Fast Range Query Processing with Strong Privacy Protection for Cloud Computing

Rui Li[1,3] Alex X. Liu[2,3] Ann L. Wang[2] Bezawada Bruhadeshwar[3]
College of Computer Science and Electronic Engineering, Hunan university, ChangSha, China[1]
Dept. of Computer Science and Engineering, Michigan State University, East Lansing, MI, U.S.A[2]
National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China[3]
Email: lirui@hnu.edu.cn, alexliu@msu.edu, liyanwan@cse.msu.edu, bru@nju.edu.cn

ABSTRACT

Privacy has been the key road block to cloud computing as clouds may not be fully trusted. This paper concerns the problem of privacy preserving range query processing on clouds. Prior schemes are weak in privacy protection as they cannot achieve index indistinguishability, and therefore allow the cloud to statistically estimate the values of data and queries using domain knowledge and history query results. In this paper, we propose the first range query processing scheme that achieves index indistinguishability under the *indistinguishability against chosen keyword attack* (IND-CKA). Our key idea is to organize indexing elements in a complete binary tree called PBtree, which satisfies *structure indistinguishability* (*i.e.*, two sets of data items have the same PBtree structure if and only if the two sets have the same number of data items) and *node indistinguishability* (*i.e.*, the values of PBtree nodes are completely random and have no statistical meaning). We prove that our scheme is secure under the widely adopted IND-CKA security model. We propose two algorithms, namely PBtree traversal width minimization and PBtree traversal depth minimization, to improve query processing efficiency. We prove that the worse case complexity of our query processing algorithm using PBtree is $O(|R| \log n)$, where n is the total number of data items and R is the set of data items in the query result. We implemented and evaluated our scheme on a real world data set with 5 million items. For example, for a query whose results contain ten data items, it takes only 0.17 milliseconds.

1. INTRODUCTION

1.1 Background and Motivation

Driven by lower cost, higher reliability, better performance, and faster deployment, data and computing services have been increasingly outsourced to clouds such as Amazon EC2 and S3 [1], Microsoft Azure [3], and Google App Engine [2]. However, privacy has been the key road block to cloud computing. On one hand, to leverage the computing and storage capability offered by clouds, we need to store data on clouds. On the other hand, due to many reasons, we may not fully trust the clouds for data privacy. First, clouds may have corrupted employees who do not follow data privacy policies. For example, in 2010, a Google engineer broke into the Gmail and Google Voice accounts of several children [4]. Second, cloud computing systems may be vulnerable to external malicious attacks, and when intrusions happen, cloud customers may not be fully informed about the potential implications on the privacy of their data. Third, clouds may base services on facilities in some foreign countries where privacy regulations are difficult to enforce.

In this paper, we consider the following popular cloud computing paradigm: a data owner stores data on a cloud, and multiple data users query the data. For a simple example, a user stores his own data and queries his own data on the cloud. For another example, multiple doctors in a clinic store and query patient medical records in a cloud. Figure 1 shows the three parties in our model: 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. The problem addressed in this paper is range query processing on clouds in a privacy preserving and yet scalable manner. For a set of records where all records have the same attribute $\mathbb{A}$, which has numerical values or can be represented as numerical values, given a range query specified by an interval $[a, b]$, the query result is the set of records whose $\mathbb{A}$ attribute falls into the interval. Range queries are fundamental operations for database SQL queries and big data analytics. In database SQL queries, the `where` clauses often contain predicates specified as ranges. For example, SQl query `select * from patients where 20 <= age and age <= 30` means to find all records of the patients whose age is in the range of $[20, 30]$. In big data analytics, many analyses involve range queries along dimensions such as time and human age.

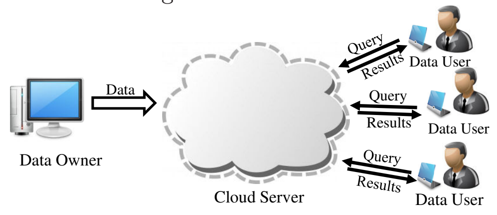


Figure 1: Cloud Computing Model

Given data items $d_1, \cdots, d_n$, the data owner encrypts these data using a symmetric key K, which is shared between the data owner and data users, generates an *index*, and then sends both the encrypted data denoted $(d_1)_k, \cdots, (d_n)_k$ and the index to the cloud. Given a query, the data user generates a *trapdoor* and then sends it to the cloud. The index and the trapdoor should allow the cloud to determine which data items satisfy the query. Yet, in this process, the cloud should not be able to infer *useful information* about the data and queries. The useful information in this context includes the values of the data items, the content of the queries, and the statistical properties of the data items. Other than encrypted data and encrypted queries, together with query results, the cloud may have information obtained from other channels, such as domain knowledge about the data (*e.g.*, age distribution). However, even with such information, a privacy preserving range query scheme should not allow the cloud to infer additional information about the data based on past query results.

Besides privacy guarantees, a privacy preserving range query scheme should be efficient in terms of query processing time, storage overhead, and communication overhead. The query processing time needs to be small because many applications require real-time queries. The storage overhead refers to the data that cloud needs to store other than encrypted data items. It needs to be small because the volume of data stored on the cloud is typically large. The communication overhead refers to the data transferred between the data owner and the cloud, other than encrypted data items, and the data transferred between data users and the cloud, other than the precise query results. It needs to be small due to bandwidth limitations and the extra time involved in uploading and downloading.

1.2 Threat Model

For the cloud, we assume that the cloud is *semi-honest* (also called *honest-but-curious*), which was proposed by Canetti *et al.* in [13] and has been widely adopted including prior privacy preserving range and keyword query work [8–11, 15–17, 19, 24–27, 31, 39]. A cloud is semi-honest means that it does follow required communication protocols and execute required algorithms correctly, but it may attempt to obtain information about the values of data items and the content of user queries with the help of domain knowledge about the data items and the queries (such as the distribution of data items and queries). For the data owner and the data users, we assume that they are trusted.

1.3 Security Model

We adopt the IND-CKA security model proposed in [19], which has been widely accepted in prior privacy preserving keyword query work. This model has two key requirements: *index indistinguishability (IND)* and *security under chosen keyword attacks (CKA)*. Informally, a range query scheme is secure under the IND-CKA model if an adversary $\mathcal{A}$ chooses two different sets S_1 and S_2 of data items, where the two sets have the same number of data items and they may or may not overlap, lets an oracle simulating the data owner to build indexes for S_1 and S_2, but $\mathcal{A}$ cannot distinguish which index is for which data set. The rationale is that if the problem is distinguishing the indexes for S_1 and S_2 is hard, then deducing at least one data item that S_1 and S_2 do not have in common must also be hard. In other words, if $\mathcal{A}$ cannot determine which data item is encoded in an index with probability non-negligibly different from 1/2, then the index reveals nothing about the data items. Such indexes are called *secure indexes*. The IND-CKA model aims to prevent an adversary $\mathcal{A}$ from deducing the plaintext values of data items from the index, other than what it already knows from previous query results or from other channels. Note that secure indexes do not hide information such as the number of data items. For applications that demand the privacy of data item numbers, they can inject dummy data items into small data sets to make all data sets to have equal sizes. Also, note that we are not interested in hiding search patterns, where a search pattern is defined as the set of trapdoors corresponding to different user queries. So far there are no searchable symmetric encryption schemes that can hide the statistical patterns of user searches because trapdoors are generated deterministically (*i.e.*, the same trapdoor will always be generated for the same keyword) [27].

1.4 Summary and Limitation of Prior Art

Prior privacy preserving query schemes fall into two categories according to their query types: range queries, which query all data items that fall into a given range, and keyword queries, which query all text documents that contain a given keyword. Privacy preserving range query schemes can also be called *range searchable symmetric encryption schemes*, and privacy preserving keyword query schemes can also be called *keyword searchable symmetric encryption schemes*. Prior privacy preserving range query schemes for the single-data-owner-multiple-data-user cloud paradigm fall into two categories: bucketing schemes [24–26] and order preserving schemes [9, 10, 31]. In bucketing schemes, the data owner partitions the whole data domain (*e.g.*, $[0, 150]$ of human ages) into multiple buckets of varying sizes (*e.g.*, 4 buckets of $[0, 12]$, $[13, 22]$, $[23, 60]$, $[61, 150]$). The index consists of pairs of a bucket ID and the encrypted data items in the bucket. The trapdoor of a range query (*e.g.*, $[10, 20]$) consists of the IDs of the buckets that overlaps with the range (*e.g.*, bucket IDs 1 and 2). All data items in a bucket are included in the query result as long as the bucket overlaps with the query. Bucketing schemes have two key limitations: weak privacy protection and high communication cost. Their privacy protection is weak because the cloud can statistically estimate the actual value of both data items and queries using domain knowledge and historical query results, as pointed out in [26]. Their communication cost is high because many data items in the query result do not satisfy the query. Reducing bucket sizes helps to reduce communication costs, but will worsen privacy protection because the number of buckets becomes closer to that of data items.

Order preserving schemes use encryption functions that preserve the relative ordering of data items even after encryption. For any two data items a and b, and an order preserving encryption function f, $a \leq b$ if and only if $f(a) \leq f(b)$. In order preserving schemes, the index for data items $d_1, \cdots, d_n$ are $f(d_1), \cdots, f(d_n)$, and the trapdoor for query $[a, b]$ is $[f(a), f(b)]$. Order preserving schemes have weak privacy protection because they allow the cloud to statistically estimate the actual values of both data items and queries [5].

The fundamental reason that the privacy protection provided by the above prior schemes is weak is because their indexes are distinguishable for the same number of data items but with different distributions. In bucketing schemes, for the same number of data items, different distributions in data values will cause buckets to have different distributions in sizes because they need to balance the number of items among buckets. In order preserving schemes, for the same number of data items, different distributions in data values will cause cipher-texts to have different distribution in the projected space. Leveraging domain knowledge about data distribution, both bucketing schemes and order preserving schemes allow the cloud to statistically estimate the values of data and queries.

1.5 Proposed Approach

In this paper, we propose the first privacy preserving range query scheme that achieves index indistinguishability. Our key idea for achieving index indistinguishability is to organize all indexing elements in a complete binary tree where each node is represented using a Bloom filter, which we call a *PBtree* (where "P" stands for privacy and "B" stands for Bloom filter). PBtrees allow us to achieve index indistinguishability because it has two important properties. First, a PBtree has the property of *structure indistinguishability*, that is, two sets of data items have the same PBtree structure if and only if the two sets have the same number of data items. The structure of the PBtree of a set of data items is determined solely by the set cardinality, not the value of data items. Second, a PBtree has the property of *node indistinguishability*, that is, for any two PBtrees constructed from data sets of the same cardinality, which have the same structure, and for any two corresponding nodes of the two PBtrees, the values of the two nodes are not distinguishable. Thus, our scheme prevents cloud from performing statistical analysis on the index even with domain knowledge.

1.6 Technical Challenges and Solutions

There are two key technical challenges. The first challenge is the *construction* of PBtrees by data owners. We address this challenge by first transforming less-than and bigger-than comparisons into set membership testing (*i.e.*, testing whether a number is in a set), which involves only equal-to comparisons, and then organize all the sets hierarchically in a PBtree. This transformation helps us to achieve node indistinguishability because the less-than or bigger-than relationship among PBtree nodes is no longer statistically meaningful. The second challenge is the *optimization* of PBtrees for fast query processing on the cloud. We address this challenge by two ideas: *PBtree traversal width minimization* and *PBtree traversal depth minimization*. The idea of *PBtree traversal width minimization* is to minimize the number of paths that the cloud needs to traverse for processing a query. We prove that the PBtree traversal width minimization problem is NP-hard, and propose an efficient approximation algorithm. The idea of *PBtree traversal depth minimization* is to minimize the traversal depth of the paths that the cloud needs to traverse for processing a query; in other words, we want the traversal of many paths to terminate as early as possible.

1.7 Key Contributions

We make three key contributions. First, we propose the first privacy preserving range query scheme that is secure under the widely adopted IND-CKA model. Second, we propose PBtrees, basic PBtree construction and query processing algorithms, and two PBtree optimization algorithms. Third, we implemented and evaluated our scheme on a large real world data set with 5 million data items. Experimental results show that our scheme is both fast and scalable. For example, for a query whose results contain ten data items, it takes only 0.17 milliseconds.

The rest of the paper proceeds as follows. We first review related work in Section 2. In Sections 3 and 4, we present our basic PBtree construction and query processing algorithms and two PBtree optimization algorithms. In Section 5, we prove that our scheme is secure under the IND-CKA security model. In Section 6, we show our experimental results. We conclude the paper in Section 7.

2. RELATED WORK

There are some privacy preserving range query work that does not fit into our cloud computing paradigm and cannot be used to solve the problem addressed in this paper. In the public-key domain, the approach in [38] supports range querying using identity based encryption primitives [12, 37]. Their encryption scheme allows a network gateway to encrypt summaries of network flows before submitting them to an untrusted repository; when a network operator suspects that an intrusion happens, a trusted third party can release a key to the operator to allow the operator to decrypt flows whose attributes fall within specified ranges, but not other flows. However, the user query privacy is not preserved.

A significant amount of work has been done in privacy preserving keyword queries [7,8,11,14–19,21,22,27,28,36,39]. However, these solutions are not optimized for range queries.

Prior work on outsourced databases has addressed problems such as secure kNN processing [32, 40, 41], privacy preserving data mining [6, 35], and query result integrity verification [30, 34, 42]. In [32, 40, 41], order preserving encryption techniques were used to compute the k-nearest neighbors of a given encrypted query point in an encrypted database. For the privacy preserving clustering mechanisms in [6, 35], certain confidential numerical attributes are perturbed in a uniform manner so that preserve the distances between any two points. Significant work has been done on query result integrity verification [30, 34, 42]. The basic idea is to include verifiable digital signatures for each returned tuple, which allow the client to verify the integrity of query results.

3. PBTREE CONSTRUCTION AND TRAPDOOR COMPUTATION

In this section, we first present our PBtree construction algorithm, which is executed by the data owner. This algorithm consists of three steps: prefix encoding, tree construction, and node randomization using Bloom filters. Second, we present our algorithm for computing the trapdoor for a given query, which is executed by the data users. With the PBtree of n data items and the trapdoor for a given query, the cloud is able to process the query on the PBtree without knowing the value of the data items and the query.

3.1 Prefix Encoding

The key idea of this step is to convert the testing of whether a data item falls into a range to the testing of whether two sets have common elements, where the basic step is testing whether two numbers are equal. To achieve this, we adopt the prefix membership verification scheme in [33]. Given a number x of w bits 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 number 6 of 5 bits is $F(6) = F(00110)$ ={00110, 0011*, 001**, 00***, 0****, *****}. Given a range $[a, b]$, we first convert the range $[a, b]$ to a minimum set of prefixes, denoted $S([a, b])$, such that the union of the prefixes is equal to $[a, b]$. For example, $S([0, 8])$ ={00***,1000}. Given a range $[a, b]$, where a and b are two numbers of w bits, the number of prefixes in $S([a, b])$ is at most $2w - 2$ [23]. 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 $x \in p$ holds. Furthermore, for any number x and prefix p, $x \in p$ 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 can see that $6 \in [0, 8]$ and $F(6) \cap S([0, 8]) = \{00 * **\}$. In this step, given n data items $d_1, \cdots, d_n$, the data owner computes the prefix families $F(d_1), \cdots, F(d_n)$; given a range $[a, b]$, the data user computes $S([a, b])$.

3.2 Tree Construction

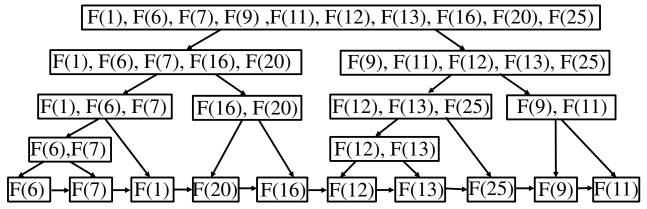
To achieve sub-linear search efficiency, we organize $F(d_1)$, $\cdots$, $F(d_n)$ in a tree structure that we call *PBtree*. We cannot use existing database indexing structures like $B+$ trees because of two reasons. First, searching on such trees (such as $B+$ trees) requires the operation of testing which of two numbers is bigger; however, PBtrees cannot support such operations for the cloud because otherwise PBtrees will share the same weaknesses with prior order preserving schemes [24–26]. Second, their structures for different sets of data items are often different even if the two sets have equal sizes; however, for any two sets of the same size, their PBtrees are required to have the same structure, *i.e.*, the two PBtrees are indistinguishable. In this paper, we organize $F(d_1), \cdots, F(d_n)$ using our PBtree structure.

DEFINITION 3.1 (PBTREE). *A PBTree for n data items is a full binary tree with n terminal nodes and n − 1 nonterminal nodes, where all n terminal nodes form a linked list from left to right and each node is represented using a Bloom filter. Each terminal node contains one data item, and each nonterminal node contains the union of its left and right children. For any nonterminal node, the size of its left child either equals to that of its right child or exceeds by one.*

According to this definition, a PBtree is a highly balanced binary search tree. The height of the PBtree for n data items is $\lfloor \log n \rfloor + 1$. We construct the PBtree from $F(d_1), \cdots, F(d_n)$ in a top-down fashion. First, we construct the root node, which is labeled with the n prefix families $\{F(d_1), \cdots, F(d_n)\}$. Second, we partition the set of n prefix families $\{F(d_1), \cdots, F(d_n)\}$ into two subsets of prefix families $S_{\texttt{left}}$ and $S_{\texttt{right}}$ such that $|S_{\texttt{left}}| = |S_{\texttt{right}}|$ if n is even and $|S_{\texttt{left}}| = |S_{\texttt{right}}| + 1$ if n is odd, and then construct two child nodes for the root, where the left child is labeled with $S_{\texttt{left}}$ and the right child is labeled with $S_{\texttt{right}}$. We recursively apply the above step to the left child and the right child, respectively, until every terminal node contains only one prefix family. At the end, we link all terminal nodes by a linked list. Figure 2 shows the PBtree for the set of prefix families $S = \{F(1), F(6), F(7), F(9), F(11), F(12), F(13), F(16), F(20), F(25)\}$.

The key property of PBtrees is stated in Theorem 3.1, which is straightforward to prove according to its construction algorithm. Note that the constraint $0 \leq |S_{\texttt{left}}| - |S_{\texttt{right}}| \leq 1$ makes the structure of the PBtree for a set of data items to solely depend on the number of data items.

THEOREM 3.1 (STRUCTURE INDISTINGUISHABILITY). *For any two sets of data items S_1 and S_2, their PBtrees have exactly the same structure if and only if $|S_1| = |S_2|$.*



F(1), F(6), F(7), F(9) ,F(11), F(12), F(13), F(16), F(20), F(25)
F(1), F(6), F(7), F(16), F(20)
F(9), F(11), F(12), F(13), F(25)
F(1), F(6), F(7)
F(16), F(20)
F(12), F(13), F(25)
F(9), F(11)
F(6),F(7)
F(12), F(13)
F(6)
F(7)
F(1)
F(20)
F(16)
F(12)
F(13)
F(25)
F(9)
F(11)

Figure 2: PBtree Example

We now describe the query processing algorithm on the above tree. For a PBtree T, we use $T.root$ to denote the root node of T, $T.left$ to denote the left subtree of T, and $T.right$ to denote the right subtree of T. For a node v, we use $L(v)$ to denote the label of v, which is a set of prefix families, and $U(v)$ to denote the union of all prefix families in $L(v)$. For example, if $L(v) = \{F(6), F(7)\}$, then $U(v) = F(6) \cup F(7)$. Given a range $[a, b]$, starting from the root $T.root$ of a PBtree T, where $L(T.root) = \{F(d_1), \cdots, F(d_n)\}$ and $U(T.root) = F(d_1) \cup \cdots \cup F(d_n)$, we check whether $U(T.root) \cap S([a, b]) = \emptyset$. If $U(T.root) \cap S([a, b]) = \emptyset$, then none of the n data items $d_1, \cdots, d_n$ falls into the range of $[a, b]$ and therefore we do not need to continue searching tree T. If $U(T.root) \cap S([a, b]) \neq \emptyset$, then there exists at least one of the n data items $d_1, \cdots, d_n$ falls into the range of $[a, b]$; thus, we need to continue to recursively conduct the same search on $T.left$ and $T.right$, if they exist. The pseudocode of the algorithm is in Algorithm 1.

Algorithm 1: PBtreeSearch(T, a, b)

```
  Input: $S$
  Output: PBtree nodes whose data item is in $[a, b]$
1 if $U(T.root) \cap S([a, b]) = \emptyset$ then
2     return null ;
3 else
4     if $T$ is leaf then
5         return L(T.root);
6     else
7         PBtreeSearch($T.left$, $[a, b]$);
          PBtreeSearch($T.right$, $[a, b]$);
```

Now, we analyze the time complexity of our query processing algorithm 1 and show that it is sub-linear in the number of data items. Let n be the number of data items indexed by the PBtree, $[a, b]$ be the query, and R be the query result. The average run-time of the search algorithm depends on $|R|$, the number of data items in the query result. Theoretically, if $|R| = 0$, then only the root of the PBtree needs to be checked and the time complexity is $O(1)$; if $|R| = n$, then all data items indexed by the PBtree need to be traversed via the linked list and the time complexity is $O(n)$. In reality, we have $|R| \ll n$ as n is typically large. For each data item in R, we need to traverse at most $2 \log n - 1$ nodes. Thus, the time complexity is $O(|R| \log n)$

3.3 Node Randomization Using Bloom Filters

Next, we present a solution based on secure keyed hash functions (HMAC) and Bloom filters to make our PBtree privacy preserving. For each node v, we use a Bloom filter denoted by $v.B$ to store the prefixes of a node's prefix families. We assume that the data owner and the users share

r secret keys, denoted $k_1, \cdots, k_r$, other than the symmetric key for encrypting and decrypting data items. Consider a PBtree node v, where set $L(v)$ consists of n prefix families and set $U(v)$ consists of m prefixes $p_1, \cdots, p_m$. Let w be the number of bits that each data item contains. Our node randomization algorithm consists of the following three steps.

One-wayness: For each prefix p_i, we use the r secret keys to compte r hashes: $\texttt{HMAC}(k_1, p_i), \cdots, \texttt{HMAC}(k_r, p_i)$. The purpose of this step is to achieve one-wayness, that is, given prefix p_i and the r secret keys, it is computationally efficient to compute the r hashes; but given the r hashes, it is computationally infeasible to compute the r secret keys and p_i; furthermore, even given the r hashes and p_i, which is the case in chosen plaintext attacks (CPA), it is still computationally infeasible to compute the r secret keys.

Decorrelation: For node v, we generate a random number $v.R$, which has the same number of bits as a secret key. We use $v.R$ to compute r hashes: $\texttt{HMAC}(v.R, \texttt{HMAC}(k_1, p_i))$, $\cdots$, $\texttt{HMAC}(v.R, \texttt{HMAC}(k_r, p_i))$. For each prefix p_i and for each $1 \le j \le r$, we let $v.B[\texttt{HMAC}(v.R, \texttt{HMAC}(k_j, p_i)) \ mod \ M] := 1$. The purpose of the random number that is unique for each node is to eliminate the correlation among different Bloom filters for different nodes. For the same prefix p, this random number allows us to hash p independently for different Bloom filters. Without the use of this random number, if prefix p_i is shared by $U(v_1)$ and $U(v_2)$ of two different nodes v_1 and v_2, then for all the r locations $\texttt{HMAC}(k_1, p_i) \ mod \ M$, $\cdots$, $\texttt{HMAC}(k_r, p_i) \ mod \ M$, both Bloom filters have the value 1. Although two Bloom filters both having 1 for all these r locations does not necessarily mean that $U(v_1)$ and $U(v_2)$ share a common prefix, without the use of this random number, if two Bloom filters have more 1s at the same locations than other pairs, then the probability that they share common prefixes is higher. Figure 3 shows the above hashing process for Bloom filters.

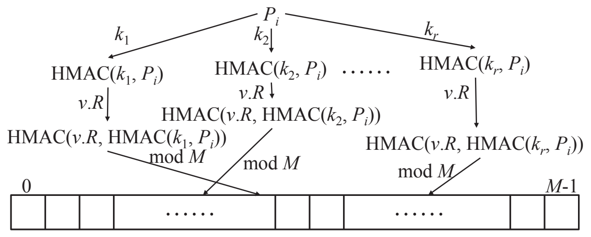


Figure 3: Secure Hashing in Bloom filters

Padding: If $m < (w+1) * n$, which means that some prefix families share common prefixes, we generate $((w+1)*n-m)*r$ random numbers and for each number x, $v.B[x \ mod \ M] := 1$. At last, we use this Bloom filter together with the random number $v.R$ to replace the label of v. The purpose of this step is to avoid a Bloom filter to expose the information how much its prefix families share common prefixes. Without the padding, some Bloom filters are inserted with less number of elements than others, which will cause it to have less 1s than others in the statistical sense.

By now the PBtree is fully constructed from data items $d_1, \cdots, d_n$ by the data owner. The data owner sends the encrypted data items and the PBtree to the cloud.

3.4 Trapdoor Computation

Given a query $[a, b]$, suppose $S([a, b])$ consists of z prefixes $p_1, \cdots, p_z$, for each prefix p_i, $1 \le i \le z$, the data user computes r hashes: $\texttt{HMAC}(k_1, p_i), \cdots, \texttt{HMAC}(k_r, p_i)$. The trapdoor for query $[a, b]$, denoted as $M_{[a,b]}$, is a matrix of $z * r$ hashes: $\texttt{HMAC}(k_1, p_1), \cdots, \texttt{HMAC}(k_r, p_1), \cdots, \texttt{HMAC}(k_1, p_z), \cdots, \texttt{HMAC}(k_r, p_z)$. We organize these $z * r$ hashes in a matrix because the cloud needs to know which r hashes are all correspond to the same prefix. The trapdoor of p_i corresponds to the ith row of the trapdoor matrix. After the computation, the data user sends $M_{[a,b]}$ to the cloud.

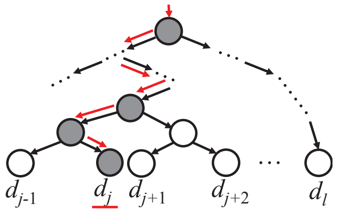


Figure 4: PBtree Example

3.5 Query Processing

After receiving a query represented as a trapdoor, the cloud uses the trapdoor to search over the PBtree. The query processing algorithm on PBtrees (*i.e.*, Algorithm 1) still applies except that the checking of whether $U(v) \cap S([a, b]) \ne \emptyset$ is implemented as checking whether there exists a row $i (1 \le i \le z)$ in matrix $M_{[a,b]}$ so that for every j $(1 \le j \le r)$ we have $v.B[\texttt{HMAC}(v.R, \texttt{HMAC}(k_j, p_i)) \ mod \ M] = 1$.

The straightforward implementation of the above query processing algorithms requires to check each row of $M_{[a,b]}$ at each visited PBTree node. Note that for a row i in $M_{[a,b]}$, if there exists j $(1 \le j \le r)$ so that $v.B[\texttt{HMAC}(v.R, \texttt{HMAC}(k_j, p_i)) \ mod \ M] = 0$, then $U(v) \cap p_i = \emptyset$. If $U(v) \cap p_i = \emptyset$, then for any descendent node v' of node v, we have $U(v') \cap p_i = \emptyset$ because $U(v') \subset U(v)$. Based on this fact, when we take $M_{[a,b]}$ to search over the PBtree, for any such row in $M_{[a,b]}$, we remove it from $M_{[a,b]}$ when we continue to search the descendent nodes of v. The searching process terminates when $M_{[a,b]}$ becomes empty or we finish searching terminal nodes.

3.6 False Positive Analysis

As each node in a PBtree is represented by a Bloom filter, which inherently has false positives, the query result on a PBtree may contain false positives. For simplicity, consider a PBtree with $n = 2^h$ leaf nodes, where the height of the PBtree is $h + 1$. Let R be the query result, which is a set of data items. We color all the terminal and nonterminal nodes on the path from a data item in R to the root of the PBtree to be grey and others to be white. Figure 4 shows such a marked PBtree where $d_j \in R$. Let f be the false positive rate of a Bloom filter in the PBtree. Note that although nodes of different levels in a PBtree may have a Bloom filter of different length, we always choose the number of hash functions r to be $\frac{m}{n} \times \ln 2$ to minimize the false positive rate to be $(1 - (1 - \frac{1}{m})^{rn})^r \approx (1 - e^{-rn/m})^r = 2^{-r} \approx 0.6185^{m/n}$; thus, by choosing the same m/n value for each node, the false positive of the Bloom filter at each node is the same. For any node $d_i \notin R$, let $\texttt{len}(d_i, R)$ be the number of white nodes on the path from d_i to the root, the probability that d_i is a false positive is $f^{\texttt{len}(d_i,R)}$. Thus, the expected number of false positives is $\Sigma_{d_i \notin R} f^{\texttt{len}(d_i,R)}$. Among all possible query result sets R of the same size a, we use M_a be denote the maximum expected number of false positives. Thus,

$$M_a = \max_{\forall R}\left(\sum_{d_i \notin R} f^{\texttt{len}(d_i,R)}\right) \tag{3.1}$$

For $a = 0$, we have

$$M_0 = 2^h \times p^{h+1} \quad (3.2)$$

For $a = 1$, say $d_j \in R$ as illustrated by Figure 4, the values of $\texttt{len}(d_i, R)$ for $d_i \notin R$ are $1, 2, 2, 3, 3, 3, 3, \cdots$. Thus, we have:

$$M_1 = f + 2f^2 + \cdots + 2^{h-1} f^h = f \frac{1-(2f)^h}{1-2f} \quad (3.3)$$

For $1 < a \leq n$, according to Equation 3.1, M_a corresponds to the case where in the $(\lceil \log a \rceil + 1)$-th layer there are a nodes colored grey and for each subtree rooted at these a nodes, there is one and only one terminal node is colored grey. Considering the $2^{\lceil \log a \rceil}$ subtrees rooted at the $(\lceil \log a \rceil + 1)$-th layer, the a subtrees have only one grey terminal node each and the rest $2^{\lceil \log a \rceil} - a$ subtrees have no grey terminal nodes. For each of the a subtrees, we can calculate the maximum expected number of false positives based on Equation 3.3; similarly, for each of the rest $2^{\lceil \log a \rceil} - a$ subtrees, we can calculate that based on Equation 3.2. Thus, M_a can be calculated as follows:

$$M_a = af \times \frac{1-(2f)^{h-\lceil \log a \rceil}}{1-2f} + (2^{\lceil \log a \rceil} - a) f (2f)^{h-\lceil \log a \rceil}$$

Figure 5 shows the relation between M_a and a, where we choose $f = 0.05$ and $h = 13$.

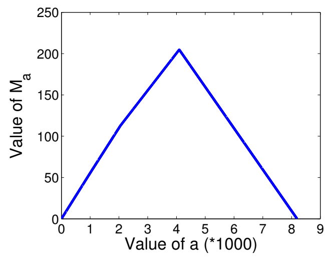


Figure 5: Relation between M_a and a

4. PBTREE SEARCH OPTIMIZATION

In this section, we optimize PBtree searching efficiency by minimizing the number of nodes that a query needs to traverse both horizontally and vertically.

4.1 Traversal Width Optimization

Recall that in the PBtree construction algorithm in Section 3.2, for a nonterminal node with prefix family set S, we partition this node into two child nodes S_1, S_2 so that $0 \leq |S_1| - |S_2| \leq 1$. This partition is critical for the performance of query processing on the PBtree because querying the common prefixes that both S_1 and S_2 share will lead to the traversal of both subtrees. Thus, in partitioning S into S_1, S_2, besides satisfying the condition $0 \leq |S_1| - |S_2| \leq 1$, we want to minimize $Max\{F_i \cap F_j | F_i \in S_1, F_j \in S_2\}$, which is the maximum number of prefixes in the intersection of two prefix families that one from S_1 and the other from S_2. This condition is to let those prefix families that share more prefixes to be partitioned in the same set. We call this problem *Equal Size Prefix Family Partition*. We next formally define this problem and prove that it is NP-hard.

Definition 4.1 (Equal Size Prefix Family Partition). *Given a set S of prefix families, we want to partition S into S_1, S_2, such that the following two conditions are satisfied:*

1. $0 \leq ||S_1| - |S_2|| \leq 1$;
2. $Max\{F_i \cap F_j | F_i \in S_1, F_j \in S_2\}$ *is minimized.*

Theorem 4.1. *The Equal Size Prefix Family Partition problem is NP-hard.*

Proof. The decision version of the Equal Size Prefix Family Partition Problem is the following: "Is it possible to partition a set S of prefix families into S_1 and S_2 such that $0 \leq ||S_1| - |S_2|| \leq 1$ and $Max\{F_i \cap F_j | F_i \in S_1, F_j \in S_2\} < k$?" We reduce the *Set Partition Problem*, a known NP-Complete problem, to the decision version of Equal Size Prefix Family Partition Problem. The Set Partition Problem is as follows: "For a multiset of positive numbers $A = \{a_1, a_2, \cdots, a_n\}$, is it possible to partition A into A_1 and A_2 such that $\sum_{a_i \in A_1} a_i = \sum_{a_j \in A_2} a_j$".

Given an instance of the set partition problem with positive number multiset $A = \{a_1, a_2, \cdots, a_n\}$, we convert it to an instance of our Equal Size Prefix Family Partition Problem with prefix family set S as follows. Let a_{max} be the largest number in A. For each number a_i in A, we first generate a_i data items $d_1, d_2, \cdots, d_{a_i}$ where each data item has $\lceil \log n \rceil + \lceil \log a_{max} \rceil$ bits, and for each data item d_j $(1 \leq j \leq a_i)$, the value of the first $\lceil \log n \rceil$ bits is i and the value of the last $\lceil \log a_{max} \rceil$ bits is $j - 1$. For example, suppose $A = \{2, 3, 4\}$, for number 2 in A, we generate 2 data items 0000 and 0001 in their binary representation. Second, for each data item d_j $(1 \leq j \leq a_i)$, we generate its prefix family $F(d_j)$. Finally, we map each number a_i in A to a_i prefix families $F(d_1), F(d_2), \cdots, F(d_{a_i})$ in S, and let $k = \lceil \log n \rceil$.

Suppose the prefix family set S constructed above has an equal size prefix family partition solution S_1 and S_2 with $Max\{F_i \cap F_j | F_i \in S_1, F_j \in S_2\} \leq k = \lceil \log n \rceil$. We next prove that A has a set partition solution. Note that if $\sum_{a_i \in A} a_i$ is odd, then the set partition problem has no solution. Thus, we only need to consider cases where $\sum_{a_i \in A} a_i$ is even, which means that $|S_1| = |S_2|$. A notable property of the a_i prefix families $F(d_1), F(d_2), \cdots, F(d_{a_i})$ constructed above is that any two of these prefix families share at least $\lceil \log n \rceil$ prefixes. For example, $F(0001)$ and $F(0001)$ share 3 prefixes. Thus, $Max\{F_i \cap F_j | F_i \in S_1, F_j \in S_2\} \leq k = \lceil \log n \rceil$ implies that for any a_i in A, the constructed a_i prefix families $F(d_1), F(d_2), \cdots, F(d_{a_i})$ are either all in S_1 or all in S_2. Otherwise, suppose $F(d_1) \in S_1$ and $F(d_2) \in S_2$, then $|F(d_1) \cap F(d_2)| \geq \lceil \log n \rceil = k$. Thus, $|S_1|$ is equal to the sum of some numbers in A and $|S_1|$ is equal to the sum of the remaining numbers in A. Finally, $|S_1| = |S_2|$ implies that A has a set partition solution. Thus, the Set Partition Problem $\leq_p$ the decision version of Equal Size Prefix Family Partition Problem, which means that the Equal Size Prefix Family Partition Problem is NP-hard. □

Next, we present our approximation algorithm to the equal size prefix family partition problem. Our algorithm consists of two phases: *partition phase* and *re-organization phase*. In the partition phase, we partition the input prefix family set into two or three subsets so that the size of each subset is no larger than $\lceil \frac{n}{2} \rceil$, where n is the size of the prefix family set. In the re-organization phase, if the first phase outputs two subsets, then we do nothing because the two subsets must satisfy the condition of $0 \leq ||S_1| - |S_2|| \leq 1$; if the first phase outputs three subsets, then we first choose one subset to split into multiple smaller subsets, and then merge these new subsets with the two other subsets to obtain two subsets that satisfy the condition of $0 \le ||S_1| - |S_2|| \le 1$.

To help to present the details of these two phases, we first define two concepts: *longest common prefix* and *child prefixes*. The longest common prefix of a set of prefix families $S = \{F_1, F_2, \cdots, F_n\}$, denoted by $LCP(S)$, is the longest prefix in $F_1 \cap F_2 \cap \cdots \cap F_n$. For example, the longest common prefix of $\{F(1101), F(1100)\}$ is $110*$. Note that for any set of prefix families, it has only one longest common prefix because no prefix family consists of two prefixes of the same length. For any prefix $b_1 b_2 \cdots b_{w-i+1} * \cdots *$, it has two child prefixes $b_1 b_2 \cdots b_{w-i+1} 0 * \cdots *$ and $b_1 b_2 \cdots b_{w-i+1} 1 * \cdots *$, which are obtained by replacing the first $*$ by 0 and 1 and called child-0 and child-1 prefixes, respectively. For example, prefix $11**$ has two child prefixes $110*$ and $111*$. For any prefix p, we use p^0 and p^1 to denote p's child-0 and child-1 prefixes, respectively.

Given a set of prefix families $S = \{F_1, F_2, \cdots, F_n\}$, the partition phase of our approximation algorithm works as follows. First, we compute $LCP(S)$, the longest common prefix of S. Second, we partition S into two subsets, one subset whose each prefix family contains $LCP(S)^0$ and one subset whose each prefix family contains $LCP(S)^1$. If any of the two subsets has a larger size than $\lceil \frac{n}{2} \rceil$, then we recursively apply the above two partition process to that subset. This process terminates when all subsets have a smaller size than $\lceil \frac{n}{2} \rceil$. Third, for any two subsets whose union has a size smaller than $\lceil \frac{n}{2} \rceil$, we call them *mergeable* and we merge them (*i.e.*, union them into one set). This process terminates when no two subsets are mergeable. Thus, we result in either two subsets or three subsets. It is impossible to result in four subsets or more because otherwise there are at least two subsets can be merged.

If the partition phase results in two subsets, then the reorganization phase does nothing. Let S_1 and S_2 be the two subsets. Because $|S_1| + |S_2| = n$, $|S_1| \le \lceil \frac{n}{2} \rceil$, and $|S_2| \le \lceil \frac{n}{2} \rceil$, we have $0 \le ||S_1| - |S_2|| \le 1$. Thus, S_1 and S_2 represent the final partition result.

If the partition phase results in three subsets, then the re-organization phase chooses one subset to split into multiple smaller subsets, and then union these new subsets with the two other subsets to obtain two subsets S_1 and S_2 that satisfy the condition of $0 \le ||S_1| - |S_2|| \le 1$. Let S_1, S_2, and S_3 be the three subsets. We choose the subset whose longest common prefix is the smallest, that is, the subset whose prefix families share the least number of prefixes. Let S_3 be the subset that we choose. We first compute its longest common prefix $LCP(S_3)$. Note that for any prefix family in S_3, it contains either $LCP(S_3)^0$ or $LCP(S_3)^1$. Second, we split S_3 into two subsets S_{31}, whose each prefix family contains $LCP(S_3)^0$, and S_{32}, whose each prefix family contains $LCP(S_3)^1$. Without loss of generality, we suppose $|S_{31}| \le |S_{32}|$. Thus, either S_1 or S_2 can be merged with S_{31}. Otherwise, if both $|S_1| + |S_{31}| > \lceil \frac{n}{2} \rceil$ and $|S_2| + |S_{31}| > \lceil \frac{n}{2} \rceil$, then $|S_1| + |S_2| + |S_3| = |S_1| + |S_2| + |S_{31}| + |S_{32}| \ge |S_1| + |S_2| + |S_{31}| + |S_{31}| = (|S_1| + |S_{31}|) + (|S_2| + |S_{31}|) > \lceil \frac{n}{2} \rceil + \lceil \frac{n}{2} \rceil \ge n$. Again, suppose $|S_1| \ge |S_2|$ and S_1 can be merged with S_{31}, we merge S_1 with S_{31}. After merging S_1 with S_{31}, we check whether S_2 can be merged with S_{32}. If they can, we merge them and output the partition result $S_1 \cup S_{31}$ and $S_2 \cup S_{32}$. If S_2 and S_{32} can not be merged, we further split S_{32}, and repeat the above process. If S_1 can not be merged with S_{31}, we merge S_{31} with S_2 and split S_{32}, and then repeat the above process. The pseudocode of this algorithm is shown in Algorithm 2.

We now analyze the worst case computational complexity of Algorithm 2. Let n be the size of the input set of prefix families and $T(n)$ be the corresponding time complexity. Each set partition operation takes $O(n)$ time and each subset merging takes $O(1)$ time. The worse case time complexity is when each set partition operation produces two subsets where the size of one subset is one. Thus, we have: $T(n) = T(n-1) + O(n)$. The computational complexity of Algorithm 2 is therefore $O(n^2)$ in the worst case.

Algorithm 2: EqualSizePrefixFamilyPartition(S)

```
Input: $S = \{F_1, F_2, \cdots, F_n\}$
Output: $S_1$ and $S_2$ where $S_1 \subset S$, $S_2 \subset S$, and
        $0 \le ||S_1| - |S_2|| \le 1$
1  Initiate an empty partition subset list $L$.
2  while ($|S| > \lceil \frac{n}{2} \rceil$) do
3      Compute $p_S$ for $S$. Parition $S$ into
       $S_1 = \{\forall F_i | LCP(S)^0 \in F_i, F_i \in S\}$, and
       $S_2 = \{\forall F_i | LCP(S)^1 \in F_i, F_i \in S\}$.
4      if ($|S_1| \ge |S_2|$) then
5          Inserte $S_2$ into $L$; $S := S_1$.
6      else
7          Insert $S_1$ into $L$; $S := S_2$.
8  Insert $S$ into $L$.
9  while (subsets $S_i$ and $S_j$ are mergable in $L$) do
10     Merge $S_i$ and $S_j$ into one subset $S_{ij}$. Replace $S_i$
       and $S_j$ with $S_{ij}$ in $L$.
11 if $L$ contains only two subsets $S_1$ and $S_2$ then
12     return $S_1$ and $S_2$.
13 else
14     Let $S_3$ be the subset that has $| \cap_{F_i \in S_3} | \le | \cap_{F_i \in S_1} |$,
       and $| \cap_{F_i \in S_3} | \le | \cap_{F_i \in S_2} |$ holds.
15     while $L$ has 3 subsets denoted by $S_1$, $S_2$, and $S_3$ do
16         Remove $S_3$ from $L$. Split $S_3$ into $S_{31}$, and $S_{32}$.
           Let $|S_1| \ge |S_2|$, and $|S_{31}| \le |S_{32}|$.
17         if ($|S_1| + |S_{31}| \le \lceil \frac{n}{2} \rceil$) then
18             Merge $S_{31}$ with $S_1$.
19             if ($|S_2| + |S_{32}| \le \lceil \frac{n}{2} \rceil$) then
20                 Merge $S_{32}$ with $S_2$.
21             else
22                 $S_3 := S_{32}$. Insert $S_3$ into $L$.
23         else
24             Merge $S_{31}$ with $S_2$.
25             $S_3 := S_{32}$. Insert $S_3$ into $L$.
26     return Labels of the two subsets in $L$.
```

4.2 Traversal Depth Optimization

Our idea for optimizing searching depth is based on the following observation: for any internal node v with label $\{F(d_1), F(d_2), \cdots, F(d_m)\}$ that a query prefix p traverses, if $p \in F(d_1) \cap F(d_2) \cap \cdots \cap F(d_m)$, then all terminal nodes of the subtree rooted at v satisfy the query; thus, we can directly jump to the left most terminal node of this subtree and collect all terminal nodes using the linked list, skipping the traversal of all nonterminal node under v in this subtree. This optimization opportunity is the motivation that we chain the terminal nodes in PBtrees. Note that here $F(d_1) \cap F(d_2) \cap \cdots \cap F(d_m) \neq \emptyset$ because it must contain the prefix of w *s. Furthermore, our searching width optimization technique significantly increases the probability that the prefix families in a nonterminal node share more than one common prefix.

For a node v labeled with $\{F(d_1), F(d_2), \cdots, F(d_m)\}$, we split $\bigcup_{i=1}^{m} F(d_i)$ into two sets: the *common set* $\mathbb{C} = \bigcap_{i=1}^{m} F(d_i)$ and the *uncommon set* $\mathbb{N} = \bigcup_{i=1}^{m} F(d_i) - \bigcap_{i=1}^{m} F(d_i)$. With this splitting, query processing at node v is modified to be the following. First, we check whether $p \in \mathbb{N}$. If $p \in \mathbb{N}$, then we continue to use the query processing algorithm in 3.5 to search p on v's left and right child nodes. If $p \notin \mathbb{N}$, then we further check $p \in \mathbb{C}$; if $p \notin \mathbb{N}$ but $p \in \mathbb{C}$, then we directly jump to the bottom to collect all terminal nodes in the subtree rooted at v; if p is in neither set, then we skip the subtree rooted at v.

The key technical challenge in searching depth optimization is how to store the common set $\mathbb{C}$ and the uncommon set $\mathbb{N}$ for each nonterminal node using Bloom filters. The straightforward solution is to use two Bloom filters, storing $\mathbb{C}$ and $\mathbb{N}$, respectively. However, this will not be space efficient as we need two bit vectors. In this paper, we propose an space efficient way to represent two Bloom filters using two sets of k hash functions $\{hc_1, hc_2, \cdots, hc_r\}$ and $\{hn_1, hn_2, \cdots, hn_r\}$ but only one bit vector B of m bits. In the PBtree construction phase, for a prefix $p \in \mathbb{C} \cup \mathbb{N}$, if $p \in \mathbb{C}$, we set $B[hc_1(p)], B[hc_2(p)], \cdots, B[hc_r(p)]$ to be 1; if $p \in \mathbb{N}$, we set $B[hn_1(p)], B[hn_2(p)], \cdots, B[hn_r(p)]$ to be 1. Thus, we check whether a $p \in \mathbb{C}$ by checking whether $\wedge_{i=1}^{r}(B[hc_i(p)] == 1)$ holds and check whether $p \in \mathbb{N}$ by checking whether $\wedge_{i=1}^{r}(B[hn_i(p)] == 1)$ holds.

Next, we analyze the false positives of this Bloom filter with two sets of r hash functions and a bit vector B of m bits. Suppose we have inserted n elements (*i.e.*, $|\mathbb{C}| + |\mathbb{N}| = n$) into this bloom filter. Recall that our query processing algorithm conducts two times of set membership testing of a query prefix p at node v: first, we test whether $p \in \mathbb{N}$; second, on the condition that $p \notin \mathbb{N}$, we test whether $p \in \mathbb{C}$. Let $f_{\mathbb{N}}$ be the probability of a false positive occurs at the first membership testing, and $f_{\mathbb{C}}$ be the probability of a false positive occurs at the second membership testing. As the n elements are randomly and independently inserted into the bit vector B, the false positive probability at the first membership testing is the same as the false positive probability of the standard Bloom filter. Thus, we have

$$f_{\mathbb{N}} = (1 - (1 - \frac{1}{m})^{rn})^r = (1 - e^{-\frac{rn}{m}})^r \tag{4.1}$$

As the second testing is only performed on the condition that $p \notin \mathbb{N}$, and similarly, when the condition $p \notin \mathbb{N}$ holds, the false positive probability at testing whether $p \in \mathbb{C}$ is the same as that at testing whether $p \in \mathbb{N}$, we have

$$f_{\mathbb{C}} = (1 - f_{\mathbb{N}}) \times (1 - (1 - \frac{1}{m})^{rn})^r = (1 - (1 - e^{-\frac{rn}{m}})^r) \times (1 - e^{-\frac{rn}{m}})^r \tag{4.2}$$

To further reduce the false positive probability in testing $p \in \mathbb{C}$ at node v, when we collect the leaves of the subtree rooted at v, we can randomly choose x leaf nodes to test whether they indeed match p; if any of the leaf nodes does not match p, which means that $p \notin \mathbb{C}$, then we exclude all leaves of the subtree rooted at v from the query result. Thus, with the testing of the x leaf nodes, the false positive probability in testing whether $p \in \mathbb{C}$ becomes the following:

$$f_{\mathbb{C}} \times (1 - e^{-\frac{rn}{m}})^{rx} = (1 - (1 - e^{-\frac{rn}{m}})^r) \times (1 - e^{-\frac{rn}{m}})^{rx+r} \tag{4.3}$$

Note that we test $p \in \mathbb{N}$ first and only when $p \notin \mathbb{N}$ we test $p \in \mathbb{C}$. Otherwise, if we use the above mentioned leaf testing method to further reduce the false positive probability in testing $p \in \mathbb{C}$, we may introduce false negatives, which is not allowed in our scheme. Suppose we first test $p \in \mathbb{C}$ first and only when $p \notin \mathbb{C}$ we test $p \in \mathbb{N}$. For a query prefix p, if $p \in \mathbb{N}$ and $p \notin \mathbb{C}$, but false positive occurs in testing $p \notin \mathbb{C}$, when we collect the leaves of the subtree rooted at v, if we test a leaf node and find it is not in $\mathbb{C}$, according to the above leaf testing method, we exclude all leaves of the subtree rooted at v from the query result; however, as $p \in \mathbb{N}$, some of these excluded leaves should be included in the query result, which are false negatives.

5. SECURITY ANALYSIS

5.1 Security Model

To achieve IND-CKA security, our PBtree uses pseudo-random functions, which are *keyed* functions and cannot be distinguished from a truly random function with non-negligible probability [29]. Let $g : \{0,1\}^n \times \{0,1\}^s \to \{0,1\}^m$ be a keyed function which takes as input n-bit strings and s-bit keys and maps to m-bit strings. Let $G : \{0,1\}^n \to \{0,1\}^m$ be a random function which maps n-bit strings to m-bit strings. Now, g is a pseudo-random function if, for a fixed value $k \in \{0,1\}^s$, the function $g(x,k)$, where $x \in \{0,1\}^n$, can be computed efficiently and if a probabilistic polynomial time adversary $\mathcal{A}$ with access to r chosen evaluations of g, *i.e.*, $(x_i, g(x_i,k))$ where $i \in [1,r]$, cannot distinguish the value $g(x_{r+1},k)$ from the output of a random function G with non-negligible probability. We have used `HMAC` for our scheme as the pseudo-random function. From the results in [17, 29], a searchable symmetric encryption scheme is secure if a probabilistic polynomial time adversary cannot distinguish between the output of a real index, which uses pseudo-random functions, and a simulated index, which uses random functions, with non-negligible probability. We show the construction of such a simulator for our scheme and thereby, prove its security under the IND-CKA model.

5.2 Security Proof

Without loss of generality, we view the PBtree as a list of Bloom filters, where each Bloom filter stores a distinct set of prefixes and answers user queries. Therefore, we note that, that the proof of security of PBtree is equivalent to proving that any given Bloom filter is IND-CKA secure satisfying the following properties: (a) the contents of the data items are not revealed from the structure of the Bloom filter in which they are stored or from the contents of other Bloom filters, and (b) given any two Bloom filters, with different number of data items, they are indistinguishable to an adversary. We consider a *non-adaptive adversary*, which has a one-time finite trace of the search history consisting of a finite set of secure trapdoors and their corresponding search results. To complete the proof, we demonstrate the construction of a probabilistic polynomial time simulator $\mathcal{S}$, which can simulate the secure index using only this finite trace of the search history. The adversary interacts with the simulator as well as the real index and is challenged to distinguish between the results of the two indexes with non-negligible probability. We consider the length of the key s as the security parameter in the following definitions:

History: H_q. Let $\mathbf{D} = \{D_1, D_2, \cdots, D_n\}$ denote the set of data items where D_i denotes the i^{th} data item. Let $\mathbf{R_{1:q}} = \{R_1, R_2, \cdots, R_q\}$, denote a sequence of q range queries where each range query is of the form, $R_i = \{a_i, b_i\}$ for $a_i, b_i, q \in \mathbb{N}$. The history H_q is defined as $H_q = \{\mathbf{D}, \mathbf{R_{1:q}}\}$, where the set $\mathbf{D}$ consists of data items, which match one or more of the range queries in $\mathbf{R_{1:q}}$. An important requirement is that q must be polynomial in the security parameter s, the key size, in order for the adversary to be polynomially bounded.

Adversary View: A_v. This is the view of the adversary of a history H_q. Each range query, $R_i = \{a_i, b_i\}$ generates a set of r_i trapdoors, $T_i = \{t_{i,1}, t_{i,2}, \cdots, t_{i,r_i}\}$ which are secure under an encryption key K. Now, the adversary view consists of: the set of trapdoors corresponding to the range queries $\mathbf{T}$, the secure index $\mathcal{I}$ for $\mathbf{D}$ and, set of the encrypted data items, $Enc_K(D)$=$\{Enc_K(D_1)$, $Enc_K(D_2)$, $\cdots$, $Enc_K(D_n)\}$. Formally, $A_v(H_q) = \{\mathbf{T}; \mathcal{I}; Enc_K(D)\}$. In addition, the adversary may also know the size of the encrypted data items.

Result Trace This is defined as the *access* and *search* patterns observed by the adversary after $\mathbf{T}$ is matched against the encrypted index $\mathcal{I}$. The access pattern is the set of matching data items, $M(T)$=$\{$ $m(t_1)$, $m(t_2)$ $\cdots$, $m(t_q)\}$, where $m(t_i)$ denotes the set of matching data item identifiers for trapdoor t_i. The search pattern is a symmetric binary matrix Π_T defined over T, such that, $\Pi_T[p, q] = 1$ if $t_p = t_q$, for, $1 \le p, q \le \sigma^{|T_i|}$. We denote the matching result trace over H_q as: $M_{(H_q)} = \{M(R_{1:q}), \Pi_T[p, q]\}$. The adversary can only see a set of matching data items for each trapdoor, which is captured using these two patterns. Therefore, each Bloom filter can be viewed as a match for a distinct, but not necessarily unique when viewed along the PBtree, set of trapdoors. The uniqueness is not possible since each range query can match multiple distinct trapdoors.

Theorem 1. The PBtree scheme is IND-CKA secure under the pseudo-random function f and the encryption algorithm *Enc*.

Proof. We consider a sample adversary view $A_v(H_q)$ and show that, given a real matching result trace $M_{(H_q)}$, it is possible to construct a polynomial time simulator $\mathcal{S} = \{S_0, S_q\}$ that can simulate this view with non-negligible probability. We denote the simulated adversary view as $A_v^*(H_q)$, the simulated index as $\mathcal{I}^*$, the simulated encrypted documents as $Enc_K(D^*)$, and the trapdoors as $\mathbf{T}^*$. By definition, each Bloom filter matches a distinct set of trapdoors, which are visible in the result trace of the query. Let ID_j denote the unique identifier of a Bloom filter. The final result of the simulator is to output trapdoors based on the chosen range query history submitted by the adversary; given that the adversary is not allowed to see the index or the trapdoors prior to submitting the history.

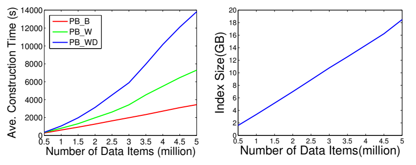
Step 1. Index Simulation To simulate the index $\mathcal{I}^*$, given that the size and number of Bloom filters is known from $\mathcal{I}$, we generate same sized bit-arrays, B^*, where random bits are set to 1 while ensuring that each Bloom filter at the same layer has approximately equal number of bits. Next, we generate random $Enc_K(D^*)$, such that each simulated data item has same size as an original encrypted data item in $Enc_K(D)$ and $|Enc_K(D^*)| = |Enc_K(D)|$.

Now, for the first Bloom filter, which represents the root of PBtree, in $\mathcal{I}^*$, we associate the entire set $Enc_K(D^*)$, *i.e.*, this filter points to the entire data item set. For the next two Bloom filters, corresponding to the child sub-sets, we consider each data item and probabilistically, where $Prob[Assign] = 1/2$ with a fair coin toss, and assign it to one of the Bloom filters. Finally, if there are any left over data items, we randomly assign them to the filters such that both the child sub-sets differ by at most one item.

Step 2. Simulator State S_0 For H_q, where $q = 0$, we denote the simulator state by S_0. We construct the adversary view as follows: $A_v^*(H_0) = \{Enc_K(D^*), \mathcal{I}^*, T^*\}$, where T^* denotes the set of trapdoors. To generate T^*, we first note that, each data item in $Enc_K(D^*)$ corresponds to a set of matching trapdoors. The length of each trapdoor is given by the pseudo-random function g, and the maximum possible size of trapdoors matching the data item is given by the length of the prefix family of the data item: $n + 1$ where, n is the length of the data items. Therefore, we generate $(n+1) * |Enc_K(D^*)|$ random trapdoors of length $|g(.)|$ each and, with a uniform probability defined over these trapdoors, we associate at most $n + 1$ trapdoors for each data item in $Enc_K(D^*)$. Note that, some trapdoors might repeat, which is desirable as two data items might match the same trapdoor. The distribution of trapdoors is now straightforward; for each Bloom filter in $\mathcal{I}^*$, we consider the data items and perform a union of the trapdoors. This distribution is consistent with the trapdoor distribution in the original index $\mathcal{I}$, *i.e.*, this simulated index satisfies all the structural properties of a real PBtree index. Now, given that g is pseudo-random, and the probability of trapdoor distribution is consistent, this distribution is indistinguishable by any probabilistic polynomial time adversary.

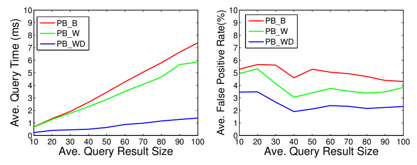
Step 3. Simulator State S_q For H_q where $q \ge 1$, we denote the simulator state by S_q. The simulator constructs the adversary view as follows: $A_v^*(H_q) = \{Enc_K(D^*), \mathcal{I}^*, T^*, T_q\}$ where T_q are trapdoors corresponding to the query trace. To construct, I^*, given $M(R_{1:q})$, consider the set of matching data items for each trapdoor, $M(T_i)$=$\{m(t_{i,1})$, $m(t_{i,2})$, $\cdots$, $m(t_{i,r_i})$ $\}$ where $1 \le i \le q$. Let $M(R_{1:q})$ contain p unique data items. For each data item in the trace, $Enc_K(D_p)$, the simulator associates the corresponding trapdoor from $M(T_i)$ and if more than one trapdoor matches the data item, then the simulator generates a union of the trapdoors. Since $p < |\mathbf{D}|$, the simulator generates $1 \le i \le |\mathbf{D}| - q + 1$ random strings, $Enc_K^*(D_i)$ of size $|Enc_K(D)|$ each and associates up to $n + 1$ trapdoors uniformly, as done in Step 2, ensuring that these strings do not match any strings from $M(T_i)$. The simulator maintains an auxiliary state $\mathcal{ST}_q$ to remember the association between the trapdoors and the matching data items. Now, for the first Bloom filter, we map all the data item identifiers : $Enc_K(D^*) = M(R_{1:q}) \cup Enc_K^*(D_i)$ where $1 \le i \le |\mathbf{D} - q| + 1$. The child Bloom filters are constructed in a similar manner as before. The simulator outputs: $\{Enc_K(D^*), \mathcal{I}^*, T^*, T_q\}$. Note that, all the steps performed by the simulator are polynomial and hence, the simulator runs in polynomial time complexity.

Now, if a probabilistic polynomial time adversary issues a range query over any data item matching the set $M(R_{1:q})$, the simulator gives the correct trapdoors. For any other data item, the trapdoors given by simulator are indistinguishable due to pseudo-random function g. Finally, since each Bloom filter contains sufficient blinding, our scheme is proven secure under the IND-CKA model. $\square$



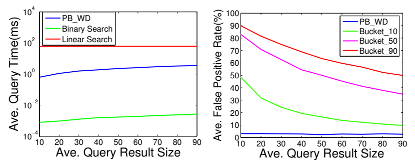
(a) Con. Time (b) Index Size

Figure 6: Time & Size



(a) Query Time (b) FP Rate

Figure 7: Optimization Evaluation



(a) Query Time (b) FP Rate

Figure 8: PBtree Evaluation

6. EXPERIMENTAL EVALUATION

6.1 Experimental Methodology

To evaluate the performance of PBtree, we considered three factors and generated the various experimental configurations. The metrics considered are: the data sets, the type of PBtree construction, and the type of the queries. Based on these metrics, we have comprehensively evaluated the construction cost of the PBtree, the query evaluation time and the observed false positive rates.

6.1.1 Data Sets

We chose the Gowalla [20] data set, which consists of 6,442,890 check-in records of users, over the period of Feb. 2009 to Oct. 2010, and extracted the time stamps. Now, given that each time stamp is represented as a tuple : $\langle year, month, date, hour, minute, second \rangle$, we performed a binary encoding for each of these attributes and treated the concatenation of the respective binary strings as a 32-bit integer value, while ignoring the unused bit positions. We perform our experiments on 10 fixed size data sets varying from 0.5 to 5 million records with a scaling factor of 0.5 million records, respectively, chosen uniformly at random from the 6 million-plus total records in the Gowalla data set.

6.1.2 PBtree Types

We performed experiments with three variants of the PBtree: the basic PBtree without any optimizations, denoted as PB_B, the PBtree with width optimization, denoted as PB_W, and the PBtree with both depth and width optimizations, denoted as PB_WD. We have not performed experiments for the case of the PBtree with only depth optimization due to the following reasoning: when searching on a Bloom filter we may need to perform two checks, which is twice the effort. If a query prefix is not found in the Bloom filter, then we need to perform a second check, using a different set of hash functions, to check if the prefix is a common prefix in the Bloom filter. As a result, depth optimization is more effective when combined with width optimization because width optimization aggregates the common prefixes in a systematic manner. Therefore, we focus only on the performance evaluation of PB_B, PB_W and PB_WD.

6.1.3 Query Types

The performance evaluation of PBtree is dependent on two factors: query types and query results size. We consider two query types: prefix and range queries. A prefix query is a query specified as a single binary prefix , whereas, a range query is specified as a numerical range and is likely to generate more than one binary prefixes. The prefix queries are effective in evaluating the performance of PBtree under the two types of optimizations we have described, and the range queries are effective to evaluate the performance of PBtree against other known approaches in literature. For each data set, we generate a distinct collection of 10 prefix query sets, where each prefix set contains 1000 prefixes, and similarly, we generate 10 distinct range query sets, where each set contains 1000 range queries. The average number of prefixes for denoting a range in our range query sets vary from 5.93 to 9.6 prefixes, respectively.

The query result size is another important factor since the worst-case run-time search complexity of PBtree is given by $O(r. \log N)$ where r is the query result size. But the challenge is that, since the data values are not in any particular sequence, it is difficult to know which range queries can generate the desired query result sizes after the PBtree is built. To handle this issue, prior to the PBtree construction, we sort the data items and determine the appropriate range queries, which will result in the desired query result sizes and use these queries in our experiments. For our experiments, we chose query ranges which result in query result sizes varying from 10 to 90 data items.

6.1.4 Implementation Details

We conducted our experiments on desktop PC running Windows 7 Professional with *32GB* memory and *3.5GHz Intel(R) Core(TM) i7-4770k* processor. We used $HMAC-SHA1$ as the pseudo-random function for the Bloom filter encoding and implemented the PBtree using C++. We set the Bloom filter parameter, $m/n = 10$, where m is the Bloom filter size and n is the number of elements, and the number of Bloom filter hash functions as 7. Although we have also experimented with other values of m/n, because of the limited space, we only show the results for $m/n = 10$.

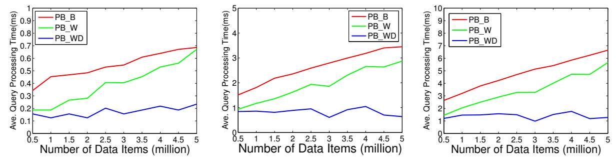
6.2 Evaluation of PBtree Construction

Our experimental results show that, the cost of PBtree construction is reasonable, both in terms of time and space. For the chosen datasets, Figure 6(a) shows that, the average time for generating, the PB_B is 276 to 3443 seconds, the PB_W is 338 to 7500 seconds, and the PB_WD is 357 to 14027 seconds, respectively. The average time required for the PB_WD construction is higher due to the equal-size partition algorithm and common prefix computation overhead involved. However, as we show later, the query processing time for PB_WD is smaller compared to the other two variants of the PBtree, and the false positive rate is lower as well. Figure 6(b) shows that, the PBtree sizes range from 1.598GB to 18.494GB for the data sets, and also, for a specific data set size, the PB_B, PB_W, and PB_WD index structures are of the same size, respectively. Finally, the PBtree construction incurs a one-time off-line construction overhead.

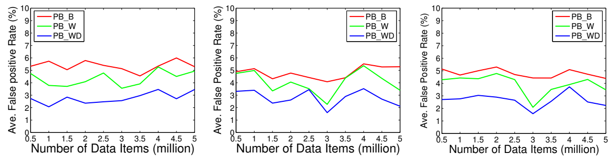
6.3 Query Evaluation Performance

6.3.1 Prefix Query Evaluation

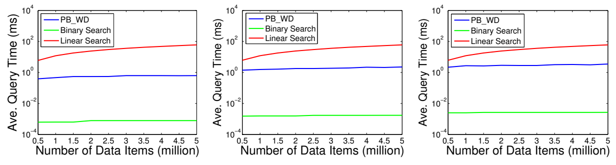
Our experimental results show that, the width and the depth optimizations are highly effective in reducing the query



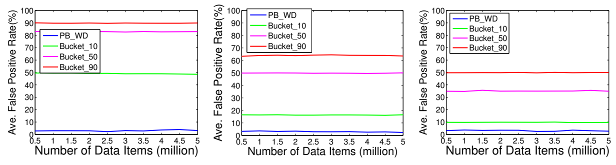
(a) R. size = 10 (b) R. Size =50 (c) R. Size =90
Figure 9: Ave. Query Time of Prefix Queries



(a) R. Size=10 (b) R. Size =50 (c) R. Size =90
Figure 10: Ave. False Positive Rate of Prefix Queries



(a) R. size = 10 (b) R. Size =50 (c) R. Size =90
Figure 11: Ave. Query Time of Range Queries



(a) R. Size=10 (b) R. Size =50 (c) R. Size =90
Figure 12: Ave. False Positive Rate of Range Queries

processing time and the false positive rates. We denote the average query result size as "R.Size" in the figures. Figure 9 and Figure 10 show the average prefix query processing times and false positive rates, respectively, on different data sets, for prefix queries issued on the corresponding PB_B, PB_W and PB_WD structures. Figure 7(a) and Figure 7(b) show the average query processing time and false positive rates, respectively, for different prefix query result sizes, on the PB_B, PB_W and PB_WD structures, which are built on 5 million data items.

The PB_WD structure exhibits higher query processing efficiency and records lower false positive among all PBtree structures. From the figures, we note that, for the same the query result sizes, PB_WD executes, 2.153, 2.309, and 2.533 times faster than PB_B, respectively; and the corresponding false positive rates in PB_WD are, 0.88, 0.8, and 0.83 times, smaller than in PB_B. In comparison, for the same query result sizes, PB_W executes 0.516, 0.406, and 0.444 times faster than PB_B, respectively; and the corresponding false positive rates in PB_W are, 0.25, 0.186, and 0.21 times, smaller than PB_B.

6.3.2 Range Query Evaluation

Compared with existing schemes, our experimental results show that PB_WD has smaller range query processing times and lower false positive rates. We compared the speed of PB_WD with two plain text schemes: *linear search*, in which we examine each item from the unsorted data set to match the range query, and *binary search*, in which we execute the range query over the sorted data using the binary search algorithm. To evaluate the accuracy of PB_WD, we compared the recorded false positive rates with those observed in the *bucket* scheme of [26]. In our experiments, both the data items and the queries follow uniform distribution and hence, each bucket contains same number of data items with bucket sizes ranging from 10 to 90.

Figure 11 and Figure 12 show the average range query processing time and the false positive rates, respectively for different query result sizes on the experimental data sets. We observed that, for the three query result sizes, the plan-text binary search is, respectively, 116, 113, and 110 times, faster than the corresponding search results on the PB_WD structure. On the other hand, PB_WD performs, 14.8, 3.3, and 1.748 times, faster query processing than the linear search scheme. Note, we use logarithmic coordinates in Figure 11 and Figure 8(a).

In terms of accuracy PB_WD outperforms the bucket scheme [26] by orders of magnitude. For instance, for the maximum query result size of 90 in our experiments, the false positive rates recorded by PB_WD are, 2.12, 21.38, and 39.96 times lesser than the bucket scheme with respective bucket sizes being 10, 50, and 90.

Finally, Figure 8(a) and Figure 8(b) show the average range query processing time and false positive rates, respectively, for different query result sizes, where the corresponding indexes, PB_WD, linear, binary and bucket, are built on a data set of 5 million data items.

7. CONCLUSIONS

In this paper, we propose the first range query processing scheme that achieves index indistinguishability, under the IND-CKA [19], which provides strong privacy guarantees. The key novelty of this paper is in proposing the PBtree data structure and associate algorithms for PBtree construction, searching, and optimization. We implemented and evaluated our scheme on a real world data set. The experimental results show that our scheme can efficiently support real time range queries with strong privacy protection.

8. ACKNOWLEDGEMENTS

This work is supported in part by the National Natural Science Foundation of China under Grant Numbers 61370226, 61472184, 61321491, 61272546, Young Teacher Growth Plan of Hunan University, China Postdoctoral Science Foundation, and the National Science Foundation under Grant Numbers CNS-1017598. We would like to thank our anonymous reviewers for their valuable suggestions and feedback for improving our paper significantly. We also like to thank Mustafa Canim and Murat Kantarcioglu for providing us the source code of their work in [26].

9. REFERENCES

[1] Amazon web services, aws.amazon.com.
[2] Google app engine, code.google.com/appengine.
[3] Microsoft azure, www.microsoft.com/azure.
[4] Google fires engineer for privacy breach. http://www.cnet.com/news/google fired engineer for privacy breach/, 2010.

[5] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *Proceedings of the ACM SIGMOD*, pages 563–574. ACM, 2004.

[6] R. Agrawal and R. Srikant. Privacy-preserving data mining. In *Proceedings of the ACM SIGMOD*, pages 439–450. ACM, 2000.

[7] L. Ballard, S. Kamara, and F. Monrose. Achieving efficient conjunctive keyword searches over encrypted data. In *Information and Communications Security*, pages 414–426. 2005.

[8] M. Bellare, A. Boldyreva, and A. ONeill. Deterministic and efficiently searchable encryption. In *Proceedings of the CRYPTO*, 2007.

[9] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In *EUROCRYPT*, pages 224–241, 2009.

[10] A. Boldyreva, N. Chenette, and A. O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *CRYPTO*, 2011.

[11] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *EUROCRYPT*, 2004.

[12] X. Boyen and B. Waters. Anonymous hierarchical identity-based encryption (without random oracles). In *CRYPTO*, 2006.

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

[14] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou. Privacy-preserving multi-keyword ranked search over encrypted cloud data. In *IEEE INFOCOM*, 2011.

[15] Y.-C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Third International Conference on Applied Cryptography and Network Security (ACNS)*, 2005.

[16] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM CCS*, 2006.

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

[18] E. Damiani, S. C. Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing confidentiality and efficiency in untrusted relational dbmss. In *CCS*, pages 93–102, 2003.

[19] Eujin-Goh. Secure indexes. Stanford University Technical Report, 2004.

[20] S. A. Eunjoon Cho and J. Leskovec. Friendship and mobility: User movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining(KDD)*, pages 1082–1090. ACM, 2011.

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

[22] P. Golle, J. Staddon, and B. Waters. Secure conjunctive keyword search over encrypted data. In *Applied Cryptography and Network Security*. 2004.

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

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

[25] 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.

[26] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *VLDB*, pages 720–731, 2004.

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

[28] M. Kantarcioglu and C. Clifton. Security issues in querying encrypted data. In *DBSec*, 2005.

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

[30] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Authenticated index structures for outsourced databases. In *Handbook of Database Security*, pages 115–136. 2008.

[31] J. Li and E. R. Omiecinski. Efficiency and security trade-off in supporting range queries on encrypted databases. In *19th DBSec*, 2005.

[32] N. Li, T. Li, and S. Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 106–115, April 2007.

[33] A. X. Liu and F. Chen. Collaborative enforcement of firewall policies in virtual private networks. In *Proc. ACM PODC*, 2008.

[34] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. *Trans. Storage*, 2(2):107–138, May 2006.

[35] S. R. M. Oliveira and O. R. Zaïane. Privacy preserving clustering by data transformation. *JIDM*, 1(1):37–52, 2010.

[36] D. Park, K. Kim, and P. Lee. Public key encryption with conjunctive field keyword search. In *Information Security Applications*, pages 73–86. 2005.

[37] A. Shamir. Identity-based cryptosystems and signature schemes. In *CRYPTO 84*, pages 47–53, 1985.

[38] E. Shi, J. Bethencourt, T.-H. H. Chan, D. Song, and A. Perrig. Multi-dimensional range query over encrypted data. In *IEEE S&P Symposium*, 2007.

[39] D. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *IEEE S&P Symposium*, 2000.

[40] L. Sweeney. K-anonymity: A model for protecting privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10(5):557–570, Oct. 2002.

[41] W. K. Wong, D. W.-L. Cheung, B. Kao, and N. Mamoulis. Secure knn computation on encrypted databases. In *SIGMOD*, pages 139–152, July 2009.

[42] Q. Zheng, S. Xu, and G. Ateniese. Efficient query integrity for outsourced dynamic databases. In *Proceedings of the 2012 ACM Workshop on Cloud Computing Security Workshop*, CCSW '12, 2012.