7-2012

Enhancing Access Privacy of Range Retrievals over B+Trees

Hwee Hwa PANG
Singapore Management University, hhpang@smu.edu.sg

Jilian ZHANG
Singapore Management University, jilian.z.2007@smu.edu.sg

Kyriakos MOURATIDIS
Singapore Management University, kyriakos@smu.edu.sg

Citation

PANG, Hwee Hwa; ZHANG, Jilian; and MOURATIDIS, Kyriakos. Enhancing Access Privacy of Range Retrievals over B+Trees. (2012). *IEEE Transactions on Knowledge and Data Engineering*. 25, (7), 1533-1547.

Available at: https://ink.library.smu.edu.sg/sis_research/1512

Enhancing Access Privacy of Range Retrievals over B+-Trees

HweeHwa Pang, Jilian Zhang and Kyriakos Mouratidis

***Abstract*—Users of databases that are hosted on shared servers cannot take for granted that their queries will not be disclosed to unauthorized parties. Even if the database is encrypted, an adversary who is monitoring the I/O activity on the server may still be able to infer some information about a user query. For the particular case of a B$^+$-tree that has its nodes encrypted, we identify properties that enable the ordering among the leaf nodes to be deduced. These properties allow us to construct adversarial algorithms to recover the B$^+$-tree structure from the I/O traces generated by range queries. Combining this structure with knowledge of the key distribution (or the plaintext database itself), the adversary can infer the selection range of user queries.**

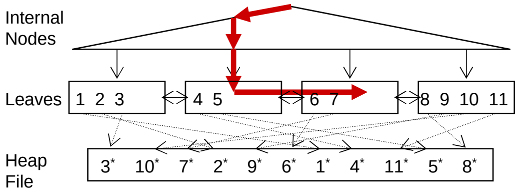
To counter the threat, we propose a privacy-enhancing PB$^+$-tree index which ensures that there is high uncertainty about what data the user has worked on, even to a knowledgeable adversary who has observed numerous query executions. The core idea in PB$^+$-tree is to conceal the order of the leaf nodes in an encrypted B$^+$-tree. In particular, it groups the nodes of the tree into buckets, and employs homomorphic encryption techniques to prevent the adversary from pinpointing the exact nodes retrieved by range queries. PB$^+$-tree can be tuned to balance its privacy strength with the computational and I/O overheads incurred. Moreover, it can be adapted to protect access privacy in cases where the attacker additionally knows a priori the access frequencies of key values. Experiments demonstrate that PB$^+$-tree effectively impairs the adversary's ability to recover the B$^+$-tree structure and deduce the query ranges in all considered scenarios.

***Index Terms*—Access privacy, range retrieval, B$^+$-tree.**

I. Introduction

The outsourcing model [1] offers enhanced data availability and disaster protection, but raises severe concerns about the privacy of data and users. Although the data can be encrypted to disallow unauthorized access, encryption does not prevent the service provider from monitoring the I/O activities of user queries, thus inferring (and potentially misusing) sensitive information of corporate or personal importance. Similar concerns arise in a wide range of shared computing environments, including common enterprise data servers administered by curious DBAs [2].

To mitigate the concerns, we aim to support efficient database querying in such an environment, while offering a high degree of protection for the privacy of user queries from the database server (and hence from the untrusted service provider or any curious individual with access to it). Ideally, this means that after observing any number of queries, the adversary should gain no information on what data were retrieved. Cryptographic techniques that achieve such privacy safeguards include Private Information Retrieval (PIR) [3] and Oblivious RAM (ORAM) [4], [5]. However, both PIR and ORAM are known to impose very heavy computation and communication overheads.



Known key distribution: Uniform between 1 and 11

Fig. 1. Inferring Key Values in an Encrypted B$^+$-tree

In this work, we aim for a weaker, yet practical, security objective: We ensure that there is *high uncertainty* about what data the user has worked on, even to a *knowledgeable adversary* who has observed numerous query executions at the server. Drawing from [6], we consider two classes of such adversaries – the first possesses the encrypted database DB^E and knowledge of the data value distribution Dist; the other has copies of both the encrypted database DB^E and the plaintext database DB itself. Assuming that the adversary is unable to decipher DB^E directly, the protection offered to the user hinges on the difficulty in deducing the mapping between DB^E and Dist/DB.

We focus on privacy protection for range retrievals over large datasets indexed by B$^+$-trees [7]. In [6], Damiani et al. proposed to build a B$^+$-tree on the search key, and encrypt its nodes (including their child pointers). As the tree structure is not visible to the server, tree traversal has to involve the user; specifically, the user has to decrypt the node(s) in the current level to determine the child node(s) to visit next, starting from the root. In this paper, we show that Damiani's encrypted B$^+$-tree can be defeated if the adversary monitors the data I/Os on the server; by tracking the sequence of nodes retrieved during range selection operations, the adversary can, over time, infer the position of each node within the B$^+$-tree. The exposed tree structure, combined with knowledge of the data distribution or the plaintext data, allows the adversary to deduce easily the key range in any user query.

To illustrate, consider the encrypted B$^+$-tree in Figure 1. The disk blocks storing the internal and leaf nodes of the B$^+$-tree are encrypted. Furthermore, the records in the underlying heap file are encrypted individually, as signified by the asterisk following each record's key value. As the

HweeHwa Pang, Jilian Zhang and Kyriakos Mouratidis are with the School of Information Systems, Singapore Management University, 80 Stamford Road, Singapore 178902.
Email: {hhpang, jilian.z.2007, kyriakos}@smu.edu.sg

bold red arrows indicate, the traversal path of a range selection starts from the root node, goes down to the leaf containing the left bound of the key range, then follows the right sibling pointers, and culminates in the leaf node containing the right bound of the range. The sequence of node accesses is visible to the adversary.

Knowing how many records are in the underlying relation (from the metadata or the size of the heap file), the adversary can deduce the B$^+$-tree height, and discount the internal nodes at the front of the access sequence. The remaining sequence gives away the ordering among the accessed leaf nodes. After a sufficient number of range queries, the adversary will eventually be able to sequence all the leaf nodes. Moreover, the number of records retrieved from the heap file following a leaf node reveals the number of key values within the node. With knowledge of the key distribution and the total number of records, the adversary can now deduce with high confidence the key values covered by each leaf and, hence, the selection range of the queries. For example, with uniform key distribution between 1 and 11, the first leaf with 3 of the 11 entries is likely to cover key values 1 to 3.

Contributions: To study the above threat systematically, we propose a formalization for it, along with a quantitative privacy measure. We also identify several inferences that may be drawn from the node access patterns, and combine them into a pair of node sequencing algorithms that an adversary may use against different range retrieval methods. With these algorithms, we show that the leaf nodes of an encrypted B$^+$-tree can be fully sequenced with a small number of query traces and negligible computation effort.

In order to deter adversaries from deducing the key values retrieved by users, we propose a privacy-enhancing B$^+$-tree structure, called PB$^+$-tree. The crux of our method is to prevent the adversary from sequencing the leaves in the encrypted B$^+$-tree; we call this *sequence privacy*. PB$^+$-tree groups the encrypted nodes of the B$^+$-tree into buckets. Using homomorphic encryption techniques [8], PB$^+$-tree is able to extract any selected node from a bucket without the server knowing the exact node being read. This obstructs the sequencing of the encrypted nodes. As a side contribution, we extend the PB$^+$-tree approach to cases where the adversary additionally knows the expected access frequencies of the search key values. Experiments show that PB$^+$-tree effectively protects sequence privacy, at the expense of a roughly 15% processing overhead compared to an unprotected B$^+$-tree. Finally, PB$^+$-tree is tunable – a larger bucket size enhances security, at the cost of higher I/O and computation overheads on the server.

II. Related Work

Securing data in storage: The objective here is to prevent the adversary, commonly the untrusted server, from inferring the mapping from the protected database to its plaintext. This category includes using encryption functions (privacy homomorphisms) to allow for arithmetic operations on protected data [9], [10], excluding however the comparison operation which is central to range selections.

Song et al. [11] describe symmetric key methods for keyword search over encrypted document collections, followed by [12] which provides stronger security definitions and an efficient construction. The techniques are applicable only to exact match (keyword) search.

Boneh et al. considered the problem of public key encryption with keyword search in [13]. A semantically secure solution is given in [14]. [15] proposes two deterministic encryption schemes with provable privacy. The privacy safeguard is at the level of *individual* encrypted records. When organized into a B$^+$-tree, the ordering among the encrypted records still reveals information on the query ranges, along the threat outlined in Section I.

Bouganim et al. [16] pushes data encryption, query evaluation and access management to a smartcard acting as mediator between the user and database server. This technique addresses data confidentiality (instead of query privacy), is unsuitable for range queries, and imposes considerable delay due to hardware limitations of the smartcard.

In OPES scheme [17], a plaintext is converted to ciphertext through order-preserving mapping functions. This scheme is secure against ciphertext-only attacks, where the adversary possesses no information beyond the protected database. As explicitly mentioned in [17], OPES (like any order-preserving scheme) fails when the data distribution or the plaintext data are known, as it is then straightforward to associate an encrypted record with its plaintext counterpart.

In [18], Hacigumus et al. proposed to provide the DBMS with hash values of the search keys to facilitate query processing. Two types of hash functions were considered. The first type (order-preserving functions) suffers from the same limitations as OPES. The second (randomized hashing) requires the user to enumerate for the server the hash partitions that overlap with the query range, and post-process them to retrieve the result; in other words, the burden of range selection is pushed to the user.

To efficiently support range queries over large datasets, it is necessary to combine encryption with a tree-index. [6] proposes to store the nodes of a B$^+$-tree as encrypted blocks. In processing a range selection, the user repeatedly retrieves a node, starting with the root, and decrypts it to identify the child node to traverse to. Upon reaching the target leaf node, he then follows the sibling pointers in the leaf level. Another scheme in [19] ensures that an encrypted tree could have resulted from many different plaintext trees. These schemes are not designed against adversaries who may observe the retrieval operations on the protected data. As we show in Section IV-A, the leaf nodes may be sequenced by tracing the block numbers in the I/O requests emanating from range selection operations. If the adversary additionally knows the value distribution of the sort key or has a copy of the plaintext data, he can deduce the query selection conditions.

Securing data in use: Private Information Retrieval (PIR) (e.g., [3] and [20]) is a well-studied access privacy mechanism that ensures the server cannot identify the retrieved data. Existing PIR schemes impose very high computation and/or communication overheads – either linear or poly-

Symbol	Interpretation
n	# leaf nodes in the B$^+$-tree or PB$^+$-tree
N_i	i-th leaf node in the B$^+$-tree or PB$^+$-tree
$\|N\|$	Node size of the B$^+$-tree or PB$^+$-tree
b	# nodes in each bucket for the PB$^+$-tree
N_{ij}	Leaf node stored in j-th slot of bucket i
$\$^N$ ($\$^B$)	An access pattern of node (bucket) identifiers
E	Probabilistic, homomorphic encryption function

TABLE I
NOTATION

logarithmic in the database size [21]. Thus, even though PIR techniques may be extended to indexed datasets [22], they are not appropriate for our purpose of supporting efficient query processing. The same holds for oblivious RAM (ORAM) approaches [4], [20], [5] for similar reasons.

To protect tree indices against analysis of traversal patterns, [23] proposes to retrieve each tree node within a redundancy set that also contains $m-1$ randomly selected nodes, one of which is empty. After reading the target node, the user migrates it to the empty node, re-encrypts all the nodes in the redundancy set, and writes them back to disk. Thus, multiple accesses of a node cannot be discovered by intersecting the redundancy sets. Detailed node migration procedures are given in [24]. In retrieving a node, the mechanism incurs $2m$ random I/Os in reading and writing the redundancy set, and $2m$ times higher communication cost than necessary in sending it back and forth to the user. That is very expensive, especially for m=8 as suggested in [23]. Also, only point queries were investigated.

III. PROBLEM FORMULATION

This section begins by formulating our system and threat models. The models allow for an adversary who may attempt to deduce the entire leaf node sequence of a protected B$^+$-tree, in order to decipher the selection range of every query that utilizes the index. The notion of leaf sequence privacy associated with a B$^+$-tree is introduced in Section III-B. Alternatively, the adversary may attempt to deduce the selection ranges of a specific group of user queries. The privacy notion for this query perspective is addressed in Section III-C. As we will show, where the index is composed of many leaf nodes and there is a large number of user queries, the query privacy notion converges with the sequence privacy notion. For this reason, we focus on the latter in this paper. Table I presents the notation used in this and the following sections.

A. System and Threat Models

Our system comprises two parties – the user and the database server. The user creates the encrypted database and runs queries against it. He also holds the public-private keys for encrypting and decrypting the data. The server provides the resources for running DBMS functions such as data storage and query processing. The server may be shared by other users or controlled by system administrators.

We assume that the adversary has full access to the server including its disk content (but not the data decryption key), and is able to observe the I/O requests generated by user queries. This may be done in several ways, such as intercepting I/O requests with a file system filter driver [25], or the storage may be hosted on an untrusted third-party server. The adversary is able to isolate the I/O requests that emanate from the same query, for example by monitoring the server under light load conditions when there is only one active query. To simplify the discussion, henceforth we equate the server with the adversary. The risks posed by the adversary depend on his knowledge level. A weak adversary may have access only to the encrypted database. In this work, however, we consider two types of *knowledgeable adversaries*, as identified by Damiani et al. in [6].

The first type of adversary possesses the distribution of the search key (Dist), not its actual values. The adversary also has a copy of the encrypted database DB^E, but not the mapping between Dist and DB^E. Dist may be obtained, for example, from published statistics and anonymized tables (in microdata publishing). The second type of adversary has copies of both the plaintext database DB and the encrypted database DB^E (but not the mapping between them). This could occur when the database is shared, and the user creates an encrypted copy for private querying.

B. Leaf Sequence Privacy

By observing the traces of range selections on the encrypted database and its index, a knowledgeable adversary may sequence the leaves of the latter and, in turn, deduce with high confidence the selection conditions of all queries (that utilize the index) as explained in the Introduction.

Privacy Objective: We aim to support efficient processing of range selections over a B$^+$-tree on the encrypted database, while preventing adversaries from ordering its leaf nodes with respect to the search key. Appendix A explains the inability of naïve approaches (such as purposely injecting empty index entries) to achieve our objective.

The granularity of our privacy protection for the index is at the node level (instead of data entries within nodes), because both the encrypted B$^+$-tree [6] and our PB$^+$-tree approach encrypt entire B$^+$-tree nodes. Furthermore, we center our study around the leaf nodes of the tree, because the leaf level provides the finest observable data resolution. Observation 1 elaborates this.

Observation 1: Let $\mathbf{N}_I$ denote the set of (logically) consecutive nodes in an internal level of the B$^+$-tree, that together cover the key range of a query. Let $\mathbf{N}_L$ denote the set of tree leaves that cover the same query range. The ordering among the nodes in $\mathbf{N}_I$ reveals no more information on the query range than the ordering of the nodes in $\mathbf{N}_L$.

Rationale: By construction of the B$^+$-tree, the combined key scope of $\mathbf{N}_I$ is a superset of the combined key scope of $\mathbf{N}_L$. This is because the first and last nodes in $\mathbf{N}_I$ may have descendant leaves that are beyond $\mathbf{N}_L$. Therefore, the information revealed by ordering $\mathbf{N}_L$ is at least as detailed as that disclosed by $\mathbf{N}_I$. □

Based on Observation 1, we focus on the leaf level of the B$^+$-tree to define sequence protection.

Definition 1 (Sequence Privacy): Let $\mathcal{P}_{leaves}$ be the set of possible permutations of the leaf nodes in the B$^+$-tree, and let $\mathbf{N}_{leaves}$ be a random variable denoting the correct sequence of these nodes. A B$^+$-tree provides sequence privacy if for all $p \in \mathcal{P}_{leaves}$, and for any set of observed access patterns A, it holds that

$$Pr[\mathbf{N}_{leaves} = p \mid A] = Pr[\mathbf{N}_{leaves} = p] \qquad (1)$$

In other words, the access patterns do not alter the belief in $\mathbf{N}_{leaves} = p$, thus the adversary gains no advantage in ordering the leaf nodes. This semantic-based privacy definition is analogous to the indistinguishability notion in PIR [3] and Oblivious RAM [4].

Following Definition 1, the access patterns of a B$^+$-tree cause a privacy leak determined by the extent that Equality 1 is violated. This gives rise to our (sequence) privacy metric.

Definition 2 (Privacy Leak): Using the notation in Definition 1, the privacy leak induced by A is quantified as

$$PL = \max_{p \in \mathcal{P}_{leaves}} |Pr[\mathbf{N}_{leaves} = p \mid A] - Pr[\mathbf{N}_{leaves} = p]| \qquad (2)$$

Against the attack of sequencing the leaf nodes of the encrypted B$^+$-tree, intuitively we derive privacy protection from having many possible node sequences that are consistent with the observed access patterns. Suppose that the B$^+$-tree contains n leaf nodes and $|\mathcal{P}_{leaves}| = n!$. Initially, every encrypted leaf node is equally likely to correspond to any of the n positions, so $Pr[\mathbf{N}_{leaves} = p] = 1/n!$ for all $p \in \mathcal{P}_{leaves}$. As the adversary observes the retrieval sequences A, he can narrow down the possible positions of the encrypted nodes, thus reducing the belief in certain $p \in \mathcal{P}_{leaves}$ to zero. For example, a node that appears fifth in a retrieval sequence may be any of leaf nodes five to n, so the number of possible positions for the node is $n - 4$. If the adversary further observes that node to precede 9 other nodes, then the number of possible positions narrows to $n - 13$. Suppose that eventually only χ possible p's are consistent with A, and that they are equally probable, i.e., the adversary has no additional information to consider one more likely than the others. In this case, we have $Pr[\mathbf{N}_{leaves} = p|A] = 1/\chi$ for those p's, and

$$PL = \max(1/\chi - 1/n!, 1/n!) \qquad (3)$$

C. Query Privacy

Instead of sequencing all the leaf nodes in the protected B$^+$-tree, the adversary may attempt to deduce the index key values in the B$^+$-tree nodes accessed by selected queries.

Suppose that the leaf nodes are numbered in sequence from 1 to n. In the simplest scenario, the adversary is interested in a solitary query which utilized one leaf node. The problem of guessing the key values covered by that leaf node is equivalent to guessing its node number. The adversary may pick randomly one of the n node numbers, or he may deduce it from the position of the node in one of the χ permutations (where χ is the number of leaf node permutations that are consistent with the access patterns). Hence the probability of success is $\max(1/n, 1/\chi)$.

Generalizing, the adversary may be interested in x (not necessarily adjacent) leaf nodes that are accessed in one or more user queries, for some $1 \leq x \leq n$. Now, the adversary's chance of randomly guessing the x node numbers is $(n-x+1)!/n!$, or he may observe the positions of the nodes in one of the χ permutations. Overall, his probability of success is $\max((n - x + 1)!/n!, 1/\chi)$.

For small x values, the adversary's success probability is determined by the first factor $(n - x + 1)!/n!$ and can be calculated easily. For large n and x values, the success probability converges to $1/\chi$. From Equation 3, the PL associated with sequencing the leaf nodes converges to $1/\chi$ too. This is intuitive – the difficulty of deducing correctly a large number of leaf nodes is expected to approach that of correctly numbering all the leaf nodes (i.e., sequencing the leaf nodes). Therefore, we focus on the sequence privacy measure PL, with the understanding that it also measures the query privacy protection at large n and x values.

IV. Range Retrieval with Encrypted B$^+$-tree

The B$^+$-tree is the standard index for range retrieval. In this section, we design concrete algorithms for the adversary to exploit the vulnerabilities of an encrypted B$^+$-tree [6] arising from the standard traversal strategy, as well as from an alternative query processing method that makes privacy breach tougher (but still achievable).

A. Inference from Sibling Pointer Traversal

With an encrypted B$^+$-tree, the server cannot decipher the nodes that are read by a query, but it can track the leaf nodes retrieved through their sibling pointers, as explained in the Introduction. From overlapping leaf node sequences $[N_\alpha, N_{\alpha+1}, \ldots, N_\beta]$ and $[N_{\beta-\gamma}, \ldots, N_\beta, \ldots, N_\delta]$, the adversary can stitch together a longer sequence $[N_\alpha, \ldots, N_{\beta-\gamma}, \ldots, N_\beta, \ldots, N_\delta]$, where $1 \leq \alpha < \beta - \gamma \leq \beta < \delta$. When enough range selections have been issued to connect all the leaf nodes, they can be sequenced completely. By further corroborating with the data distribution (Dist) or plaintext database (DB), the adversary may estimate the key ranges of posed queries. However, even partial sequences derived in the interim provide the adversary with ordering information, albeit incomplete.

Suppose that the access sequences allow us to stitch together ρ partial sequences covering t of the n leaf nodes; in other words, there are $n-t$ leaf nodes that do not belong to any partial sequence. There are $(n-t+\rho)!$ permutations of the partial sequences and unsequenced nodes, thus

$$PL = \max\left(\frac{1}{(n-t+\rho)!} - \frac{1}{n!}, \frac{1}{n!}\right) \qquad (4)$$

The increase in PL for this method is proportional to the query selection range and to the number of observed queries, topping off at $1-1/n!$ quickly (signifying complete disclosure) as shown in Section VII.

B. Inference from Subtree Retrieval

Instead of traversing sibling pointers, an alternative is to fetch the leaf nodes within the query range through their covering subtree, as illustrated in Figure 2. Starting from the root node, the modified procedure retrieves all

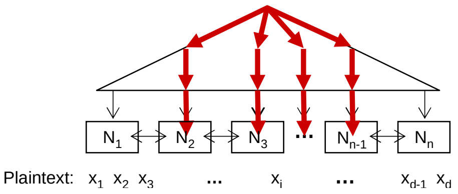


Fig. 2. Range Query with Subtree Retrieval in B$^+$-tree

the child nodes that cover the query range; this is repeated for each level on the way down to the leaf nodes. In the process, all the required nodes in each level are sorted by their physical addresses, and requested together. This modified B$^+$-tree traversal mechanism is intended to mask the logical ordering among the nodes within each level. However, an adversary can still order the leaf nodes based on the observation that *adjoining leaf nodes should co-occur more frequently in the range retrieval operations, relative to non-adjoining ones*. For example, the retrieval operations that contain both N_1 and N_2 cannot be fewer than those containing N_1 and N_3. Section IV-B1 shows how to sequence the leaf nodes from their co-occurrence patterns in range retrievals.

1) Sequencing Algorithm: We first highlight a number of defining properties in a logical sequence of leaf nodes; justification/examples for these properties are given in our technical report [26]. Based on these properties, we develop an algorithm to deduce the node ordering from I/O requests observed at the server for selections that involve *at least two successive leaf nodes*. As in Section IV-A, we assume that the adversary can prune away the leading requests for internal B$^+$-tree nodes. Thus, each range retrieval produces an access pattern of leaf node addresses.

Our algorithm uses heavily the following notation:

- V denotes the set of leaf nodes in the B$^+$-tree.
- $N_1, \ldots, N_n$ is the logical sequence of B$^+$-tree leaf nodes.
- $A = \{a_{ij}\}$ denotes the *association matrix* of the leaf nodes, such that a_{ij} is the number of range retrievals in which N_i and N_j co-occur. A is symmetric, i.e., $a_{ij} = a_{ji}\ \forall 1 \le i, j \le n$.
- $C = \{c_i\}$ denotes the vector of access counts for the leaf nodes, i.e., c_i is the number of range retrievals that contain N_i.
- S denotes a (partial) sequence of leaf nodes. $S.left$ and $S.right$ refer to the leftmost and rightmost nodes in S, respectively.

The following observations lay the foundation for a complete algorithm (presented afterwards) for sequencing when subtree retrieval is employed.

Observation 2: In a logical sequence of leaf nodes, $\forall 1 \le i < j < k \le n$, $a_{ij} \ge a_{ik}$ and $a_{jk} \ge a_{ik}$.

Observation 3: In a logical sequence of leaf nodes, $\forall 1 \le i \le n$ such that $a_{i,i+1} = a_{i,i+2}$, we have $c_{i+1} \le c_{i+2}$.

Definition 3: Two nodes N_i and N_j are **indistinguishable** if there is no range retrieval involving one of them but not the other; otherwise, N_i and N_j are **distinguishable**.

Observation 4: Two nodes N_i and N_j are indistinguishable if and only if $c_i = c_j = a_{ij}$.

Pre-processing: We first aggregate the indistinguishable nodes into hypernodes. We sort by c_i the nodes in C and in A. Following that, we identify pairs of indistinguishable nodes N_i and N_j according to Observation 4, and replace them[1] with a hypernode N_i^H in C and A. This leaves us with only distinguishable nodes/hyper-nodes, which exhibit the following key properties that facilitate sequencing.

Definition 4: N_k is a **differentiator** for a pair of distinguishable nodes N_i and N_j if $a_{ik} \neq a_{jk}$.

Observation 5: Suppose that the leaf nodes are sequenced correctly. Consider N_i, N_{i+1} and N_{i+2}. If N_{i+2} is not a differentiator for N_i and N_{i+1}, then $\nexists$ any differentiator N_j for N_i and N_{i+1} such that $i + 2 < j \le n$.

Observation 6: Suppose that the leaf nodes are sequenced correctly. Two nodes N_i and N_{i+1} are distinguishable only if at least one of their immediate left and right neighbors is a differentiator for them, i.e., $a_{i-1,i} > a_{i-1,i+1}$ or $a_{i,i+2} < a_{i+1,i+2}$.

Observation 7: Consider pairwise distinguishable leaf nodes N_i, N_{i+1}, N_{i+2} that are sequenced correctly, where N_i is not a differentiator for N_{i+1} and N_{i+2}. The access counts must satisfy the condition $c_{i+1} < c_{i+2}$.

Definition 5: The direction of a node N_k with respect to a partial sequence $S = \langle N_i, N_{i+1}, \ldots, N_{i+j} \rangle$ is **determinable** if there is evidence to place N_k definitely to the left or right of S.

Observation 8: The direction of a node N_k with respect to a partial sequence $S = \langle N_i, N_{i+1}, \ldots, N_{i+j} \rangle$ is *determinable* if

(a) $a_{l,k} < a_{i+j,k}$ for some $i \le l < i + j$, in which case N_k falls to the right of S; or

(b) $a_{l,k} < a_{i,k}$ for some $i < l \le i + j$, in which case N_k falls to the left of S; or

(c) $a_{i,k} = \ldots = a_{i+j,k}$, and $\exists N_l$ that is determinable with respect to S and $max(a_{i,l}, a_{i+j,l}) < a_{i,k}$; in this case if $max(a_{i,l}, a_{i+j,l}) \le a_{k,l}$ then N_k falls on the same side of S as N_l, otherwise N_k falls on the opposite side of S from N_l.

Observation 9: Given a set V of distinguishable nodes and hypernodes, and a partial sequence S, a necessary condition for sequencing V is that there exists at least one node $N_k \in V$ such that the direction of N_k relative to S is determinable.

Observation 10: Two partial sequences S_1 and S_2 are ordered correctly as $S_1 S_2$ if and only if at least one of the following conditions is satisfied:

(a) $a_{S_1.right,S_2.left} > a_{S_1.right,S_2.right}$; or $a_{S_1.right,S_2.left} = a_{S_1.right,S_2.right}$ and there exists some node N_k not part of S_1 and S_2 such that $a_{S_2.right,k} > a_{S_2.left,k} \ge a_{S_1.right,k} \ge a_{S_1.left,k}$.

(b) $a_{S_1.left,S_2.left} < a_{S_1.right,S_2.left}$; or $a_{S_1.left,S_2.left} =$ $a_{S_1.right,S_2.left}$ and there exists some node

[1] Indistinguishable nodes cause uncertainty in the deduced node sequence, and are accounted for in the PL measure in Section IV-B2.

Algorithm 1 Sequence the leaf nodes of an encrypted B$^+$-tree from their co-occurrences in range selections

```
1: Group the indistinguishable nodes into hypernodes.
2: Set $V$ to contain the distinguishable leaf nodes and hypernodes.
3: $S$ = SequenceNodes($V$).
4: **if** $S$ covers all nodes in $V$ **then** output complete sequence $S$.
5: **else** output partial sequence $S$.
   **Function:** SequenceNodes($V$)
6: Set $S = \langle N_i, N_j \rangle$ where the pair $N_i, N_j$ has the largest $a_{ij}$ in $A$. Resolve ties arbitrarily.
7: GrowSequence($S$, $V$).
8: Return $S$.
   **Function:** GrowSequence($S$, $V$)
9: Initialize $U = \emptyset$ (for ambiguous nodes).
10: Let $R = \{N_k | max(a_{S.left,k}, a_{S.right,k}) > 0$ and $(\forall N_l \in V - S - \{N_k\}, max(a_{S.left,k}, a_{S.right,k}) \geq max(a_{S.left,l}, a_{S.right,l}))\}$.
11: **while** $R$ is not empty **do**
12:     $V = V - R$.
13:     $U = U \cup \{N_k | N_k \in R, a_{S.left,k} = a_{S.right,k}\}$.
14:     $R_{right} = \{N_k | N_k \in R, a_{S.left,k} < a_{S.right,k}\}$.
15:     $R_{left} = \{N_k | N_k \in R, a_{S.left,k} > a_{S.right,k}\}$.
16:     **if** $R_{left}$ and $R_{right}$ are both empty **then** return Fail.
17:     **if** $R_{right}$ is not empty **then**
18:         $R_{right} = R_{right} \cup \{N_k | N_k \in U, a_{S.right,l} \leq a_{S.right,k}$ and $a_{kl} \geq a_{S.right,l}$ for any $N_l \in R_{right}\}$.
19:         $R_{left} = R_{left} \cup \{N_k | N_k \in U, a_{S.right,l} \leq a_{S.right,k}$ and $a_{kl} < a_{S.right,l}$ for any $N_l \in R_{right}\}$.
20:         $U = U - (R_{left} \cup R_{right})$.
21:     Repeat lines 17–20 for $R_{left}$.
22:     **if** $R_{right}$ is not empty **then** ExpandRight($R_{right}$, $S$, $V$).
23:     **if** $R_{left}$ is not empty **then** ExpandLeft($R_{left}$, $S$, $V$).
24:     Let $R = \{N_k | max(a_{S.left,k}, a_{S.right,k}) > 0$ and $(\forall N_l \in V - S - \{N_k\}, max(a_{S.left,k}, a_{S.right,k}) \geq max(a_{S.left,l}, a_{S.right,l}))\}$.
25: Return Success.
   **Function:** ExpandRight($R_{right}$, $S$, $V$)
26: **while** $R_{right}$ is not empty **do**
27:     $T = \{N_i | N_i \in R_{right}, \forall N_j \in R_{right}, a_{S.right,i} \geq a_{S.right,j}\}$.
28:     $R_{right} = R_{right} - T$.
29:     **if** $|T| = 1$ **then**
30:         Remove $N_i$ from $T$, append it to the right of $S$.
31:     **else if** $|T| = 2$ **then**
32:         Suppose $T = \{N_i, N_j\}$ and $c_i < c_j$.
33:         Remove $N_i$ from $T$, append it to the right of $S$.
34:         Remove $N_j$ from $T$, append it to the right of $S$.
35:     **else** ($|T| > 2$)
36:         $S_{right}$ = SequenceNodes($T$).
37:         **if** $S_{right}$ contains all the nodes in $T$ **then**
38:             Append $S_{right}$ to the right of $S$.
39:         //else, return to GrowSequence(.)
```

N_k not part of S_1 and S_2 such that $a_{S_1.left,k} > a_{S_1.right,k} \geq a_{S_2.left,k} \geq a_{S_2.right,k}$.

Sequencing: Algorithm 1 exploits the above observations to sequence the leaf nodes of the B$^+$-tree, using the access count vector C and the association matrix A formed from range retrieval patterns. It begins by forming a partial sequence S from the pair of nodes that have the largest co-occurrence count (line 6), which guarantees that they are immediate neighbors of each other; ties are broken arbitrarily except for a special case discussed below. The partial sequence is then extended by iteratively adding the immediate neighbor of either edge.

We extend the sequence as follows. We place into set R the (one or more) unconnected nodes that have the largest co-occurrence count with either edge of S (line 10). If there is no such node, the algorithm exits. Otherwise, the nodes in R are divided into three subsets: (i) U contains those for which there is still insufficient information to be placed to the left or right of S, (ii) R_{right} contains nodes that lie to the right of S based on Observation 8(a), and (iii) R_{left} contains nodes that lie to the left of S according to Observation 8(b). The nodes in R_{right} may enable us to move some nodes from U to R_{left} and R_{right}, using Observation 8(c) (lines 17–20); R_{left} is also used for this purpose (line 21). With that, we add the nodes in R_{left} and R_{right} to the left side and right side of S.

The procedure for extending S with R_{right} is also given in Algorithm 1; the process for R_{left} is symmetric and thus omitted. We move into set T the nodes from R_{right} that have the highest co-occurrence count with the right edge of S (lines 27–28). If there is just one such node, we simply add it to the right edge of S (lines 29–30). If there are two nodes in T with the same co-occurrence count with the right edge of S, the relative order between them is determined by their access counts, based on Observation 7 (lines 31–34). If T contains more than two nodes, lines 35–38 recursively invoke function SequenceNodes(.) to sequence T into S_{right}, and stitch S_{right} to the right of S (abiding by the conditions in Observation 10, which may require flipping S_{right} around).

Where the observed range retrievals are insufficient to achieve total ordering among the leaf nodes, Algorithm 1 can find multiple maximal partial sequences. The idea is to repeatedly execute the algorithm with the seed S formed by the pair of nodes in the unsequenced pool with the highest co-occurrence count.

The above co-occurrence inference applies to any privacy mechanism that horizontally partitions the records according to key values and subsequently encrypts them (e.g., [18]). Access patterns stemming from range retrievals would again contain logically adjoining partitions, hence revealing their relative ordering.

Discussion on Algorithm 1: The sequencing begins in line 6 with the pair of nodes N_i, N_j with the largest co-occurrence count a_{ij}. If there exist two (or more) such pairs involving distinct nodes N_i, N_j, N_k, N_l such that $a_{ij} = a_{kl}$, we may pick any pair. The case, however, where there is a logical node sequence N_i, N_{i+1}, N_{i+2} with $a_{i,i+1} = a_{i+1,i+2}$ warrants closer examination to resolve the tie:

- Prior to this point, all the indistinguishable nodes have been replaced by hypernodes in line 1 of the algorithm. Hence N_i, N_{i+1}, N_{i+2} must be distinguishable nodes or hypernodes. By Observation 4, the node access counts must satisfy $c_i \neq c_{i+1}$ and $c_{i+1} \neq c_{i+2}$.
- Since we are dealing with range selection operations on successive logical nodes, it is not possible for a range query to cover N_i and N_{i+2} while skipping N_{i+1}; there-

fore the node access count $c_{i+1} = a_{i,i+1} = a_{i+1,i+2}$, implying that (a) $a_{i,i+2} = a_{i,i+1} = a_{i+1,i+2}$, and (b) $c_i \geq c_{i+1}$ and $c_{i+2} \geq c_{i+1}$.

Combining the two points above, we conclude that $c_i > c_{i+1}$ and $c_{i+2} > c_{i+1}$. Hence, amongst the three nodes being considered, the one with the lowest node access count is N_{i+1}, the node that is logically in between the other two. We may therefore begin with any one of the two pairs involving N_{i+1}, leading to pairing N_i and N_{i+1}, or pairing N_{i+1} and N_{i+2}.

2) Privacy Analysis: We begin by aggregating the indistinguishable leaf nodes into hypernodes N_1^H, N_2^H, …, N_h^H, with $|N_i^H|$ indistinguishable nodes constituting hypernode N_i^H. The number of distinguishable nodes and hypernodes that are input to the sequencing algorithm is $n - \sum_{i=1}^{h} |N_i^H| + h$. Suppose that the algorithm outputs ρ partial sequences (each with two or more nodes) that together cover t nodes. There are $(n - \sum_{i=1}^{h} |N_i^H| + h - t + \rho)!$ permutations of the partial sequences and unsequenced nodes/hypernodes. For a given permutation, either side of each partial sequence may precede the other in the overall node ordering; moreover, there are $|N_i^H|!$ permutations of the nodes within each hypernode N_i^H. Therefore there are $2^\rho \cdot \prod_{i=1}^{h} |N_i^H|! \cdot (n - \sum_{i=1}^{h} |N_i^H| + h - t + \rho)!$ possible sequences of the leaf nodes, leading to

$$PL = \max\left(\frac{1}{n!}, \frac{1}{2^\rho \prod_{i=1}^{h} |N_i^H|!(n - \sum_{i=1}^{h} |N_i^H| + h - t + \rho)!} - \frac{1}{n!}\right) \quad (5)$$

The maximum PL here is $1/2 - 1/n!$, because even with the nodes completely sequenced, the co-occurrence patterns provide insufficient information to pin the edges of the node sequence to the lower/upper end of the key range.

As we show in the experiments, query processing with subtree retrieval in the encrypted B$^+$-tree is indeed more secure than with plain sibling pointer traversal. However, the PL of the former still rises rapidly. This motivates our PB$^+$-tree method presented next.

V. PB$^+$-TREE: COUNTERING NODE SEQUENCING

In this section, we describe our privacy-enhancing B$^+$-tree (PB$^+$-tree). Following an overview of the method in Section V-A, Section V-B elaborates on the storage organization and retrieval techniques. Section V-C then analyzes how an adversary may attack the PB$^+$-tree. Finally, Section V-D discusses updates on the PB$^+$-tree, and how its security strength could be enhanced through periodic re-organization.

A general comment about the attack method against PB$^+$-tree, as well as the adversarial algorithms in Section IV, is that they are only some of the possible procedures a knowledgeable adversary could follow to sequence the index nodes. While we cannot eliminate the possibility that more effective attack algorithms exist, we need to equip the adversary with specific techniques (the most sophisticated we could devise) in order to evaluate the strength of each privacy protection scheme.

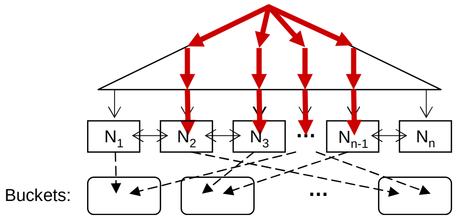


Fig. 3. Storage of B$^+$-tree Nodes in Encrypted Buckets

A. Overview of PB$^+$-tree

The PB$^+$-tree has the same structure as the standard B$^+$-tree. The records of the base relation are stored in a separate unsorted heap file; i.e., the PB$^+$-tree is an unclustered index. Note that the index cannot be clustered; otherwise, the key value of a record can be deduced trivially from its offset within the base relation file, since the adversary knows the key distribution (from Dist or DB). This restriction applies to the encrypted B$^+$-tree too.

As in [6], we encrypt the content (including the key values and child pointers) of every PB$^+$-tree node and each data record, so that the adversary cannot see the key values looked up by the user. To enhance access privacy, however, we need to incorporate additional security mechanisms.

To deter the adversary from tracking the traversed nodes in range retrievals (as in Section IV-B), we do not store them individually. Instead, the nodes in each index level are grouped randomly into buckets, where each bucket occupies a disk block[2]. In extracting a required node from its host bucket, we employ homomorphic encryption techniques so that, to an adversary, all the nodes within the bucket appear equally likely to be the extraction target. Consequently, the encrypted nodes that make up a range retrieval cannot be tracked with certainty.

Just like the PB$^+$-tree nodes, the encrypted records in the underlying heap file are grouped into buckets. The retrieval of a record from its host bucket also involves homomorphic encryption techniques. Since the same security mechanisms apply to the PB$^+$-tree nodes and records, we shall not discuss record protection separately.

Figure 3 illustrates the assignment of the PB$^+$-tree leaf nodes to buckets. Suppose that the first bucket holds N_1 and N_9, the second bucket holds N_3 and N_{n-1}, and the last bucket holds N_2 and N_{81}. When a range retrieval operation requests for N_1 and N_2, the adversary is only aware that one node is accessed from the first bucket, and another from the last bucket. Without more information, to an adversary the pair of accessed nodes is equally likely to be $\{N_1, N_2\}$, $\{N_1, N_{81}\}$, $\{N_9, N_2\}$ or $\{N_9, N_{81}\}$.

We follow the subtree retrieval strategy in Section IV-B. Starting with the root node, all the child nodes covering the query range are sorted by bucket address and requested together. This process is repeated all the way down the PB$^+$-tree to retrieve the matching records. Since the requests for all the required nodes in each level of the PB$^+$-tree arrive at the server at once, multiple nodes that reside in

[2] Unlike the encrypted B$^+$-tree where each node takes up a disk page, here multiple nodes (with smaller capacity) are placed in each bucket/disk block. The rationale behind this design choice is explained in Section V-B.

the same bucket can be fetched from disk with a single I/O operation. Likewise, one I/O suffices to fetch the encrypted records hosted in the same bucket. This is an important optimization to reduce the I/O cost of the PB$^+$-tree, which we confirm through experiments in Section VII.

B. Node Placement and Retrieval in PB$^+$-tree

Suppose that b (> 1) nodes from the same PB$^+$-tree level are assigned to each bucket, and that each node has a size of $|N|$ "fragments". Conceptually, a bucket contains $b \times |N|$ encrypted fragments $E(d_{ik})$, where d_{ik} denotes the k-th fragment of the i-th node in the bucket. Moreover, E is a probabilistic, homomorphic encryption function like BGN [8] that allows a ciphertext to undergo a multiplication, followed by an arbitrary number of additions[3]. Each node address is an $\langle addr, i\rangle$-pair where $addr$ is the disk address of the host bucket and i is the node's offset within the bucket. To retrieve the i-th node in a bucket, the user sends to the server the bucket's address and a vector $q = \langle q_1, \ldots, q_b\rangle$ in which $q_i = E(1)$ and $q_j = E(0)$ $\forall j \neq i$; the various q_j's are distinct because E is a probabilistic encryption. After fetching the bucket, the server composes the encrypted content of the target node by computing $\sum_{j=1}^{b} q_j \times E(d_{jk}) = E(d_{ik})$, for $1 \leq k \leq |N|$.

We set the bucket size equal to the physical block size of the disk, thus reducing the fanout of the PB$^+$-tree by a factor of b. In the worst case, this increases the height of the tree by one (because meaningful values of b, expected to be no more than 10 or 20, are much smaller than the fanout which is typically in the order of hundreds), and may incur an I/O overhead compared to the encrypted B$^+$-tree. We prefer this alternative over allocating b physical blocks per bucket, as there is no guarantee that those blocks will be placed contiguously on the disk, and fetching a node could lead to multiple random I/Os.

In view that the child pointers in the PB$^+$-tree are encrypted, each level of the PB$^+$-tree entails one round of user-server interactions, since the user needs to decrypt the node(s) in the current level to determine the child node(s) to traverse to. The latency caused by the interactions is tolerable if the network connection between the user and server is fast, as assumed in Damiani et al.'s model in [6]; this is our default setting as well. In case of slower networks, one option is to embed into the server a secure co-processor [27] that is controlled by the user. In traversing down the PB$^+$-tree, the server interactions with the secure co-processor (which acts as a trusted agent for the user) go over the system bus, which is much faster than the network. However, co-processors typically have limited computing resources, and their programming is cumbersome.

C. Vulnerability Analysis of the PB$^+$-tree

Suppose that the adversary has the history of I/O requests issued by range retrieval operations that together cover all the leaf nodes of the PB$^+$-tree. Since the retrieval technique of the PB$^+$-tree masks the exact node addresses, each access pattern obtained by the adversary is a set of addresses of the buckets that hold the required leaf nodes. Before an access pattern can be useful for sequencing the leaf nodes, the adversary first has to deduce the node address that underlies each bucket access, in order to derive the corresponding access pattern of leaf node addresses. To facilitate the deduction, the adversary may exploit the correlation between bucket access patterns that involve common leaf nodes, as explained below.

When a bucket appears in two bucket access patterns, it could be because they retrieve the same node, or they retrieve different nodes that just happen to reside in the same bucket. Intuitively, if the patterns have two buckets in common, the overlap is more likely to be indeed due to identical nodes accessed, and so on. Interestingly, when the number of common buckets exceeds a limit, the probability that part of the bucket overlap is coincidental increases. Below, we study the probability that a bucket overlap is indeed due to identical nodes retrieved. We utilize this analysis to process first those overlaps that are highly likely to stem from identical nodes, and use them to reinforce the confidence about other overlaps that we are less certain about. The following formulation quantifies the confidence in bucket overlaps of different lengths.

Suppose the adversary observes two range retrieval operations that produce (leaf-level) *bucket* access patterns $\mathbb{S}_\alpha^B$ and $\mathbb{S}_\beta^B$, with x common buckets between them. The adversary may conjecture that the common buckets stem from sub-patterns $\mathbb{S}_\alpha^N$ and $\mathbb{S}_\beta^N$ within the respective range operations that span the same x successive leaf nodes. By Bayes' rule, the likelihood of this conjecture is

$$\begin{aligned}
&\text{Prob}(|\mathbb{S}_\alpha^N \cap \mathbb{S}_\beta^N| = x \bigm| |\mathbb{S}_\alpha^B \cap \mathbb{S}_\beta^B| = x) = \\
&\quad \big(\text{Prob}(|\mathbb{S}_\alpha^B \cap \mathbb{S}_\beta^B| = x \bigm| |\mathbb{S}_\alpha^N \cap \mathbb{S}_\beta^N| = x)\times \\
&\qquad\qquad \text{Prob}(|\mathbb{S}_\alpha^N \cap \mathbb{S}_\beta^N| = x)\big) \;/ \\
&\quad (\sum_{i=0}^{x} \text{Prob}(|\mathbb{S}_\alpha^B \cap \mathbb{S}_\beta^B| = x \bigm| |\mathbb{S}_\alpha^N \cap \mathbb{S}_\beta^N| = i) \times \\
&\qquad\qquad \text{Prob}(|\mathbb{S}_\alpha^N \cap \mathbb{S}_\beta^N| = i)\big)
\end{aligned}$$

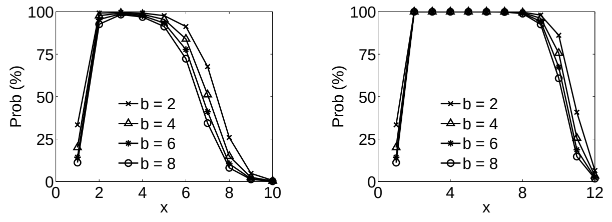
Since each of $\mathbb{S}_\alpha^N$ and $\mathbb{S}_\beta^N$ comprises x successive nodes among the n leaf nodes,

$$\text{Prob}(|\mathbb{S}_\alpha^N \cap \mathbb{S}_\beta^N| = i) = \begin{cases} \frac{1}{n-x+1} & \text{if } i = x \\ \frac{2(n-2x+i+1)}{(n-x+1)^2} & \text{if } 1 \leq i < x \\ \frac{(n-2x+1)(n-2x+2)}{(n-x+1)^2} & \text{if } i = 0 \end{cases}$$

Assuming that most of the queries retrieve a small number of leaf nodes relative to n, the buckets in $\mathbb{S}_\alpha^B$ are expected to be distinct, and likewise for $\mathbb{S}_\beta^B$. Thus, $\text{Prob}(|\mathbb{S}_\alpha^B \cap \mathbb{S}_\beta^B| = x \bigm| |\mathbb{S}_\alpha^N \cap \mathbb{S}_\beta^N| = i) = (x-i)!(\frac{b}{n})^{x-i}$. After substituting the component probabilities and simplifying, we get

$$\begin{aligned}
&\text{Prob}(|\mathbb{S}_\alpha^N \cap \mathbb{S}_\beta^N| = x \bigm| |\mathbb{S}_\alpha^B \cap \mathbb{S}_\beta^B| = x) = \\
&\frac{n-x+1}{\begin{aligned}[(n-x+1) + \textstyle\sum_{i=1}^{x-1} 2(x-i)!(\frac{b}{n})^i(n-2x+i+1) \\ + x!(\frac{b}{n})^x(n-2x+1)(n-2x+2)]\end{aligned}} \qquad (6)
\end{aligned}$$

[3] In applying BGN, we use 1024 bits for the key, and configure the setting to allow for messages of 512 bits. Thus our PB$^+$-tree nodes and records are encrypted and extracted in fragments of 512 bits.



(a) n = 1,000 nodes (b) n = 1 million nodes

Fig. 4. Prob($|\mathbb{S}^N_\alpha \cap \mathbb{S}^N_\beta| = x \mid |\mathbb{S}^B_\alpha \cap \mathbb{S}^B_\beta| = x$)

Figure 4 illustrates Prob($|\mathbb{S}^N_\alpha \cap \mathbb{S}^N_\beta| = x \mid |\mathbb{S}^B_\alpha \cap \mathbb{S}^B_\beta| = x$) for various n, b and x values. The results show that, for practical combinations of n and b, a pair of $\mathbb{S}^B_\alpha$ and $\mathbb{S}^B_\beta$ bucket access patterns that share just one common bucket address (i.e., $x = 1$) lends very little credence to an inference that the overlap is attributable to the same node. However, when $\mathbb{S}^B_\alpha$ and $\mathbb{S}^B_\beta$ overlap by two to five buckets in the case of n=1,000, and by two to eight buckets in the case of n=1 million, there is strong evidence that the overlapping buckets resulted from identical leaf node requests. Beyond five or eight buckets, there is an increasing probability that only some but not all of the overlapping buckets are due to requests for the same nodes, i.e., there are false positives. Hence, an inference that *all* of the underlying node requests are the same should be made only if it is supported by additional pairs of bucket access patterns; how this is done is explained shortly.

The above observations lead to Algorithm 2, which transforms the bucket co-occurrence patterns observed by the adversary into node access patterns for sequencing the leaf nodes. The input consists of leaf level bucket accesses that cover at least two buckets. Let N_{ij} denote the PB$^+$-tree leaf node that resides in the j-th slot of bucket i. In the first segment (lines 1–8), after sorting the bucket patterns by length (we will see the rationale shortly), each bucket identifier $bucId_i$ in a bucket pattern is mapped to an ambiguous node identifier N_{i*} (i.e., unknown slot within bucket i). Line 8 creates a vector cnt, in which the j-th cell holds the next available slot identifier to use upon the disambiguation of a node N_{i*} in bucket i.

In the next segment (lines 9–18), for every pair of bucket patterns that overlap by two or more buckets, we instantiate the slot number in the identifier of the nodes in the overlap. As the bucket patterns are sorted in increasing length, in the initial iterations we are processing bucket patterns that (are short and thus tend to) overlap on a small number of buckets ($|\mathbb{S}^B_\alpha \cap \mathbb{S}^B_\beta|$). As explained previously, such an overlap provides high confidence that the underlying nodes are identical in the two access patterns. Later iterations encounter pairs of bucket patterns with increasingly longer overlaps. For such long-overlap pairs, some of the common nodes will have already been resolved by earlier, shorter bucket patterns. This helps to reduce instances of erroneously equating distinct nodes across long-overlap patterns. In line 17, N_{ij} is a node that has been disambiguated and assigned slot identifier j in bucket i in a previous iteration. If $\mathbb{S}^N_\alpha$ contains N_{ij}, then N_{i*} in $\mathbb{S}^N_\beta$ is instantiated with slot identifier j, and vice versa.

Algorithm 2 Deduce node patterns $\mathbb{S}^N$ from bucket patterns $\mathbb{S}^B$

```
    // Initialization.
 1: Sort access patterns $\mathbb{S}^B$ in increasing length order.
 2: Set $s = |\mathbb{S}^B|$ (i.e., number of bucket access patterns).
 3: for $\alpha$ = 1 to $s$ do
 4:     Create node pattern $\mathbb{S}^N_\alpha$.
 5:     for all bucket identifiers $bucId_i \in \mathbb{S}^B_\alpha$ do
 6:         Map $bucId_i$ to $N_{i*}$.
 7:         Insert $N_{i*}$ into $\mathbb{S}^N_\alpha$.
 8: Create $\lceil \frac{n}{b} \rceil$-vector $cnt = [\ 1\ 1\ \ldots\ 1\ ]$.
    // Disambiguate the node identifiers.
 9: for $\alpha$ = 1 to $s$ do
10:     for $\beta = \alpha + 1$ to $s$ do
11:         if $|\mathbb{S}^B_\alpha \cap \mathbb{S}^B_\beta| \geq 2$ then
12:             for all $bucId_i \in \mathbb{S}^B_\alpha \cap \mathbb{S}^B_\beta$ do
13:                 if both $\mathbb{S}^N_\alpha$ and $\mathbb{S}^N_\beta$ contain $N_{i*}$ then
14:                     Change $N_{i*}$ to $N_{i,cnt[i]}$ in $\mathbb{S}^N_\alpha$.
15:                     Change $N_{i*}$ to $N_{i,cnt[i]}$ in $\mathbb{S}^N_\beta$.
16:                     Increment $cnt[i]$.
17:                 else if $\mathbb{S}^N_\alpha$ or $\mathbb{S}^N_\beta$ contains $N_{ij}$ then
18:                     Change $N_{i*}$ to $N_{ij}$ in $\mathbb{S}^N_\beta$ or $\mathbb{S}^N_\alpha$.
    // Remove unresolved access patterns.
19: for $\alpha$ = 1 to $s$ do
20:     if $\mathbb{S}^N_\alpha$ contains any unresolved node $N_{i*}$ then delete $\mathbb{S}^N_\alpha$.
```

In the last segment of the procedure (lines 19–20), the access patterns that contain any node with an ambiguous identifier (i.e., the slot number in the identifier is uninstantiated) are deleted. The adversary cannot utilize such an access pattern by simply removing the ambiguous nodes from it, because this would disrupt the contiguity of the pattern, and there is no way of inferring how to group the remaining nodes into contiguous sub-patterns.

When Algorithm 2 completes, every N_{ij} in the remaining access patterns represents a unique leaf node. The node patterns are then used to derive the association matrix A and the access count vector C, which in turn are input to Algorithm 1 to sequence the leaf nodes. PL computation from the sequencing outcome follows Formula 5.

While Algorithm 2 attempts to avoid equating different nodes across access patterns (in lines 9–18), such errors cannot be totally eliminated. This is an additional reason PB$^+$-tree is much harder to attack than the encrypted B$^+$-tree. In evaluating its privacy strength in Section VII, we exclude from the sequencing phase those access patterns that contain nodes with wrongly instantiated slot numbers. This effectively benefits the adversary by preventing the generation of incorrectly ordered sequences; i.e., the PL reported in the experiments for PB$^+$-tree is only a upper bound of the privacy leak that it actually permits[4].

D. Update and Node Migration

While our primary focus is on query processing, PB$^+$-tree also supports updates. The user first fetches the bucket

[4] The alternative is to include the wrongly instantiated node patterns in the sequencing process. Any wrongly ordered nodes in the partial sequences generated can be discounted in the PL computation later. We found the resulting PL to be much lower, which is disadvantageous to the adversary. To be conservative in judging the strength of PB$^+$-tree, we decided against this alternative.

that hosts the affected leaf node. Besides changing the leaf node, the user also re-encrypts the other nodes in the bucket, before writing it back to disk. Since the encryption function is probabilistic, all the nodes will appear to have changed, so the adversary cannot pinpoint the modified node. Updates that propagate up to the internal nodes are handled similarly.

The ability to support updates means that PB$^+$-tree can be directly employed in tandem with the node migration techniques from [23], [24] to prevent the adversary from tracking the accesses to a node over time, just like data shuffling in oblivious RAM [4], [20]. As we will see in Section VII, PB$^+$-tree requires the adversary to accumulate a much longer I/O history, relative to the encrypted B$^+$-tree, before the leaf nodes can be sequenced correctly. This allows node migrations to be triggered only sporadically so as to minimize the resulting communication and I/O overheads, without compromising security.

VI. PB$^+$-TREE: COUNTERING ACCESS TRACKING

Having presented the PB$^+$-tree, we now explain how it remains effective even if the search key values are not accessed uniformly, and the adversary manages to acquire knowledge of the access frequency distribution. Access frequencies are not to be confused with Dist; they refer to how frequently the various key values are accessed by queries, whereas Dist refers to how the values of the search key attribute are distributed in its domain.

To exploit knowledge of the access frequencies, the adversary tracks the access counts of the PB$^+$-tree buckets over time. The observed access frequencies can then be matched with the expected access frequencies of various search key values. For example, suppose that the heap file in Figure 1 stores the records of a university's alumni, with the "years_since_graduation" attribute as search key. If fresh graduates are expected to be inactive, their records (with key value 1) will be retrieved very rarely. This knowledge allows the adversary to deduce that key value 1 is likely to map to the bucket with lowest observed access frequency.

To effectively counter this attack, we need to ensure that the buckets in each index level as well as in the heap file have roughly the same access frequency. This implies that our node-to-bucket assignment cannot be random, and must instead even out the summed access frequencies across buckets. Consider the n encrypted leaves of the PB$^+$-tree, and let $f(N)$ denote the expected access frequency of leaf N. Formally, our node assignment problem is to pack the n encrypted nodes into $\lceil n/b \rceil$ buckets, each with a capacity of b nodes. With N_{ij} denoting the node assigned to the j-th slot of bucket B_i, the aggregate access frequency of B_i is $f(B_i) = \sum_{j=1}^{b} f(N_{ij})$. The node assignment $\mathbf{B}$ should minimize the objective function (i.e., the spread)

$$\Psi(\mathbf{B}) = \overline{f}(\mathbf{B}) - \underline{f}(\mathbf{B}) \tag{7}$$

where $\overline{f}(\mathbf{B}) = max_i\{f(B_i)\}$ is the highest bucket frequency, and $\underline{f}(\mathbf{B}) = min_i\{f(B_i)\}$ the lowest.

Our node assignment problem is closely related to the *balanced number partitioning* problem in complexity theory: Given n numbers, the objective is to group them into m partitions so as to minimize the largest partition sum, subject to the constraint that each partition should hold either $\lceil n/m \rceil$ or $\lfloor n/m \rfloor$ numbers. The problem is NP hard, and only approximate solutions are possible for arbitrary number of partitions m [28]. The most effective heuristic algorithm is the Balanced Largest Differencing Method (BLDM), first proposed in [29] for $m = 2$ and subsequently generalized to $m \geq 2$ in [30]. PB$^+$-tree employs the BLDM method of [30], as described below.

We first add fictitious nodes with zero access frequencies in the pool of actual PB$^+$-tree leaves, so that their total number becomes $n = mb$ for some positive integer m. Then, we sort them in ascending frequency order. Denoting the sorted frequencies by $f_1, f_2, \ldots, f_n$, the sequence is divided[5] into b m-tuples, each of the form $F_i = [f_{(i-1)m+1}, f_{(i-1)m+2}, \ldots, f_{im}]$, for $1 \leq i \leq b$. The differential $\delta(F_i)$ of an m-tuple F_i is the difference between its largest and smallest frequency.

Next, the m-tuples are *folded* iteratively to produce the final buckets: Two m-tuples F_α and F_β are *folded* by combining/summing the first frequency in F_α with the last in F_β, the second frequency in F_α with the penultimate in F_β, and so on. In each iteration, we fold the two m-tuples with the largest differentials until only one tuple remains. In the final tuple $F_\gamma = [f_{\gamma,1}, f_{\gamma,2}, \ldots, f_{\gamma,m}]$, each $f_{\gamma,i}$ is the sum of b of the initial node access frequencies; moreover, the various $f_{\gamma,i}$'s ($1 \leq i \leq m$) are expected to be similar. We thus assign to one bucket the b nodes whose access frequencies contribute to each $f_{\gamma,i}$, leading to m equally sized buckets with similar aggregate access frequencies.

Discussion: There is a subtle difference in objective function between our node assignment problem and balanced number partitioning. To illustrate, consider two candidate assignments into $m = 3$ buckets, the first yielding aggregate frequencies 12, 9, 9, and the second 12, 10, 8. The first assignment is preferable for our problem formulation (according to objective function $\Psi(\mathbf{B})$); however, both assignments are equally favorable in balanced number partitioning. Despite this difference, our experiments in Section VII show that the node assignments generated by BLDM are fully adequate for our PB$^+$-tree configuration.

Another concern is that the access frequencies may be so skewed that there is no node assignment that can even out the bucket frequencies. For example, one of the PB$^+$-tree nodes may receive a disproportionately high number of accesses. To lessen the problem, we may replicate this node and, essentially, spread its accesses across the replicas. With enough replicas, we will be able to suppress the access frequency of each of them and enable a more balanced assignment (over the replicas and the remaining nodes). As this problem is not the main focus of our work, we leave a detailed solution to future work.

[5] For ease of presentation, in the context of partitioning we refer to frequencies and nodes interchangeably.

Parameter	Description	Default
d	Number of records in the index	2 million
$qlen$	Query length (# records)	[250,16000]
$\#queries$	Number of queries	20000
b	Bucket capacity (nodes/bucket)	4

TABLE II
DEFAULT PARAMETERS FOR SYNTHETIC WORKLOAD

Finally, the node access frequencies may drift over time. If the adversary is aware of these changes, and they lead to significantly higher or lower access frequencies in certain buckets, the user will want to re-balance the bucket frequencies. This need not entail a costly reorganization of the entire PB$^+$-tree. Instead, the user could simply re-assign nodes among the affected buckets only. Moreover, to prevent the adversary from inferring exactly which buckets participated in the re-assignment, the user may additionally re-encrypt some randomly chosen "victim" buckets (so that they appear to be updated too).

VII. EMPIRICAL EVALUATION

In this section, we empirically evaluate the encrypted B$^+$-tree [6] and our PB$^+$-tree in light of the adversarial tools provided in the paper.

A. Experiment Set-Up

We first describe the experiment set-up. Varied parameters and their default values are summarized in Table II.

Indexing schemes: For brevity, we denote the PB$^+$-tree by PB. For the encrypted B$^+$-tree, we include both the sibling traversal (Section IV-A) and subtree retrieval (Section IV-B) strategies, denoted by ST and SR respectively. For ST and SR, the block addresses of the index nodes accessed by the queries are visible to the adversary. For PB, the adversary can observe which buckets are retrieved but not the exact nodes. The inference attack by the adversary proceeds according to the stitching procedure in Section IV-A for ST, Algorithm 1 for SR, and Algorithm 2 followed by Algorithm 1 for PB.

Synthetic workload: By default, our experiments are conducted with a synthetic workload, which allows us to control the parameters of the data and query sets. We create a heap file $\mathbf{R}$ containing d records, each 256 bytes in size including an 8-byte integer key, and build an unclustered B$^+$-tree index over $\mathbf{R}$. For ST and SR, the node size is the same as the block size of the file system (4 Kbytes), while each node and record is encrypted with AES [31]. For PB, a bucket occupies one block so each node is allotted $1/b$ of a block. We make the decision to use the same block size for PB and its competitors for fairness. The PB nodes are encrypted with the BGN scheme [8]; the records within each data block of $\mathbf{R}$ are also encrypted with BGN.

The workload consists of $\#queries$ range selections. By default, the query length, i.e., the number of records retrieved by the query, varies uniformly between 250 and 16,000. This query length ensures that each range spans at least two leaf nodes (of the encrypted B$^+$-tree), so that the access patterns can be used to sequence the leaf nodes. The uniformly distributed query length has a large variance (relative to, for example, Normal distribution and fixed length), hence it is the most discriminative against PB and favors its competitors[6].

TPC-H workload: To confirm our findings, we also experiment with TPC-H (http://www.tpc.org/tpch), a standard decision support benchmark. Using the `Lineitem` table which consists of 6 million records, we construct encrypted B$^+$-tree and PB$^+$-tree indexes. We then collect 80,000 instances of the range query Q14 (i.e., the Promotion Effect Query) to run against the three methods; the query lengths vary from 100 to 82,050 records.

System configuration: The server runs Windows Server 2003 and is equipped with an Intel Core 2 Duo 3.0 GHz CPU with a 6 Mbyte cache, and an ST3320813AS hard disk. The user machine is a notebook computer with an Intel 1.33GHz CPU that connects by a gigabit network switch to the server.

Performance factors: Our evaluation centers on the following metrics: (i) the privacy leak PL, defined in Section III; (ii) the attack time, which quantifies the processing effort required by the adversary to infer the leaf node ordering; and (iii) the query response time, including the I/O and CPU costs to answer range selections.

B. Index Construction Cost

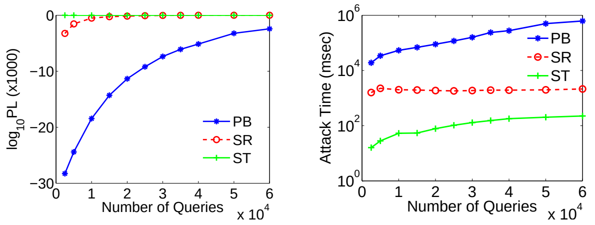
We begin by examining the index construction cost. With the default settings in Table II, building the encrypted B$^+$-tree along with the underlying encrypted heap file incurs 1337.74 seconds of I/O and 31.58 seconds of computation, summing to a total construction time of 1369.32 seconds. For PB the construction time is 1655.59 seconds, including 1339.29 and 316.30 seconds of I/O and CPU cost. In terms of I/O, PB is slightly slower than the encrypted B$^+$-tree because the smaller fanout of the former leads to a larger number of index nodes. On the other hand, the difference in CPU time between the two indices is more significant. This is because the PB nodes require BGN encryption, which is costlier than the AES scheme for the encrypted B$^+$-tree.

C. Sensitivity to the Number of Queries

To examine the privacy strength and performance of the encrypted B$^+$-tree (ST and SR) and PB$^+$-tree, we vary $\#queries$ while keeping the remaining parameters at their default values in Table II. Figures 5(a) and 5(b) present the $\log_{10}PL$ values, and the attack time in logarithmic scale. Each reported value is the average across 1000 executions.

The results show that ST quickly approaches a PL of 1 (in Figure 5(a)), indicating a total exposure of the leaf node ordering. Moreover, the security breach is achieved at a very low cost to the adversary, with attack time ranging from 16 msec to 224 msec (Figure 5(b)). Avoiding sibling traversal in favor of subtree retrieval provides only marginally better privacy: SR approaches a PL of 0.5 (i.e., where all of the leaf nodes are completely sequenced as explained in Section IV-B2) after only around 10 thousand queries, and

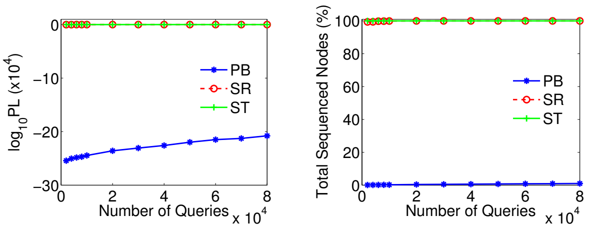
[6] As explained in Section V-C, the adversarial tool in Algorithm 2 takes advantage of length variations among access patterns, by resolving the shorter, more certain overlaps before tackling longer ones. A high variance in query length thus benefits the adversary. Detailed results for different distributions are reported in [26], but omitted here due to space constraint.



(a) Privacy Leak

(b) Attack Time

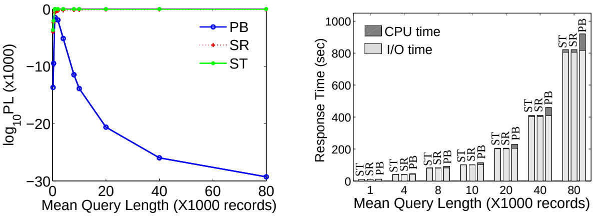
Fig. 5. Sensitivity to the Number of Queries



(a) Privacy Leak

(b) % Sequenced Nodes

Fig. 6. Experiment with Q14 Queries in TPC-H Benchmark



(a) Privacy Leak

(b) Query Response Time

Fig. 7. Sensitivity to the Query Length

the attack time ranges from 1.58 to 2.13 seconds. These results confirm the vulnerability of the encrypted B$^+$-tree.

Compared to the encrypted B$^+$-tree, PB offers much stronger privacy protection. At 60 thousand queries, the cost incurred by the adversary to analyze the observed access patterns (i.e., the attack time) is almost three orders of magnitude higher than for SR. Even then, the privacy leak remains below $PL = 10^{-2427}$, indicating a considerable degree of uncertainty in the leaf node ordering.

Of course, the superior security of PB is achieved at the expense of higher processing overheads. The average query response time, which is independent of $\#queries$, is 82.20 seconds for ST, 82.21 seconds for SR, and 92.02 seconds for PB. The slightly longer response time of SR over ST is due to the former retrieving the entire subtree that covers the query range in the encrypted B$^+$-tree, which however is typically small. PB is slower primarily due to the BGN cryptographic computations (by the server and the user), with total computation cost accounting for 13% of PB's response time and the remaining 87% attributed to I/O cost. The I/O penalty is low because many necessary internal and leaf nodes share common buckets. As explained in Section V-A, PB also follows the subtree retrieval paradigm, thus all the nodes in each level are requested together and processed in order of physical address $\langle$bucket#, slot#$\rangle$; this allows the nodes that reside in the same bucket to be fetched with a single I/O operation. Overall, PB is only 15% slower than a plain (unprotected) B$^+$-tree, which takes 80.42 seconds. This confirms that PB's privacy comes without significant performance sacrifice.

To confirm the generality of our findings, we repeat the experiment with the TPC-H workload. Figure 6 shows the corresponding PL metric and percentage of sequenced nodes. ST and SR continue to be vulnerable. In contrast, PB achieves even stronger protection in this workload owing to two factors: (i) With a larger number of records, the TPC-H data generate a PB$^+$-tree with more nodes that is harder to sequence correctly from the bucket access patterns. (ii) The query lengths here are longer than optimal for sequencing (Section V-C); the next section scrutinizes this effect.

D. Sensitivity to the Query Length

To study the sensitivity of the three methods (ST, SR and PB) to the query length, we vary the upper bound of $qlen$ from 600 to 160,000 records while keeping the lower bound at 250 records. In other words, the range of $qlen$ widens from [250,600] to [250,160000]. We continue to keep $qlen$ uniformly distributed within each range. The other parameters remain at their default settings.

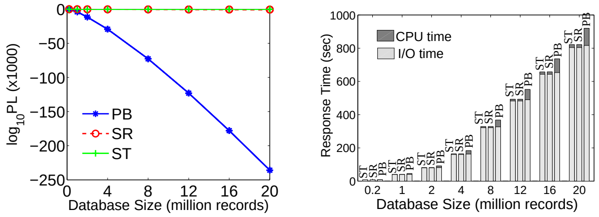
According to the results in Figure 7, ST and SR again approach quickly their maximum PL (of 1 and 0.5 respectively). Longer queries produce access patterns that cover more leaf nodes, allowing for more effective sequencing.

On the other hand, Figure 7(a) shows that the PL of PB initially increases with the query length, peaks at about $qlen = 1000$, and then drops. This evidence corroborates the analysis in Section V-C, which indicates that the inferences of the adversary are most accurate when they are based on access patterns that overlap by two to five leaf nodes. The number of such access patterns increases with $qlen$ initially, which explains PB's deterioration in PL. However, even when PB is at its weakest (around $qlen = 1000$), it achieves $PL = 10^{-1497}$ compared to ST's $10^{-2.8}$ and SR's 10^{-544}. As $qlen$ increases further, the overlap between access patterns also gradually rises beyond the optimal range for sequencing and impedes the inference algorithm of the adversary. The attack time (not shown) of PB is consistently at least two orders of magnitude longer than those of ST and SR. The response times of PB are consistent with the observations in Section VII-C, i.e., PB is about 15% slower than SR. Note that the CPU portion of the bars in Figure 7(c) reflects the total computation times at the server and the user.

E. Sensitivity to the Database Size

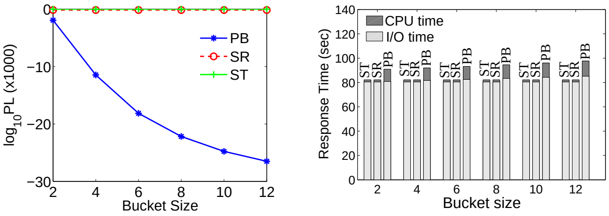
Next, we examine how the schemes scale with the database size d. A larger d increases the number of leaf nodes in the index, so the adversary accordingly requires more access patterns in order to sequence the nodes. Therefore, we scale $\#queries$ to maintain a ratio of 1:100 with d. We also increase the upper bound of the query length to maintain the same selectivity factor; for example, at 20 million records, the query length is between $[250, 160000]$. The remaining parameters are set to their default values.

The results, presented in Figure 8, show that the privacy level of ST and SR is insensitive to d, as long as $\#queries$ increases proportionally to d. This is not the case for PB,



(a) Privacy Leak (b) Query Response Time

Fig. 8. Sensitivity to the Database Size



(a) Privacy Leak (b) Query Response Time

Fig. 9. Sensitivity to the Bucket Capacity

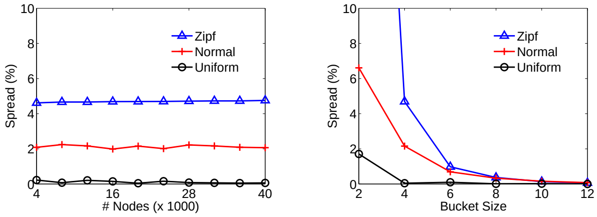
which enjoys a descent in PL. The reason is that a larger $\#queries$ does not lead to an equivalent increase in the combined coverage of the overlap between pairs of access patterns, which determines the effectiveness of Algorithm 2. The query response time, dominated by I/O time, can be speeded up by buffering the frequently accessed nodes or employing a disk array to retrieve data in parallel.

F. Sensitivity to the Bucket Capacity

To study PB's sensitivity to the bucket capacity, in Figure 9 we vary b while using default settings for the other parameters. Although b is irrelevant to ST and SR, they are included in the charts for comparison. Since PB achieves privacy by masking the node accesses within buckets, we expect a larger bucket capacity b to significantly improve PB's security. This is confirmed in Figures 9(a). At the same time, Figure 9(b) shows that a larger b increases moderately the query processing overhead; as each bucket is constrained by the block size of the file system, a larger b reduces the node size, leading to a larger index structure and thus some extra I/Os for query processing. The response time of PB is 10.6% to 18.8% longer than that of SR, as b varies from 2 to 12. Essentially, b determines the tradeoff between performance and privacy in PB.

Combined with the previous experiment, these results indicate that for a small database, we may pick a higher bucket capacity to ensure strong privacy. For larger databases, PB is likely to be sufficiently secure even with low node-to-bucket ratios, so we may set b=2 to boost query processing performance.

G. Balanced Bucketization for Known Access Frequencies



(a) $b = 4$, Vary n (b) $m = 5000$, Vary b

Fig. 10. Node-to-Bucket Assignment with BLDM

Here we study the effectiveness of the PB$^+$-tree adaptation for cases where the adversary possesses a priori knowledge of the search key access frequencies; i.e., we evaluate how successfully it evens out the bucket access frequencies. As explained in Section VI, we employ the BLDM method of [30] to group the PB$^+$-tree nodes into buckets. We focus on the leaf level of the tree, as it provides the finest granularity within the index (see Observation 1).

First, we set the bucket capacity $b = 4$, and vary the number of leaf nodes n from 4000 to 40000 (i.e., the number of resulting buckets m varies from 1000 to 10000). We experiment with three different distributions for the node access frequencies – uniform distribution[7] between 0 and $\frac{2}{n}$, normal distribution with a mean of $\frac{1}{n}$ and a standard deviation of $\frac{1}{4n}$, and Zipf distribution with a 0.4 exponent. We quantify the effectiveness of a node assignment $\mathbf{B}$ by the normalized spread, defined as $\Psi(\mathbf{B})$ (the difference between the highest and lowest bucket frequencies) divided by the mean bucket frequency. For each n, we carry out 100 trials and report the average normalized spread. The results, summarized in Figure 10(a), show that even with a very skewed distribution like Zipf, the normalized spread can be consistently kept below 5%.

Next, we fix $m = 5000$ and vary b from 2 to 12. Figure 10(b) plots the results. Here, the normalized spread can be very high at $b = 2$; in fact, for the Zipf distribution we obtain a measurement of 49%. However, it drops sharply with increasing b. For $b \geq 6$, we achieve less than 1% normalized spreads for all three distributions.

This experiment shows that if the adversary knows the expected search key access frequencies, we should configure the PB$^+$-tree with $b \geq 4$. With that bucket size, the normalized spread between buckets can be suppressed sufficiently to deter the adversary from differentiating between buckets through their access counts.

H. Summary of Experiment Results

Our evaluation confirms that, applying the adversarial tools provided in this paper, the privacy leak PL of ST and SR rises to its maximum of 1 and 0.5 respectively, almost as soon as the queries cover all the leaf nodes. In contrast, PB keeps PL at very low levels even for large numbers of query execution traces (60 thousand or more in our experiments), while incurring a manageable 15% query processing overhead. This implies that PB$^+$-tree offers a secure and practical long-term solution that does not require frequent node migration (discussed in Section V-D). Additionally, unlike its competitors, PB's security gets stronger for larger databases. One of its key properties is that the tradeoff between privacy and query performance can be effectively controlled by the user (via

[7] Here "uniform" implies that the access frequency of each node is equally likely to receive any value in a specified interval (not that all node frequencies are equal).

the bucket capacity), thus providing flexibility and ensuring wide applicability. Finally, PB is able to balance its bucket access frequencies to prevent the adversary from exploiting a priori knowledge of the search key access frequencies.

VIII. Conclusion

This paper studies the problem of protecting the key scope of range queries that are executed on untrusted database servers. We show that merely encrypting the nodes of a standard B$^+$-tree index is not secure, as an adversary can sequence its leaf nodes and deduce the key range of each accessed node. As remedy, we introduce the privacy-enhancing B$^+$-tree (PB$^+$-tree) that conceals the exact node addresses from the adversary, by grouping them into buckets and by employing homomorphic encryption techniques to retrieve them from their host buckets. Extensive experiments confirm the effectiveness and practicality of the PB$^+$-tree.

An interesting extension is to multi-dimensional structures, like the R-tree. Here, an adversary could utilize the observed access patterns to deduce the relative positions of the encrypted R-tree nodes in the data space; this is analogous to sequencing the B$^+$-tree leaf nodes in one-dimensional space.

Acknowledgment

This project was supported by the Singapore National Research Foundation under its International Research Centre @ Singapore Funding Initiative and administered by the IDM Programme Office.

References

[1] H. Hacigumus, B. Iyer, and S. Mehrotra, "Providing Database as a Service," in *IEEE ICDE*, 2002.

[2] Computer Security Institute, "CSI/FBI Computer Crime and Security Survey," 2008.

[3] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private Information Retrieval," in *IEEE FOCS*, 1995.

[4] O. Goldreich and R. Ostrovsky, "Software Protection and Simulation on Oblivious RAM," *JACM*, vol. 45, no. 1, 1996.

[5] B. Pinkas and T. Reinman, "Oblivious ram revisited," in *CRYPTO*, 2010.

[6] E. Damiani, S. C. di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati, "Balancing Confidentiality and Efficiency in Untrusted Relational DBMSs," in *ACM CCS*, 2003.

[7] D. Comer, "Ubiquitous B-Tree," *ACM Computing Surveys*, vol. 11, no. 2, 1979.

[8] D. Boneh, E.-J. Goh, and K. Nissim, "Evaluating 2-DNF Formulas on Ciphertexts," in *Theory of Cryptography*, 2005.

[9] R. Rivest, L. Adleman, and M. Dertouzos, "On Data Banks and Privacy Homomorphisms," in *Foundations of Secure Computation*, 1978.

[10] N. Ahituv, Y. Lapid, and S. Neumann, "Processing Encrypted Data," *CACM*, vol. 30, no. 9, 1987.

[11] D. X. Song, D. Wagner, and A. Perrig, "Practical Techniques for Searches on Encrypted Data," in *IEEE S&P*, 2000.

[12] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky, "Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions," in *ACM CCS*, 2006.

[13] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano, "Public Key Encryption with Keyword Search," in *EUROCRYPT*, 2004.

[14] D. Boneh, E. Kushilevitz, R. Ostrovsky, and W. E. S. III, "Public Key Encryption that Allows PIR Queries," in *CRYPTO*, 2007.

[15] M. Bellare, A. Boldyreva, and A. O'Neill, "Deterministic and Efficiently Searchable Encryption," in *CRYPTO*, 2007.

[16] L. Bouganim and P. Pucheral, "Chip-Secured Data Access: Confidential Data on Untrusted Servers," in *VLDB*, 2002.

[17] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Order Preserving Encryption for Numeric Data," in *ACM SIGMOD*, 2004.

[18] H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra, "Executing SQL over Encrypted Data in the Database-Service-Provider Model," in *ACM SIGMOD*, 2002.

[19] H. Wang and L. V. Lakshmanan, "Efficient Secure Query Evaluation over Encrypted XML Databases," in *VLDB*, 2006.

[20] P. Williams, R. Sion, and B. Carbunar, "Building Castles Out Of Mud: Practical Access Pattern Privacy and Correctness on Untrusted Storage," in *ACM CCS*, 2008.

[21] P. Williams and R. Sion, "Usable PIR," in *NDSS*, 2008.

[22] S. Papadopoulous, S. Bakiras, and D. Papadias, "Nearest Neighbor Search with Strong Location Privacy," *PVLDB*, vol. 3, no. 1-2, 2010.

[23] P. Lin and K. S. Candan, "Hiding Traversal of Tree Structured Data from Untrusted Data Stores," in *WOSIS*, 2004.

[24] D. T. Khanh, "Oblivious Search and Updates for Outsourced Tree-Structured Data on Untrusted Servers," *IJCSA*, vol. 2, 2005.

[25] "File System Filter Drivers," http://www.microsoft.com/whdc/driver/filterdrv/default.mspx.

[26] H. Pang, J. Zhang, and K. Mouratidis, "Enhancing Access Privacy of Range Retrievals over B$^+$-Trees," in *Technical Report, School of Information Systems, Singapore Management University*, 2011.

[27] T. Arnold and L. V. Doorn, "The IBM PCIXCC: A New Cryptographic Coprocessor for the IBM eServer," *IBM Journal of Research and Development*, vol. 48, May 2004.

[28] M. Dell'Amico and S. Martello, "Bounds for the cardinality constrainted $p||c_{max}$ problem," *Journal of Scheduling*, vol. 4, 2001.

[29] B. Yakir, "The Differencing Algorithm LDM for Partitioning: A Proof of a Conjecture of Karmarkar and Karp," *Mathematics of Operations Research*, vol. 21, no. 1, pp. 85–99, 1996.

[30] W. Michiels, J. H. M. Korst, E. H. L. Aarts, and J. van Leeuwen, "Performance Ratios for the Differencing Method Applied to the Balanced Number Partitioning Problem," in *STACS*, 2003.

[31] AES, "Advanced Encryption Standard," *National Institute of Science and Technology*, 2001.

HweeHwa Pang received the BSc (first class honors) and MS degrees from the National University of Singapore in 1989 and 1991, respectively, and the PhD degree from the University of Wisconsin-Madison in 1994, all in Computer Science. He is a Professor at the School of Information Systems, Singapore Management University. Prior to that, he was a Principal Scientist and Division Director at the A*Star Institute for Infocomm Research. His current research interests include database management systems, data security, and information retrieval.

Jilian Zhang received the BSc degree from Northwestern Polytechnical University in 2003, and the MSc degree from Guangxi Normal University in 2006. Since 2007 he has been a PhD candidate in the School of Information Systems, Singapore Management University, under the supervision of professors HweeHwa Pang and Kyriakos Mouratidis. He has worked on query authentication for outsourced databases, database privacy, and spatial databases.

Kyriakos Mouratidis is an Assistant Professor at the School of Information Systems, Singapore Management University. He received his BSc degree from the Aristotle University of Thessaloniki, Greece, and his PhD degree in Computer Science from the Hong Kong University of Science and Technology. His main research area is spatial databases, with a focus on continuous query processing, road network databases and spatial optimization problems. He has also worked on preference-based queries, wireless broadcasting systems, and certain database privacy topics.