II

rate is 1.39%, while $m/n \geq 5$, no false positive is observed. We let n_p/n_e vary from 1.1 to 1.6 with a scaling factor 0.1. Experimental results show that matching fail happens to some elements when $n_p/n_e \leq 1.3$. We observed that when $n_p/n_e = 1.1$, the average number of matching fail elements in a data set containing $10K$ elements is 429, while no mismatching elements is observed when $n_p/n_e \geq 1.4$. Thus, we set m/n=5 and n_p/n_e=1.4. Note that to achieve structure indistinguishable, given a data set D, we estimate the value of n_e for IBFs in each layer of the IBtree built for D in advance and let each IBF in same layers have same size.

B. Index Size

Experimental results show that the index size of `IBtree_CQ` *is scalable*. With 2 and 3 dimensional data set sizes growing from $1K$ to $10K$, the index sizes of `IBtree_CQ` grows from $2.4MB$ to $26.8MB$, and from $3.5MB$ to $41MB$, respectively. With 2 and 3 dimensional data set sizes growing from 0.5 million to 5 million, the index sizes of `IBtree_CQ` grows from $1.21GB$ to $10.28GB$, and from $2.02GB$ to $17.95GB$, respectively.

Experimental results show that our space reduction algorithm significantly reduces index space. `IBtree_CQ` and `IBtree_C` consume orders of magnitude less space than `IBtree_Basic`. For example, for 2-dimensional data sets of $10K$, the index sizes of `IBtree_CQ`, `IBtree_C`, and `IBtree_Basic`, are $26.8MB$, $34.4MB$, and $14.9GB$, respectively. Figure 4, which uses logarithmic coordinates, shows the index sizes for these three schemes.

Experimental results show that our traversal width and depth minimization algorithms also reduce index size. For example, for 2-dimensional data sets of 5 million, the index sizes of `IBtree_CQ` and `IBtree_C` are $10.28GB$ and $20.40GB$, respectively. Figure 5 shows the index sizes of these two schemes.

C. Index Construction Time

Experimental results show that the index construction time of `IBtree_CQ` *is practically acceptable.* With 2 and 3 dimensional data set sizes growing from $1K$ to $10K$, the index construction time of `IBtree_CQ` grows from 0.02 to 0.41 minutes, and from 0.05 to 1.21 minutes, respectively. With 2 and 3 dimensional data set sizes growing from 0.5 million to 5 million, the index construction time of `IBtree_CQ` grows from 9.6 to 284.3 minutes, and from 17.2 to 661.4 minutes, respectively.

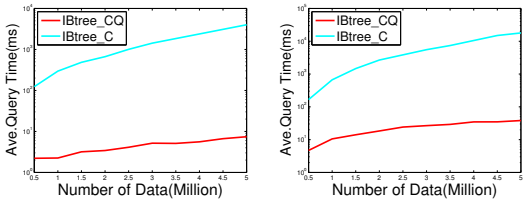
Experimental results show that our space reduction algorithm significantly reduce index construction time. The index construction time of `IBtree_CQ` and `IBtree_C` are also orders of magnitude less than that of `IBtree_Basic`. For example, for 2-dimensional data sets of $10K$, the index construction time of `IBtree_CQ`, `IBtree_C`, and `IBtree_Basic` are 0.41, 0.37, and 1.34 minutes, respectively. Figure 6 shows the construction time for these three schemes.

Experimental results show that our traversal width and depth optimization algorithms also can reduce index construction time. For example, for 2-dimensional data sets of 5 million, the construction time of `IBtree_CQ` and `IBtree_C` are 284.3 minutes and 509.2 minutes, respectively, as shown in Figure 7.

D. Query Processing Time

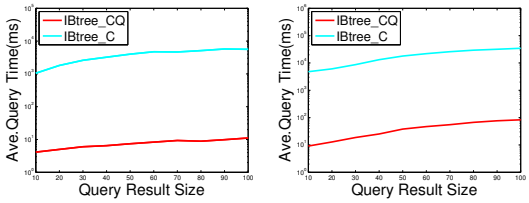
Experimental results show that the query processing time of `IBtree_CQ` *is in the millisecond scale.* When we fixed the query result size to be 50, with 2 and 3 dimensional data set sizes growing from 0.5 million to 5 million, the average query processing time of `IBtree_CQ` grows from 2.2 to 7.4 milliseconds, and from 4.7 to 38.1 milliseconds, respectively. When we fix the data set size to be 5 million, with 2 and 3 dimensional data set sizes growing from 0 to 100, for that of 2-dimensional data, the average query processing time of `IBtree_CQ` grows from 4.1 to 11.1 milliseconds, and from 9.1 to 83.2 milliseconds, respectively.

Experimental results show that our traversal width and depth minimization algorithms significantly improve searching efficiency. The time consumed in processing queries by `IBtree_CQ` is orders of magnitude less than `IBtree_C`. When we fixed the query result size to be 50, with data set sizes growing from 0.5 million to 5 million, for the average query processing time of 2-dimensional data, `IBtree_CQ` grows from 2.2 to 7.4 milliseconds, whereas `IBtree_C` grows from 123.7 to 3996.4 milliseconds; for that of 3-dimensional data, `IBtree_CQ` grows from 4.7 to 38.1 milliseconds, whereas



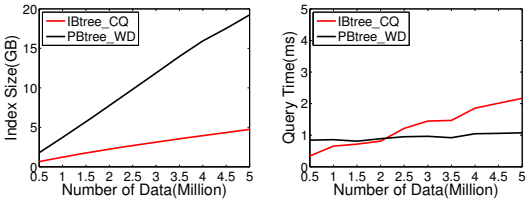
(a) 2-Dimensional Data (b) 3-Dimensional Data

Fig. 8: Ave. Query Time when $|R| = 50$



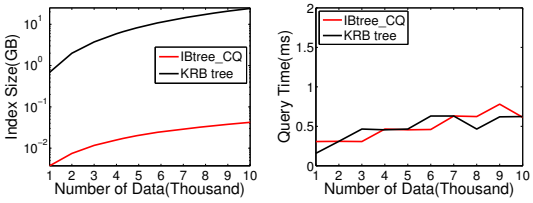
(a) 2-Dimensional Data (b) 3-Dimensional Data

Fig. 9: Ave. Query Time when $|D| = 5$ million



(a) Index Size (b) Query Time ($|R| = 50$)

Fig. 10: Compared with PBtree



(a) Index Size (b) Query Time ($|R| = 50$)

Fig. 11: Compared with KRB tree

IBtree_C grows from 166 to 17911 milliseconds. When we fix the data set size to be 5 million, with result set sizes growing from 0 to 100, for that of 2-dimensional data, IBtree_CQ grows from 4.1 to 11.1 milliseconds, whereas IBtree_C grows from 1049.4 to 5621.5 milliseconds; for that of 3-dimensional data, IBtree_CQ grows from 9.1 to 83.2 milliseconds, whereas IBtree_C grows from 4791 to 34350 milliseconds. Figure 8 and figure 9, both of which use logarithmic coordinates, show the average query processing time for these two schemes.

E. Compared with PBtree and KRB

We compared IBtree_CQ with $PBtree\_WD$ on data sets in Category II with range queries and IBtree_CQ with KRB on data sets in Category III with keyword queries. As the space limitation, we only show the experimental results of index sizes and query processing time with query result size $|R| = 50$.

Experimental results show that IBtree_CQ *consumes less space than* $PBtree\_WD$. With $|D|$ growing from 0.5 million to 5 million, the IBtree_CQ size grows from 0.659GB to 4.744GB. Whereas, the PBtree sizes grows from 1.785GB to 19.243GB. When data sets of size 0.5million, IBtree_CQ consumes 1.708 times less space than $PBtree\_WD$. When data sets of size 5 million, IBtree_CQ consumes 3.056 times less space than $PBtree\_WD$. Figure 10(a) shows the index size for these two schemes.

Experimental results show that IBtree_CQ *is as fast as* $PBtree\_WD$ *on query processing.* When $|D| = 0.5$million and $|R| = 50$, the average query processing time of IBtree_WD and $PBtree\_WD$ tree are 0.342 ms and 0.843 ms, respectively. When $|D| = 2.5$million and $|R| = 50$, the average query processing time of IBtree_WD and $PBtree\_WD$ are 1.217ms and 0.952ms, respectively. When $|D| = 5$million and $|R| = 50$, the average query processing time of IBtree_WD and $PBtree\_WD$ are 2.168ms and 1.075ms, respectively. Figure 10(b) shows the query time for these two schemes. Note that the Bloom ration in $PBtree\_WD$ is 10.

Experimental results show that the space consumed by $IBtree\_CQ$ *is orders of magnitude less than KRB.* With $|D|$ growing from 1K to 10K, the IBtree_WD size grows from 3.84 MB to 43.26 MB, whereas KRB grows from 704.96 MB to 24.24 GB. For document sets of size 1K and 10K, IBtree_WD consumes 181.96 and 572.73 times less space than KRB, respectively. Figure 11(a) shows the average index size for these three schemes. Note that we use logarithmic coordinates in Figure 11(a).

Experimental results show that IBtree_WD *is as fast as KRB on query processing.* When $|D| = 1$K and $|R| = 50$, the average query processing time of IBtree_WD and KRB are 0.307ms and 0.158 ms, respectively. When $|D| = 5$K and $|R| = 50$, the average query processing time of IBtree_WD and KRB are 0.456 ms and 0.465 ms, respectively. When $|D| = 10$K and $|R| = 50$, the average query processing time of IBtree_WD and KRB are 0.614ms and 0.624 ms, respectively. Figure 11(b) shows the query time of these two schemes.

VII. Conclusions

We make three key contributions in this paper. First, we propose the first privacy preserving conjunctive query processing scheme that achieves all three requirements of adaptive security, efficient query processing, and scalable index size. Our scheme embraces several novel ideas such as IBFs and IBtrees, which can be used in other applications. We formally prove that our scheme is adaptively secure under a random oracle model. Second, we propose several optimization algorithms such as IBtree traversal width and depth minimization algorithms and the IBtree space compression algorithm. Third, we implemented our scheme in C++ and evaluated its performance on two real-world data sets. Experimental results show that our scheme is fast in terms of query processing time and scalable in terms of index size.

Acknowledgment

This work is partially supported by the Natural Science Foundation of China under Grants No. 61370226, 61672156, 61472184, and 61321491, the National Science Foundation under Grants No. CNS-1318563, CNS-1524698, CNS-1421407, and IIP-1632051, and the Jiangsu Innovation and Entrepreneurship (Shuangchuang) Program. Alex X. Liu is also affiliated with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI, USA.

References

[1] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *Proc. of the International Conference on Management of Data (SIGMOD)*, pages 563–574. ACM, 2004.

[2] A. Arasu, S. Blanas, K. Eguro, and R. Kaushik. Orthogonal security with cipherbase. In *Proc. of the 6th Conf. on Innovative Data Systems Research(CIDR)*, 2013.

[3] K. Bache and M. Lichman. UCI machine learning repository, http://archive.ics.uci.edu/ml, 2013.

[4] M. Bellare, A. Boldyreva, and A. ONeill. Deterministic and efficiently searchable encryption. In *Proc. Inte. Cryptology Conf. (CRYPTO)*, pages 535–552, 2007.

[5] B. V. Ben Fisch, F. Krell, A. Kumarasubramanian, V. Kolesnikov, T. Malkin, and S. M.Bellovin. Malicious client security in blind seer: A scalable private dbms. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, pages 395–410. IEEE, 2015.

[6] A. Boldyreva, N. Chenette, and A. O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Proc. Inte. Cryptology Conf. (CRYPTO)*, pages 578–595, 2011.

[7] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Proc. of the Advances in Cryptology Eurocrypt (EUROCRYPT)*, pages 506–522, 2004.

[8] D. Boneh and B. Waters. Conjunctive, subset, and range queries on encrypted data. In *Proc. of the 4th Theory of Cryptography Conference (TCC)*, pages 535–554, 2007.

[9] R. Canetti, U. Feige, O. Goldreich, and M. Naor. Adaptively secure multi-party computation. In *Proc. 28th ACM symposium on Theory of computing (STOC)*, pages 639–648. ACM, 1996.

[10] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rosu, and M. Stiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*. ISOC, 2014.

[11] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Proc. Inte. Cryptology Conf. (CRYPTO)*, pages 353–373, August 2013.

[12] Y.-C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Proc. of the Third Inte. Conf. on Applied Cryptography and Network Security (ACNS)*, pages 442–455, 2005.

[13] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *Proc. of the Theroy and Application of Cryptology and Information Security(ASIACRYPT)*, pages 577–594, 2010.

[14] S. Kamara and T. Moataz. SQL on Structurally-Encrypted Databases. IACR ePrint 2016/453.

[15] E. Cho, S. A.Myers, and J. Leskovec. Friendship and mobility: User movement in location-based social networks. In *Proc. of the 17th ACM SIGKDD Inte. Conf. on Knowledge Discovery and Data Mining(KDD)*, pages 1082–1090. ACM, 2011.

[16] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proc. of the ACM Conf. on Computer and Communications Security(CCS)*, 2006.

[17] I. Demertzis, S. Papadopoulos, and O. Papapetrou. Practical private range search revisited. In *Proc. of the International Conference on Management of Data (SIGMOD)*. ACM, 2016.

[18] E. Goh. Secure indexes. Stanford University Technical Report, 2004.

[19] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *Journal of ACM*, 43(3):431–473, 1996.

[20] P. Gupta and N. McKeown. Algorithms for packet classification. *IEEE Network*, 15(2):24–32, 2001.

[21] H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *Proc. of the International Conference on Management of Data (SIGMOD)*, pages 216–227, 2002.

[22] B. Hore, S. Mehrotra, M. Canim, and M. Kantarcioglu. Secure multidimensional range queries over outsourced data. *The VLDB Journal*, 21(3):333–358, June 2012.

[23] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Proc. of the ISOC Network and Distributed System Security Symposium (NDSS)*. ISOC, 2012.

[24] S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Proc. of the Financial Cryptography and Data Security (FC)*, pages 258–274, 2013.

[25] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *Proc. of the ACM Conf. on Computer and Communications Security(CCS)*, pages 965–976, 2012.

[26] B. Bezawada, A. X. Liu, B. Jayaraman, A. Wang, and R. Li Privacy Preserving String Matching for Cloud Computing. In *the IEEE 35th Conf. on Distributed Computing Systems*, pages 609–618, 2015.

[27] P. Karras, A. Nikitin, M. Saad, R. Bhatt, D. Antyukhov, and S. Idreos. Adaptive indexing over encrypted numeric data. In *Proc. of the International Conference on Management of Data (SIGMOD)*, pages 171–183. ACM, 2016.

[28] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC Press, 2007.

[29] F. Kerschbaum and A. Schropfer. Optimal average-complexity ideal-security order-preserving encryption. In *Proc. of the ACM Conf. on Computer and Communications Security(CCS)*, pages 275–286, 2014.

[30] K. Kurosawa and Y. Ohtaki. Uc-secure searchable symmetric encryption. In *Proc. of the Financial Cryptography (FC)*, pages 285–298, 2012.

[31] J. Li and E. R. Omiecinski. Efficiency and security trade-off in supporting range queries on encrypted databases. In *Proc. of the 19th IFIP Annual Conference on Data and Applications Security and Privacy*, pages 69–83, 2005.

[32] R. Li, A. X.Liu, L. Wang, and B. Bezawada. Fast range query processing with strong privacy protection for cloud computing. In *Proc. of the 40th Inte. Conf. on Very Large Data Bases (VLDB)*, pages 1953–1964. IEEE, 2014.

[33] P. V. Liesdonk, S. Sedghi, J. Doumen, P. Hartel, and W. Jonker. Computationally efficient searchable symmetric encryption. In *Proc. of the 7th VLDB conference on Secure Data Management (SDM)*, pages 87–100, 2010.

[34] M. Naveed, S. Kamara, and C. V.Wright. Inference attack on property-preserving encrypted databases. In *Proc. of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 644–655. ACM, 2015.

[35] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovi. Blind seer: A scalable private dbms. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, pages 359–374. IEEE, 2014.

[36] R. A. Popa, F. H. Li, and N. Zeldovich. An ideal-security protocol for order-preserving encoding. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, pages 463–477, 2013.

[37] R. A. Popa, C. M.S.Redfied, N. Zeldovich, and H. Balakrish. Cryptdb: Protecting confidentiality with encrypred query processing. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 85–100. ACM, 2011.

[38] B. K. Samanthula, W. Jiang, and E. Bertino. Privacy-preserving complex query evaluation over semantically secure encrypted data. In *Proc. of the European Symposium on Research in Computer Security (ESORICS)*, pages 400–418. Springer International Publishing, 2014.

[39] E. Shi, J. Bethencourt, T.-H. H. Chan, D. Song, and A. Perrig. Multi-dimensional range queries over encrypted data. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, pages 350–364, 2007.

[40] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, pages 44–55. IEEE, 2000.

[41] B. Wang, Y. Hou, M. Li, H. Wang, and H. Li. Maple: Scalable multi-dimensional range search over encrypted cloud data with tree-based index. In *Proc. of the ACM Symposium on Information, Computer and Communication Security (ASIACCS)*, pages 111–122, 2014.

[42] P. Wang and C. Ravishankar. Secure and efficient range queries on outsourced databases using r-trees. In *Proc. of the IEEE International Conference on Data Engineering (ICDE)*, pages 314–325, 2013.

[43] W. K. Wong, D. W.-L. Cheung, B. Kao, and N. Mamoulis. Secure knn computation on encrypted databases. In *Proc. of the International Conference on Management of Data (SIGMOD)*, pages 139–152, 2009.

[44] A. Yao. How to generate and exchange secrets. In *Proc. of the 27th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 162–167. IEEE, 1986.

[45] B. Yao, F. Li, and X. Xiao. Secure nearest neighbor revisited. In *Proc. of the IEEE International Conference on Data Engineering (ICDE)*, pages 733–744. IEEE, 2013.