Access Pattern Confidentiality-Preserving Relational Databases: Deployment Concept and Efficiency Evaluation

Alexander Degitz, Jens Köhler, Hannes Hartenstein
Karlsruhe Institute of Technology (KIT) - Steinbuch Centre for Computing & Institute of Telematics
Karlsruhe, Germany
{alexander.degitz, jens.koehler, hartenstein}@kit.edu

ABSTRACT

In this paper, we address the question whether access pattern confidentiality-preserving databases with an underlaying B-tree index structure are feasible in practice by proposing integrative deployment concepts that support important database query functionalities based on ORAM and shuffled B-trees. Furthermore, we provide a rigorous efficiency evaluation to determine the cost of the proposed concepts with regard to storage, network, and query latency as well as to investigate the influence of scenario factors like database size, number of records, and network bandwidth. In particular, we show that ORAM-based concepts only cause an overhead of factor 5.9 for evaluating equality conditions on a database with up to 10 million records.

Categories and Subject Descriptors

I.8 [**Database and storage security**]: Management and querying of encrypted data; H.2.4 [**Information systems**]: Information storage systems; Cloud based storage

1. INTRODUCTION

Many potential cloud customers are reluctant to make use of Database-as-a-Service (DaaS) offerings and to outsource confidential data to cloud storage providers (SPs) due to the possibility that the SPs do not behave honestly or do not apply sufficient protective measures to protect the data against third party attackers. Enforcing the confidentiality of data *before* it is outsourced to the SP mitigates the risk of a data confidentiality breach. With the advent of the internet of things, it is expected that users are not even aware of *who* will store their data anymore [2]. Thus, techniques that enforce data confidentiality in the trusted domain of the user before the data is outsourced will gain even more importance. For the sake of readability and without loss of generality, we abstract from the specific nature of the attacker and assume that the SP aims to breach data confidentiality in the following. Outsourcing confidential relational databases is a special case of confidential data outsourcing as the outsourced data does not only have to be protected but also has to be efficiently searchable. To achieve that, a variety of confidentiality preserving indexing approaches (CPIs) were proposed that allow to efficiently evaluate database queries on outsourced encrypted data [12]. Most existing CPIs only protect the confidentiality of data at rest, i.e., data that is persisted by the SP. However, in reality, the SP is also able to monitor database queries and the encrypted query results, as well as database record inserts, updates, and deletions. Such observations can be used to reveal confidential data. For instance, if the SP observes that two encrypted records 1 and 2 are the result of a query for all records which contain an attribute value X, most CPIs guarantee that the plaintext attribute value X is hidden from the SP. However, the SP can deduce that the two returned records contain the *same* attribute value. The SP might then be able to apply background knowledge like "record 1 contains attribute value X" to learn that "record 2 contains attribute value X".

In particular, to protect against SPs that can monitor queries, a CPI has to guarantee that the SP is not able to determine which encrypted records matched an executed query and the SP is not able to correlate executed queries (access pattern confidentiality). Both properties can be trivially achieved by retrieving the entire database for each query to make the queries indistinguishable. This induces significant efficiency costs in terms of latency, transmission, and computation. Oblivious RAM (ORAM) approaches [5, 9, 13, 4, 15] allow to store and retrieve records based on fixed identifiers and enforce pattern and access confidentiality by selectively retrieving, re-encrypting, and re-submitting specific parts of the outsourced database in an interactive way. However, ORAM schemes do not support the search capabilities that are required by relational databases and, as of now, it is unclear whether their efficiency is acceptable for relational database scenarios[1].

In this paper, we investigate the question whether preserving access **PAT**tern **CONF**identiality in relational **D**ata**B**ases is practically feasible by proposing PATCONFDB, a deployment concept for ORAM schemes in the database context that supports common database searches and provides pattern and access confidentiality. We show that PATCONFDB provides confidentiality guarantees against honest-but-curious attackers who can view the persisted data *and* monitor the executed database queries, including record insert, update, as well as delete operations. We show that by

[1]http://outsourcedbits.org/2013/12/20/how-to-search-on-encrypted-data-part-4-oblivious-rams/

making well-considered use of ORAM protocols, it is possible to achieve significant efficiency benefits compared to retrieving the entire database for each query. Furthermore, we compare the efficiency of PATCONFDB to previously proposed shuffled B-tree approaches [7, 17, 3] which also aim to provide access pattern confidentiality, but are not able to provide strict security guarantees. In order to make a shuffled B-tree approach comparable, we extend it to support various query types that are prevalent in database query workloads. The main contributions of this paper are:

- PATCONFDB, a **concept to efficiently apply ORAM protocols in DaaS settings** that supports searching a database for records which match specified equality, range, and prefix query conditions.
- A **concept to enhance shuffled B-tree approaches** to be used in DaaS settings, so that update operations as well as range and prefix queries can be executed.
- An **efficiency evaluation** that shows that significant performance increases are possible by considerately applying ORAM protocols. Furthermore, we compare the PATCONFDB approach to the less secure shuffled B-tree approach and provide **guidelines for choosing the most suitable indexing approach for a given outsourcing scenario**.

In Section 2 related work is discussed. In Section 3, the specific requirements of relational databases for deployment concepts of access pattern confidentiality-preserving schemes are summarized. In Section 4, we introduce PATCONFDB, an ORAM-based access pattern confidentiality-preserving indexing approach, and show how shuffled B-tree indexes can be extended to also satisfy the requirements. We provide an efficiency evaluation of the proposed schemes in Section 5. Furthermore, we discuss possible functionality extensions for our deployment concepts Section 6 and conclude the paper in Section 7.

2. RELATED WORK

Many protocols have been proposed to efficiently perform keyword searches over encrypted data, i.e. [11, 1, 12]. While these approaches ensure content confidentiality, they leak the access patterns of queries. Based on an exemplary database, Islam et al. [10] showed that, with a small amount of background knowledge about the data, up to 80% of the encrypted data can be revealed by an attacker who watches the query and the corresponding results through a frequency analysis. This brought forward the need for CPIs that also ensure access pattern confidentiality.

Oblivious RAM (ORAM) was first proposed by Goldreich and Ostrovsky as a way to ensure software protection [8]. ORAM prevents that an attacker who observes the RAM learns any information about the RAM access patterns of executed programs. Improved schemes were proposed over the last few years which have put a focus on ORAM to be a considerable alternative when it comes to secure data outsourcing [5, 9, 13, 4, 15], i.e., storing and retrieving data records based on fixed identifiers. Outsourced relational databases, on the other hand, are more complex and require efficient search for database records that contain a specific attribute value or contain attribute values that fall in a specific range. ORAM schemes do not support such search queries and, to the best of our knowledge, it is unclear whether applying ORAM to encrypt relational databases is feasible with regard to efficiency. In this paper, we do not propose a new ORAM scheme, but PATCONFDB, a concept on how to use existing ORAM schemes to enable the execution of search queries on relational databases.

Encrypted B-trees [3] were proposed to enforce the confidentiality of data at rest. To increase the computational cost of inference attacks based on access patterns, shuffling the B-tree after each database query was proposed [6, 7, 17]. Note that, despite making access pattern based inference attacks harder, the Shuffle Index does not provide strict security guarantees like ORAM does. Furthermore, to the best of our knowledge, no approach based on shuffled B-trees provides the ability to insert, update or delete data. In this paper, we extend the Shuffle Index approach [6] to support them. Our extensions can be applied to all schemes that make use of a shuffled B-tree.

3. DEPLOYMENT REQUIREMENTS

3.1 Database Functionality Requirements

Databases typically support **inserts**, **updates**, and **deletions** of database records. Furthermore, typically it is required to search databases for specific records. The structured query language (SQL) specifies a variety of different query types. In the following, we consider the following types of queries:

1. **Equality selection:** Query for records that contain a specific attribute value A.
   *Example: SELECT * WHERE Name = 'Andy';*
2. **Prefix selection:** Query for records containing an attribute value that starts with prefix A.
   *Example: SELECT * WHERE Name LIKE 'An%';*
3. **Range selection:** Query for records that contain an attribute value that is smaller than value A and bigger than value B.
   *Example: SELECT * WHERE Name Between 'An' AND 'Ce';*

We discuss the challenges of providing access and pattern confidentiality for additional query types in Section 6.

3.2 Confidentiality Requirements

An approach to securely outsource confidential databases has to enforce both *content* and *access pattern confidentiality* to protect against attackers that are able to monitor queries on the outsourced data.

Content confidentiality: A database outsourcing approach provides content confidentiality if an attacker that is able to view the outsourced data is not able to learn the content of the database's records.

Access pattern confidentiality: We adopted our definition of access pattern confidentiality from [16]. An approach provides access pattern confidentiality if an attacker that monitors database queries and query results as well as database insert, update, and deletions is not able to tell 1) which parts of the database were accessed by a database operation, 2) when a part of a database was last updated, 3) whether specific data was repeatedly accessed, and 4) whether the database was queried or updated. This particularly means that even a series of accesses to the same record is unrecognizable for attackers and therefore does not leak any information.

4. DEPLOYMENT CONCEPTS

4.1 PATCONFDB

In this section, we introduce the PATCONFDB concept that uses existing ORAM schemes as building blocks to support searches on outsourced data while enforcing access pattern confidentiality. The interface of all existing ORAM schemes is *ORAM.get(ID)* and *ORAM.put(ID, block)*, i.e., data blocks can be stored and retrieved based on a fixed ID. All existing ORAM schemes share the following characteristic. To retrieve a stored data block, first a set of encrypted data block containers (DC) has to be downloaded and decrypted. The decrypted data blocks then have to be re-encrypted and the resulting encrypted DCs have to be re-uploaded to the SP. PATCONFDB is agnostic to the underlying ORAM scheme and initializes multiple instances of the used ORAM scheme. These *ORAM instances* are independent from each other. Each ORAM instance hides the content and the access pattern of the database from the SP.

In most database scenarios, records have to be searchable based on record attributes. This creates the need for indexes to be outsourced to an external SP. We use a B-tree index structure for PATCONFDB so that the efficiency evaluation is better comparable to the Shuffle Index, which also uses a B-tree. To search for records that contain a specific attribute value, the highest level node of the B-tree is retrieved. The retrieved node can be used to determine which node of the next lower B-tree layer is closest to the queried value. This process is repeated until the leaf of the B-tree is reached, which contains the record identifiers of the records that contain a matching attribute value.

Each node of a B-tree can be encrypted to protect the confidentiality of the data at rest. However, the attacker would still learn access patterns and distinguish different types of queries by monitoring which nodes were retrieved to evaluate a given query. To hide which nodes were retrieved, PATCONFDB stores every layer of the B-tree in a separate ORAM instance as shown in Figure 1. For the remainder of this paper we call the ORAM instances that contain the *index layers* and the ORAM instance that contains the actual data records *record store*. In the following, we describe how PATCONFDB supports the database functionality that is described in Section 3.

Equality selections. To retrieve all records that contain a specific attribute value, first the root node of the B-tree has to be retrieved and decrypted. Based on the decrypted node it can be determined which node X has to be retrieved from the next lower layer in the B-tree. To retrieve X, the ORAM instance that corresponds to X's layer has to be queried. If the B-tree is stored in n ORAM instances, n ORAM instances have to be accessed to retrieve the B-tree's leaf node that contains the identifier of the records that match the database query. Based on the record identifiers, the matching records can be retrieved from the record store ORAM instance. If more than one record matches the query, the equality selection has to be evaluated again for each matching record before retrieving it from the record store, even if all identifiers of matching records are already known. This mechanism is needed to keep queries indistinguishable, as explained in Section 4.2.

Prefix / range selections. To keep the type of queries indistinguishable, a prefix selection is executed as a sequential series of equality selections through the same mechanism as described for equality selections. That is, for each attribute value that lies in the queried range, an equality selection query is evaluated and the results of these queries are aggregated to the result of the range/prefix selection. A prefix selection can be considered as a special kind of range selection, that spans over every attribute value of records inside the queried prefix.

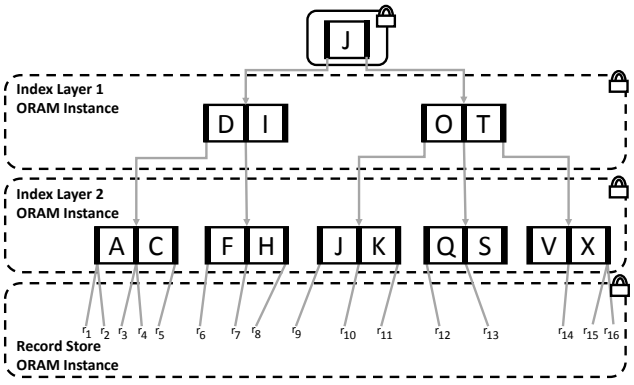


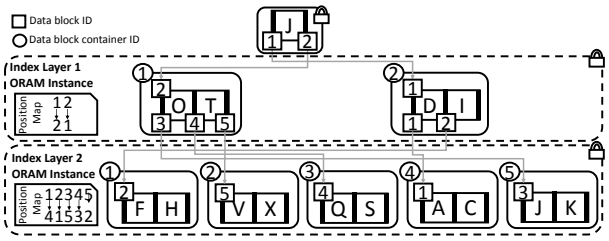
Figure 1: Hierarchy of the ORAM instances in PATCONFDB.

Insert / delete / update. To insert a new record, an equality selection for the attribute value of the new record is performed, the set of identifiers of records that contain the attribute value is retrieved and the identifier of the inserted record is added before re-encrypting and re-uploading the data to the corresponding index layers. Furthermore, the records that matched the equality selection are retrieved from the record store ORAM instance and the new record is inserted in the record store when re-encrypting and re-uploading the retrieved records. Deleting a record works analogously, by removing the record before re-encrypting and re-uploading the retrieved records and the record identifiers. Updating a record is considered a concatenation of a delete and an insert operation.

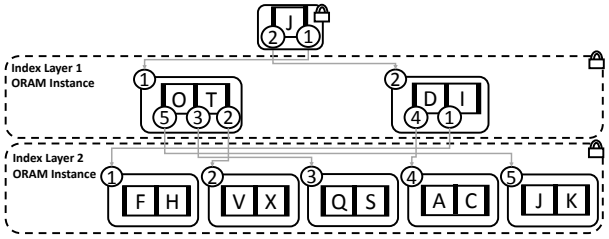
As ORAM schemes allow to store and retrieve data blocks based on a fixed ID but shuffle the stored encrypted data blocks within the DCs with every access, each ORAM instance has to map the fixed block IDs to the current location within the ORAM instance to determine which encrypted data blocks have to be returned. This is illustrated in Figure 2a. For instance, to retrieve node 1 (containing "D" and "I") of index layer 1, the encrypted data block 2 has to be retrieved from the ORAM instance. Thus, data block ID 1 has to be mapped on the encrypted data block ID 2 first. To perform this mapping without harming access pattern confidentiality, more data has to be transmitted and further round trips are necessary between client and SP (see [14]).

The overhead for mapping fixed data block IDs on ORAM instance locations can be avoided in the PATCONFDB case. As shown in Figure 2b, the DC ID can be directly stored in the corresponding B-tree node of the next upper layer. Thus, when executing a query, the IDs of DCs that have to be retrieved from the next layer are already known to the client without any further mapping. For instance, after retrieving and decrypting the root node (containing "J"), the client already knows that the left B-tree node of the next lower layer is contained in the encrypted data block 2.

As ORAM requires to shuffle data blocks and to store them in different DCs after each retrieval, data blocks of the lower layers have to be re-encrypted and re-uploaded



(a) ORAM index layers



(b) Efficiency improved ORAM index layers

Figure 2: Efficiency optimized index layers.

prior to data blocks of the upper layers. When a query is executed, first data blocks have to be retrieved from index layer 1 to get the B-tree node that contains the ID of the DC that has to be retrieved from index layer 2. But the data blocks in index layer 1 cannot be written back right away, as – due to shuffling – the data blocks of index layer 2 are likely to be stored in different DCs than before. Only after new DCs have been assigned to the data blocks of the record store ORAM instance, the data blocks in the upper index layers can be updated and written back recursively.

4.2 Security of PATCONFDB

The PATCONFDB concept consists of multiple ORAM instances. We assume that the utilized ORAM scheme to create each ORAM instance satisfies the ORAM security notion, i.e., any read or write operations on the data within the instance are indistinguishable. We now examine the security implications of executing queries across multiple index layers, i.e., multiple ORAM instances.

Selection queries. As shown in Section 4.1, sequential equality selections are used to evaluate range and prefix selections. Thus, it suffices to show that equality selections are indistinguishable from the perspective of the SP. Every ORAM instance has to be queried exactly once to evaluate an equality selection, so honest-but-curious attackers see a constant number of indistinguishable ORAM accesses. From this, they learn the number of index layers, but they neither learn the parent-child-relationship of nodes in the B-tree nor the total size of data stored in the database. If an equality selection matches multiple records, the query is executed over all index layers for each matching record. Otherwise, it would be possible for an attacker to observe the number of matching records by counting the sequential accesses to the record store ORAM instance. Attackers could still monitor bursts in queries, but this could as likely be caused by an equality selection that has matched many records as by a prefix or range selection. So the type of selection query remains hidden from attackers. The frequency of queries can still be observed by the attacker. Access frequency is an information leak that can be found in all current ORAM schemes. One solution would be to query the database periodically so that bursts in queries could not be detected. Of course, this also generates a large overhead in network traffic and query latency.

Insert / delete / update. Since insert, update, and delete operations are achieved by equality selection queries, attackers are not able to distinguish them from equality selection queries and, thus, prefix/range selection queries.

Since every query or write operation on PATCONFDB leads to the same pattern of successive indistinguishable operations on ORAM instances, the SP is not able to monitor access patterns. Thus, PATCONFDB enforces access pattern confidentiality.

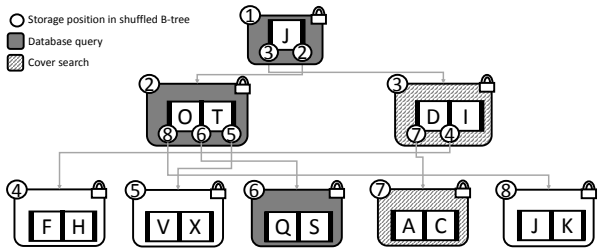
4.3 Shuffled B-tree Index Extensions

To enhance access pattern confidentiality for the retrieval of a single record, three methods are used by existing shuffled B-tree approaches [6, 7, 17]: 1) Cover searches, as seen in Figure 3, are randomly chosen nodes that are retrieved in parallel to the nodes that are actually relevant for the executed query in each layer. For example, if two cover searches are executed, attackers see three retrieved nodes for every layer of the Shuffle Tree. Attackers cannot distinguish between cover searches and nodes that are relevant for the executed query. 2) Node caching at the trusted client is used to increase access pattern obfuscation. After a node is being accessed, it will replace the least recently used node in the client-side cache. All nodes on the path from the root node to the leaf node are stored inside the cache. If a node that is in the cache is retrieved, an additional cover search is executed so an attacker cannot infer that the node was inside the cache. 3) Node shuffling is used to increase the difficulty for attackers to learn the parent-child relationship of the nodes. After each query execution, the content of all nodes stored on the client is shuffled using a random permutation before they are written back to the encrypted B-tree. Node shuffling is outlined in Figure 3.

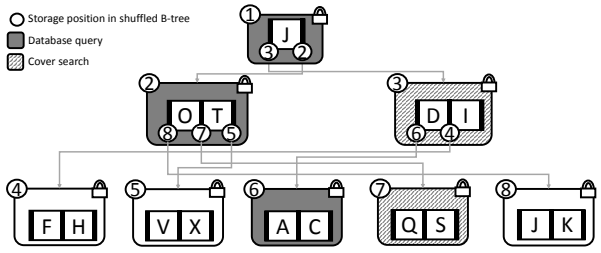
In the remainder of this paper we denote the leaf nodes of the Shuffle Tree as *data nodes* and the non-leaf nodes as *navigation nodes*. In the following, we show how the concept of shuffled B-trees can be extended to not only support equality selections, but also range and prefix selections as well as database modifications, i.e., insert, update, and deletion of database records.

Insert / delete. Current Shuffle Index concepts [6, 17] assume an upfront knowledge of all database records. Knowing the exact distribution of records is not possible in most real world scenarios. Thus, concept extensions that allow to insert, update, and delete records are necessary. We propose a modified shuffled B-tree scheme with a dynamically expanding and shrinking B-tree that is initialized with only a small amount of navigation nodes.

The concept to insert records is outlined in Figure 4. Whenever a record is inserted, an equality selection query for the attribute value of the new record is performed to find the node X to store the record in. If this node X is full, a new node N is created to store the record in. The reference to N is stored in the parent node of node X. If the parent node is full as well, another new node is inserted on the next higher layer of the B-tree. This works recursively to the root of the tree, which does not have any size limitations, because it does not need to be of a certain size to be indistinguishable from any other node. Up to half of the records in the full node X are copied to the new node N while inserting the



(a) Query execution with one cover search



(b) Shuffled B-tree after node shuffling

Figure 3: Shuffled B-tree: Query execution with cover searches.

new record. After the insertion, the nodes are re-encrypted, shuffled and re-uploaded in the same way they would have been in case of a regular equality selection query. The deletion of a record works similar. When the last record in a data node is deleted, the data node itself and its reference in the parent node is deleted.

Update. To ensure the alphabetical sorting of all records, an update that changes the attribute value of the record works as a deletion of the old record followed by an insertion of a new one.

Prefix / range selections. Since insert, update and delete operations are already distinguishable from selections, it is not necessary to make prefix and range selections indistinguishable from equality selections. Therefore, we can use the following method to enhance the performance of prefix and range selections. Instead of retrieving only one data node, every data node inside one navigation node that matches the query can be downloaded. Even though x cover nodes have to be retrieved to obfuscate each data node that was actually retrieved, this method significantly reduces the network traffic and the number of sequential data node retrievals, as discussed in Section 5.2.

4.4 Security of Shuffled B-trees

The security guarantees of the **Shuffle Index** are based on the assumption that attackers are not able to determine the parent-child relationship of nodes in different layers of the Shuffle Tree. To achieve this, the content of all queried

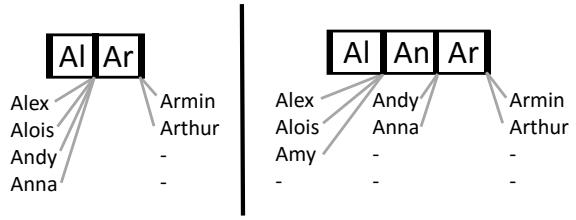


Figure 4: An excerpt of a Shuffle Tree before and after the insertion of the record 'Amy'.

nodes is shuffled, padded, and encrypted with a new nonce in every access. Yang et al. [17] published a proof that, by using the shuffle mechanism, the probability that a node is the child of a parent node X is equal to the probability of it being the child of any other parent node Y after a large enough number of accesses were performed after the last retrieval of X. Since this number of accesses is not further specified, it remains unclear *when* shuffled B-trees enforce access pattern confidentiality sufficiently.

To use the Shuffle Index in relational databases, methods to **insert, update and delete** data had to be implemented. As described in Section 4.3, this leads to the expansion and shrinkage of the Shuffle Tree over time. Therefore, information on the size of the database is leaked and the insert, update, as well as delete operations are distinguishable. Thus, the fourth access pattern confidentiality requirement defined in Section 3.2 is not satisfied. However, this does not lead to the breach of the other three confidentiality requirements. Besides the total size of the database, attackers can also infer how many records can be stored in one node by monitoring a series of inserts and track how often a new node is being created. Users have to decide if these information leaks are acceptable for their specific scenarios.

Prefix / range selections. The performance optimization for prefix/range selection queries as proposed in Section 4.3 introduces two minor information leaks. Attackers can distinguish range and prefix selection queries from other database queries and they can estimate the amount of records that matched a prefix/range query to a certain degree. Attackers can observe the total number of data nodes that were retrieved, but they do not know how many of those nodes are cover nodes. We argue that both information leaks are acceptable since the type of query is distinguishable anyway due to the dynamically expanding shuffled B-tree. The amount of retrieved records would have also been observable to a certain degree without this optimization, because bursts in queries correlate to the amount of retrieved records.

5. EFFICIENCY EVALUATION

5.1 Query Latency Overhead

We evaluated the query latency overhead induced by the use of PATCONFDB and the modified shuffled B-tree by measuring the query latency of equality, prefix, and range selections. For this evaluation, we used the state-of-the-art Path ORAM [16] scheme as ORAM scheme for PATCONFDB. Path ORAM organizes the ORAM DCs in a binary tree. Each leaf of this binary tree is assigned a unique *position* and each data block is assigned to a position. The invariant of Path ORAM states that every data block with an assigned position p is stored on the path from the root DC to the leaf DC with position p. So for every read or write operation on a Path ORAM instance with n DCs, $log(n)$ DCs are retrieved from the SP. After the operation the DCs are re-encrypted with a nonce and re-uploaded to the SP.

We instantiated PATCONFDB with two index layers and one record store. During our measurements, we optimized the capacity of DCs in each layer, resulting in container capacities between 10 and 100 data blocks. Our shuffled B-tree extensions are implemented with a Shuffle Index [6] as described in Section 4.3 with two cover searches, a cache size of five and a Shuffle Tree height of three. In our measurements, we found these parameter choices to have the

	DB size	Records	Bandwidth and latency
Scenario 1	1 GB	1 mio	1 Gbit/s and 5 ms
Scenario 2	1 GB	1 mio	10 Mbit/s and 50 ms
Scenario 3	1 GB	10.000	1 Gbit/s and 5 ms
Scenario 4	1 GB	10.000	10 Mbit/s and 50 ms
Scenario 5	10 GB	10 mio	1 Gbit/s and 5 ms
Scenario 6	10 GB	10 mio	10 Mbit/s and 50 ms
Scenario 7	10 GB	100.000	1 Gbit/s and 5 ms
Scenario 8	10 GB	100.000	10 Mbit/s and 50 ms

Table 1: Summary of the scenario parameters chosen for each scenario.

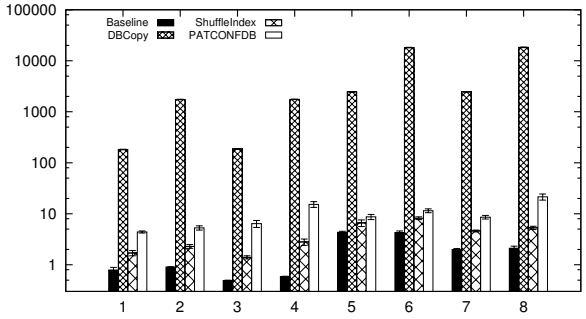


Figure 5: Measured query latency of equality selections over all scenarios on a logarithmic y scale.

best trade-off between security and performance. To better compare the query latency, the naive approach of enforcing access pattern confidentiality by downloading the whole encrypted database on every access is evaluated, hereafter referred to as *DBCopy*. To specify the overhead compared to an approach that does not provide any security mechanisms, an unencrypted database retrieval protocol was also evaluated, hereafter referred to as *Baseline*.

The query latency has been measured on a client computer with an Intel Core i7 CPU with 2,4GHz and 8GB RAM. As a database server a Microsoft SQL Server Express was used on a virtual machine in Hyper-V with 2 virtual CPUs and 16GB RAM. The test data has been created with a random number generator and is evenly distributed. Before every test run, each database is initialized so that the generated database records fill 10% of it. In Table 1 the parametrization of our 2^k-factorial experimental design scenarios is shown.

The query latency of equality selections measured over all scenarios is shown in Figure 5. It can be seen, that an increase of the size of the database results in an increased query latency for all tested protocols (scen. 1-4 vs. 5-8). The bandwidth of the network link is only a critical factor for the DBCopy protocol, because an equality selection only retrieves a small number of records, so the total network traffic is low for all other protocols (scen. 1,3,5,7 vs. 2,4,6,8). The Baseline and the PATCONFDB protocol are significantly influenced by the number of records (scen. 1,2 vs. 3,4). For the Baseline protocol this is the case, because it does not have an index tree to efficiently search for attribute values. For the PATCONFDB protocol it takes a long time to sequentially query index layers which contain a large number of identifiers.

The query latency of range selections measured over all scenarios is shown in Figure 6. Now that the bandwidth of the network link influences the query latency significantly

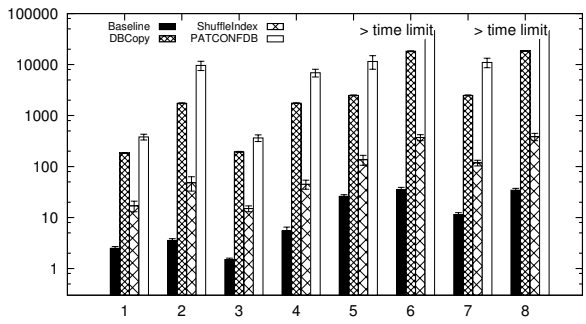


Figure 6: Measured query latency of range selections over all scenarios on a logarithmic y scale.

in case more records are queried (scen. 1,3,5,7 vs. 2,4,6,8). Note that the confidence intervals for the queries executed with PATCONFDB are significantly larger than the ones of the other protocols. This is caused by different sized network overheads as explained in Section 5.2. For scenarios 6 and 8, PATCONFDB even exceeded our time limit of 7h for one query. The Shuffle Index is much less influenced by the bandwidth of the network link, but its query latency is already by a factor of 10 higher than the latency of the Baseline protocol.

Our measurements show that DBCopy can outperform PATCONFDB for range selections if more than about 0,1% off all records are queried in range selections. The results indicate that the PATCONFDB performs better in scenarios with databases that contain small amounts of records with a large size, whereas the Shuffle Index performs better in scenarios with databases that contain a large number of small records. For scenarios in which many records are queried in range selections a network link with a high bandwidth to the SP is needed to keep query latency low.

5.2 Network Overhead

In the following, we investigate the network overhead of PATCONFDB and shuffled B-trees, i.e., the amount of data that has to be transmitted between the client and the SP. First, we provide analytical models to highlight the factors that influence the network overhead both for PATCONFDB and shuffled B-trees. Then we present and interpret the network overheads we measured for the scenarios that we introduced in Section 5.1.

The network overhead N_P^s to retrieve a record from a **PATCONFDB** instance that is based on ORAM scheme s can be calculated as follows:

$$N_P^s = N_r^s + N^t + \sum_{i=1}^{H_I} N_i^s$$

for H_I = number of index layers, N_r^s = network overhead induced by the retrieval of a record from the record store depending on the ORAM scheme s, N^t = size of the root node of the PATCONFDB B-tree, N_i^s = network overhead induced by the retrieval of a record from an index layer depending on the ORAM scheme s.

For an implementation of PATCONFDB with Path O-RAM, the retrieval of a record requires every Path ORAM instance to be accessed two times (Retrieval and Upload of DCs). Since the DCs are stored in a binary tree and all DC on the path from the root to the leaf of the tree are retrieved, the network traffic for every ORAM instance equals the tree height h multiplied with the size of the DC d.

Scenario	SI(EQ)	SI(PR)	PC(EQ)	PC(PR)
1 + 2	562,0	6,56	7080	6557
3 + 4	110,4	6,31	683,1	669,8
5 + 6	1460	6,57	5901	5761
7 + 8	111,3	6,26	1025	971,3

Table 2: Ratio of Shuffle Index (SI) and PATCONFDB (PC) network overhead to the size of the queried records for equality (EQ) and prefix (PR) selections.

The network overhead N_S to retrieve a record from a **Shuffle Index** can be calculated as follows:

$$N_S = 2 \cdot r + n \cdot (h-2) \cdot (1 + a + 2 \cdot c) + d \cdot (1 + a + 2 \cdot c)$$

for h = height of Shuffle Tree, r = root node size, n = navigation node size, d = data node size, c = number of executed cover searches, a = cache size. The network traffic is induced by three factors: I) Retrieval and upload of the root node ($2 \cdot r$), II) retrieval and upload of navigation nodes including cover nodes ($n \cdot (h-2) \cdot (1+a+2 \cdot c)$), III) read and write of the data nodes ($d \cdot (1+a+2 \cdot c)$). Note that actually needed nodes are retrieved and then uploaded as part of the whole cache, inducing a total overhead of $n \cdot (1 + a)$ per B-tree layer.

The **measured network overheads** for the scenarios that we introduced in Section 5.1 are shown in Table 2 for PATCONFDB and the Shuffle Index. The table shows the ratio of the induced network traffic of each approach to the size of the queried records. For instance, if the queried record has a size of 100B and a network traffic of 5,62KB is measured for the evaluation of the query, the relative overhead amounts to 562. The scenarios that only differ in network bandwidth are paired in this table, since the network bandwidth does not affect the amount of network traffic.

For equality selections, the measurements indicate that the total number of records stored in the database has a large impact on the relative network overhead for both schemes (scen. 1,2,5,6 vs. 3,4,7,8). Since the total network traffic for equality selections in both schemes remains low (<1,5MB), the large network overhead does not have a similarly large impact on the query latency measured in Section 5.1. In the case of prefix selections, the performance optimization proposed in Section 4.3 for the Shuffle Index reduces the relative network overhead significantly. Since prefix selections in the PATCONFDB are executed as a series of sequential equality selections, no significant reduction of the relative network overhead is measured. The relative overhead is only reduced, if by chance multiple records that match the query are retrieved within the same DC.

Since the relative network overhead of PATCONFDB remains at a high level, it is not feasible to use PATCONFDB in scenarios, in which a large number of records is queried at the same time or in a short time frame. The Shuffle Index, however, can also be used in scenarios which include large range or prefix selections.

5.3 Storage Overhead

In the following we investigate the storage overhead on the external SP. This storage overhead consists of the storage that is needed for the index as well as of the additionally required storage to store record, i.e., the padding of nodes or the initialization of a database to its maximum size.

The storage overhead of the **PATCONFDB** approach is predominantly influenced by the ratio u of the actual database records' size to the chosen maximum database size. If the maximum database size is 100GB but only 10GB of data is stored inside, the relative storage overhead to the actual data size is 10. The size of the index depends on the maximum number of records that can be stored in a PATCONFDB instance. Each different attribute value of any record is stored inside of the lowest index layer to provide references to every record that is stored in the database. Since index layers are not allowed to grow in size, they have to be initialized to their maximum size as well. So, in the worst case of records that contain only the indexed attribute, the lowest index layer has the same size as the record store. However, the number of DCs that have to be referenced decreases fast for the upper index layers, so that the additional storage overhead for them is very low. In our evaluations the storage overhead for a PATCONFDB instance that is filled to its maximum capacity never exceeded a factor of 2,2 in comparison to the plain text database size with an average factor of 2,08, which we argue is feasible for relational databases. If the PATCONFDB instance is not filled to its maximum capacity, that factor has to be divided by the ratio of used space u to calculate the overall storage overhead.

In contrast to PATCONFDB, the **Shuffle Index** does not need to be initialized to its maximum database size. The storage overhead of the Shuffle Index is induced by the storage needed for navigation nodes and by the ratio of the average number to the maximum number of records stored in data nodes. This used space of data nodes u influences the storage overhead similarly as seen with PATCONFDB. Since the database records are stored, so that they are sorted alphabetically in the leaf nodes of the Shuffle Tree, an index layer that contains every attribute value as seen with the PATCONFDB approach is not necessary. This combined with the dynamic fan out of navigation nodes significantly reduces the storage overhead induced by navigation nodes. In our evaluations the storage overhead for a Shuffle Index instance that only contains data nodes which are filled to their maximum capacity, never exceeded a factor of 1,02 in comparison to the plain text database size. That factor has to be divided by the ratio u of used space of data nodes to calculate the overall storage overhead.

It can be seen, that the storage overhead induced by indexing techniques is a low and constant factor for both approaches and therefore does not restrict the feasibility of access pattern preserving relational databases. However, the constraint of PATCONFDB that the database has to be initialized to its maximum size, could be problematic in scenarios where the size of the database fluctuates frequently.

6. FUNCTIONALITY EXTENSIONS

We address the challenges in complying to access pattern confidentiality in DaaS scenarios with **multiple types of indexes**, i.e. strings, number, and dates, for the use of the PATCONFDB approach. We discuss two situations in which PATCONFDB would leak information about the pattern of queries, if used in the same way as in a single index type scenario. A basic equality selection results in one query to each index layer and one query to the record store.

Insert and delete operations have to access I (number of index types) DCs from the index layer to insert or delete every attribute value of each record to the corresponding data record. Therefore an attacker can differentiate between an equality selection and an insert or a delete operation.

To prevent this information leak, I DCs from the index layer have to be retrieved, before the record store is queried.

Update operations have to access a DC of every index type, which is involved in the SQL operation, from the index layer and then one DC of the record store. Since it is impossible to know the current value before it is update, another DC has to be queried from the index layer, to delete the reference that is no longer needed. Therefore an attacker can differentiate between an equality selection and an update operation. To prevent this information leak, a randomly chosen set of DCs from the index structure would have to be retrieved after the actual query is executed.

Further query types. In this paper, we investigated how equality, range, and prefix selections can be evaluated on access pattern confidentiality-preserving databases. We introduced equality, range and prefix selections based on strings. However, our concepts can be seamlessly applied for range and prefix selections on other data types like integers. We argue that our investigated query types are already sufficient for many scenarios in which relational databases are used. PATCONFDB and modified shuffled B-trees, as introduced in Section 4, can be extended to support table joins and nested queries. Again the queries are divided into sequential equality selections so that they are indistinguishable from any other query. For this to work, all indexes of all tables have to be stored in the same B-tree. If more than one B-tree is used for indexing, every B-tree has to be accessed every time a query is executed for the queries to be indistinguishable. In future work we plan to implement and evaluate these query types to give recommendations on which setup performs best for which database scenario.

7. CONCLUSIONS & FUTURE WORK

To investigate the feasibility of access and pattern confidentiality-preserving relational databases with a B-tree index, we proposed PATCONFDB, an ORAM-based concept to achieve access pattern confidentiality, and extended existing shuffled B-tree approaches to support essential database operations. In particular, empirical measurements of these concepts showed that enforcing access pattern confidentiality in relational databases only induces an overhead of factor 5.9 for evaluating equality conditions on a database with up to 10 million records. The extended Shuffle Index induces an even smaller overhead of factor 2.3 for the same setup, but provides no strict and well-defined access pattern confidentiality guarantees. Our performance evaluation showed to which extend the induced overhead depends on the network link to the SP, the structure of the data and the query workload of the outsourcing scenario. In particular, our results showed that shuffled B-trees in general outperform ORAM-based B-trees and, thus, can be used in a wider variety of DaaS scenarios, if no strict security guarantees are required.

The findings of this paper highlight multiple future research directions. It is worthwhile to aim for further efficiency improvements of ORAM-based database indexes, as well as to evaluate the performance of other index structures like bitmaps. The superior efficiency of shuffled B-tree approaches makes it also worthwhile to aim for a better understanding of their security guarantees. Furthermore, we plan to investigate how both ORAM-based and shuffle-based schemes can be extended to support relational databases with multiple index types and queries that include more complex operations.

8. REFERENCES

[1] M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and efficiently searchable encryption. In *Proc. of the Annual Intl. Cryptology Conf. on Advances in Cryptology (CRYPTO)*, pages 535–552, 2007.

[2] P. Brody and V. Pureswaran. Device democracy: Saving the future of the internet of things. *IBM Global Business Services Executive Report*, 2014.

[3] A. Ceselli, E. Damiani, S. D. C. D. Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Modeling and assessing inference exposure in encrypted databases. *ACM Transactions on Information and System Security*, 8(1):119–152, 2005.

[4] I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious ram without random oracles. In *Proc. of the Conf. on Theory of Cryptography (TCC)*, pages 144–163, 2011.

[5] J. Dautrich, E. Stefanov, and E. Shi. Burst ORAM: Minimizing ORAM response times for bursty access patterns. In *Proc. of the USENIX Security Symposium*, pages 749–764, 2014.

[6] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati. Efficient and private access to outsourced data. In *Proc. of the IEEE Intl. Conf. on Distributed Computing Systems (ICDCS)*, pages 710–719, 2011.

[7] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati. Distributed shuffling for preserving access confidentiality. In *Proc. of the European Symp. on Research in Computer Security (ESORICS)*, pages 628–645, 2013.

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

[9] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *Proc. of the Intl. Conf. on Automata, Languages and Programming (ICALP)*, pages 576–587, 2011.

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

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

[12] J. Köhler, K. Jünemann, and H. Hartenstein. Confidential database-as-a-service approaches: taxonomy and survey. *Journal of Cloud Computing*, 4(1), 2015.

[13] R. Ostrovsky. Efficient computation on oblivious RAMs. In *Proc. of the ACM Symposium on Theory of Computing (STOC)*, pages 514–523, 1990.

[14] L. Ren, X. Yu, C. W. Fletcher, M. van Dijk, and S. Devadas. Design space exploration and optimization of path oblivious RAM in secure processors. In *Proc. of the Intl. Symp. on Computer Architecture (ISCA)*, pages 571–582, 2013.

[15] E. Stefanov and E. Shi. ObliviStore: High performance oblivious distributed cloud data store. In *Proc. of the Network and Distributed Systems Security (NDSS) Symposium*, 2013.

[16] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *Proc. of the ACM Conf. on Computer and Communications Security (CCS)*, pages 299–310, 2013.

[17] K. Yang, J. Zhang, W. Zhang, and D. Qiao. A light-weight solution to preservation of access pattern privacy in un-trusted clouds. In *Proce. of the European Conf. on Research in Computer Security (ESORICS)*, pages 528–547, 2011.