TRANSACTIONS ON DATA PRIVACY 11 (2018) 81–109

PATCONFDB: Design and Evaluation of Access Pattern Confidentiality-Preserving Indexes

Alexander Degitz, Hannes Hartenstein

Institute of Telematics, Karlsruhe Institute of Technology (KIT), Germany

E-mail: alexander.degitz@partner.kit.edu, hannes.hartenstein@kit.edu

Received 27 October 2017; received in revised form 28 May 2018; accepted 18 June 2018

Abstract. When sensitive data is outsourced to an untrustworthy cloud storage provider, encryption techniques can be used to enforce data confidentiality. Ideally, such encryption techniques should not only enforce the confidentiality of data at rest but also the confidentiality of data accesses, as database access patterns can leak information about the database's contents. Oblivious RAM (ORAM) approaches were proposed to hide access patterns, but they currently support a very limited set of database query operations. In this paper[1], we propose PATCONFDB that supports database query functionalities and hides query access patterns. PATCONFDB represents a construction based on ORAM schemes: when an index (B-tree) is outsourced, multiple ORAM instances are used to maintain access pattern confidentiality. PATCONFDB can make use of up-to-date ORAM schemes, for example an implementation of Burst ORAM is used to significantly boost the performance of accesses. We compare PATCONFDB with a shuffled B-tree protocol, provide a discussion on security properties, and give recommendations of which protocol to use in which usage scenario. We provide a rigorous efficiency evaluation to determine the storage and network overhead as well as query latency. In particular, we show that PATCONFDB with ORAM-based schemes like Burst ORAM only causes a marginal latency overhead when evaluating equality conditions on databases of up to 10 million records. However, the network overhead still remains a challenge.

Keywords. Confidential Database-as-a-Service, Access Pattern Confidentiality

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 might be "curious" or do not apply sufficient protective measures to protect the data against third party attackers. With the advent of the Internet of Things, it is expected that users might not even be aware of *who* will store their data anymore [3]. Enforcing the confidentiality of data *before* it is outsourced to an SP[2] mitigates the risk

[1]Extended version of paper "Access Pattern Confidentiality-Preserving Relational Databases: Deployment Concept and Efficiency Evaluation" [8] at EDBT/ICDT 2016

[2]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.

of a data confidentiality breach. Thus, techniques that enforce data confidentiality before the data is outsourced will gain even more importance. However, outsourced data needs also protection from attackers who are able to monitor and analyze access operations on outsourced data.

Outsourcing confidential 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 queried efficiently. To achieve that, a variety of confidentiality preserving indexing approaches (CPIs) were proposed that allow to efficiently evaluate database queries on outsourced encrypted data [14]. 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 plain text 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".

The property of access pattern confidentiality (APC) can, of course, be trivially achieved by retrieving the entire database for each query to make the queries indistinguishable. This naive approach, however, induces significant efficiency costs in terms of latency, transmission, and computation. Oblivious RAM (ORAM) schemes [5, 9, 11, 15, 16, 20] allow to store and retrieve records based on fixed identifiers and enforce APC by selectively retrieving, re-encrypting, and re-submitting specific parts of the outsourced data in an interactive way. However, ORAM schemes do natively not support the query capabilities that are required by relational databases and, so far, it has been unclear whether their efficiency would be acceptable for relational database scenarios[3].

In this paper (which is based on our workshop paper [8]) we present the design of a CPI with APC property, named **PAT**tern **CONF**identiality in **D**ata**B**ases, based on ORAM primitives. 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 evaluate security and efficiency properties of PATCONFDB and compare them to shuffled B-tree approaches [1, 6, 7, 21, 19], a 'competing technology'. The main contributions of this paper are:

- PATCONFDB, a **concept to efficiently apply ORAM schemes in DaaS settings** that supports querying a database for records which match specified equality, range, and prefix query conditions and guarantees access pattern confidentiality.
- A **new variant of shuffled B-tree approaches to increase the efficiency** in DaaS settings. The shuffled B-tree variant is used to analyze the performance gain when prefix and range queries are allowed to be distinguishable from equality selections.
- An **efficiency evaluation** that shows that significant performance increases are possible by considerately applying ORAM schemes. In particular, we show that when Burst ORAM [5] is used in PATCONFDB, there is only a small overhead for equality selections compared to an unencrypted database access. However, for range or prefix selections a large overhead in query times remains when using ORAM schemes. Furthermore, we compare the PATCONFDB approach to the shuffled B-tree variant and provide **guidelines for choosing a suitable indexing approach for a given outsourcing scenario**.

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

In Section 2 related work on Oblivious RAM and shuffled B-tree approach 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: on the one hand we show how a B-tree index can be outsourced using multiple ORAM instances (Section 4.2), on the other hand we give a bitmap index-based example that does not outsource the index and only needs a single ORAM instance (Section 4.3). In Section 5 we introduce the shuffled B-tree variant that allows efficient range/prefix queries that are distinguishable from equality selections. We provide an efficiency evaluation of the proposed schemes (outsourced B-tree with multiple ORAM instances, bitmap index not oursourced with single ORAM instance, our variant of shuffled B-trees) in Section 6. The evaluation, thus, compares the ORAM-based approaches with strict APC property with the shuffled B-tree approach that provides less strict APC guarantees. We discuss possible functionality extensions in Section 7 and conclude the paper in Section 8.

2 Related Work

Many protocols have been proposed to efficiently perform keyword searches over encrypted data, i.e. [2, 13, 14]. While these approaches ensure content confidentiality, they leak the access patterns of queries. Based on an exemplary database, Islam et al. [12] 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 [10]. 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, 11, 15, 16, 20], i.e., storing and retrieving data records based on fixed identifiers. Outsourced relational databases, on the other hand, are more complex and require efficient index structures 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 queries and, to the best of our knowledge, it is unclear whether applying ORAM to encrypt relational databases is feasible with regard to functionality and 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 queries on relational databases.

Encrypted B-trees [4] 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 [1, 6, 7, 19]. Note that, despite making access pattern based inference attacks harder, the Shuffle Index does not provide strict APC guarantees like ORAM does. In the extended version [21] of the Shuffle Index paper [6] the authors added the functionality to perform update operations as well as range and prefix selections, which are executed as a series of equality selections. In this paper we propose a different approach to execute prefix and range selections for shuffled B-tree schemes, which allows these queries to retrieve multiple matching data sets in each query and are, thus, distinguishable from equality selections. The proposed variant can be applied to all schemes that make use of a shuffled B-tree.

3 Deployment Requirements

3.1 Database Functionality Requirements

Databases need to support **inserts**, **updates**, and **deletions** of data records. Furthermore it is required to query databases for specific records. The structured query language (SQL) specifies a variety of different query types. The CPIs at least need to be able to execute the following basic types of queries to be usable in a DaaS scenario:

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 pattern confidentiality for additional query types in Section 7.

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 outsourced data.

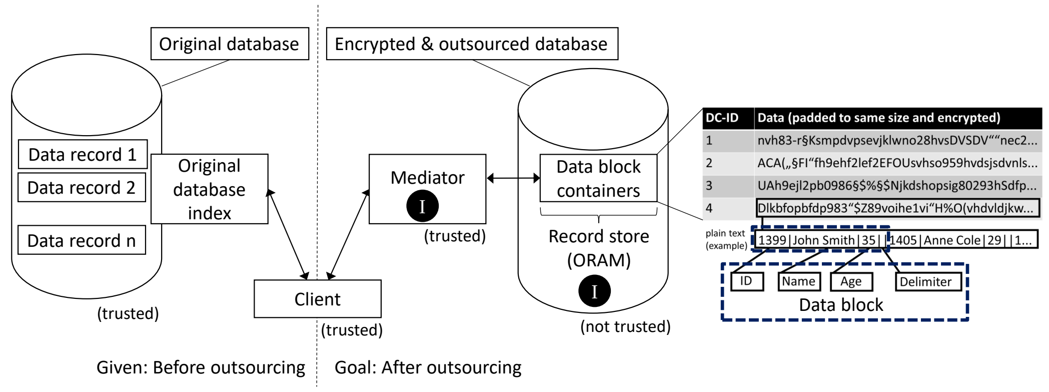
Content confidentiality (CC): A database outsourcing approach provides content confidentiality if an attacker that is able to view the outsourced data is not able to learn the content of the database's records.

Access pattern confidentiality (APC): We adopt the definition of access pattern confidentiality from [18]. 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. The probability for an attacker to guess the plain text of any encrypted data record remains the same before and after any observed access.

To fulfill the strict requirements of APC, an attacker, who has live access to the outsourced database and the network traffic from and to the database server, must not be able to tell if a certain data block has been read before. For this to be ensured, the data blocks have to be encrypted with a probabilistic encryption scheme and a new random seed after every query. If this is the case, Content Confidentiality is ensured as well. For the remainder of this paper, we will only mention APC as a requirement, since CC is included in APC.

4 Design of APC Indexes

To satisfy the deployment requirements, we propose PATCONFDB, a CPI that combines existing ORAM schemes with a B-tree index structure in Section 4.2 and an underlying



DC-ID	Data (padded to same size and encrypted)
1	nvh83-r§Ksmpdvpsevjklwno28hvsDVSDV““nec2...
2	ACA(„§FI“fh9ehf2lef2EFOUsvhso959hvdsjsdvnls...
3	UAh9ejl2pb0986§$%§$Njkdshopsig80293hSdfp...
4	Dlkbfopbfdp983“$Z89voihe1vi“H%O(vhdvldjkw...

Figure 1: Architecture of a PATCONFDB deployment concept. The circle I represents a possible location for storing the index structure, depending on the index used.

bitmap in Section 4.3. Before the details of PATCONFDB are explained, the general architecture and database structure are introduced in the following subsection.

4.1 General Architecture

The architecture of a secure data outsourcing scenario in which the CPI hides access patterns is shown in Figure 1. Clients query data records by sending their request to a mediator or broker on which the CPI is running. The mediator is located inside of a trusted network, so traffic from clients to the mediator can be in plain text. The mediator is the only entity querying the database on the external storage provider.

Note that ORAM itself requires one or more index structures (typically called position maps). In combination with the index structure for the required attribute(s), a CPI with APC property is formed. Such a CPIs can store all of its index structures locally on the mediator, or can outsource parts of the index structure to the SP. For PATCONFDB, we will evaluate both options: parts of the index structures are outsourced in the case of an underlying B-tree (Section 4.2) and are stored on the mediator in the case of a bitmap (Section 4.3). The outsourced data, thus, stores encrypted data block containers (DCs) that contain outsourced data blocks, which can either contain data records or index information. A functional comparison of outsourced and non-outsourced index structures is given in Section 4.4.

On the right side of Figure 1 it can be seen that an outsourced data block container (DC) can contain many data blocks. DCs are always padded to the same length and probabilistically encrypted to make them indistinguishable for the SP. The size of DCs, which determines how many data blocks can be stored inside, is one of the most important parameters on the efficiency of an ORAM instance. We will further elaborate the influence of the DC size in Section 6.

When a client queries a data record, the mediator uses an index structure to identify and retrieve the data block container, which contains the queried data block. After decrypting the data block, the mediator can reconstruct the data record and return it to the client.

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

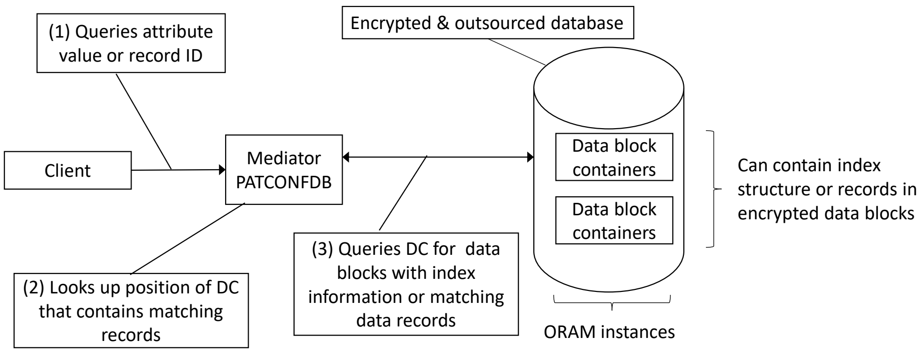

(1) Queries attribute value or record ID
Encrypted & outsourced database
Client
Mediator PATCONFDB
Data block containers
Data block containers
Can contain index structure or records in encrypted data blocks
(2) Looks up position of DC that contains matching records
(3) Queries DC for data blocks with index information or matching data records
ORAM instances

Figure 2: Process of a client query which retrieves matching data records.

the SP. 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. PATCONFDB makes use of this interface as illustrated in Figure 2. First a client queries for an attribute value like 'Name = Scott' or the ID of a record to retrieve. In a second step, the mediator looks up the ID and position of the DC that contains the matching record. In a third step the actual *ORAM.get(ID)* method is called to retrieve the data block that contains the matching data record, which is returned to the client.

New ORAM schemes are proposed every year. For now, they can be categorized into two main categories, depending on how they retrieve stored data block containers (DCs):

1) Approaches like Oblivistore and Path ORAM first retrieve and decrypt a set of encrypted data block containers, of which one contains the queried data block. The decrypted data blocks then have to be re-encrypted and the resulting encrypted DCs have to be re-uploaded to the SP. This keeps computation and data block management simple, but causes a large overhead in network bandwidth.

2) Approaches like Burst ORAM and Onion ORAM only retrieve one DC, which is computed by the SP through an XOR-operation or a homomorphic addition of several DCs. The mediator or client then has to recompute the queried data block by inverting the computation of the SP. With these approaches, the SP still does not know which data block was retrieved. Afterwards there are also DCs written back to the server to make read and write operations indistinguishable. Approaches of this category have a very low overhead in network bandwidth, but a potentially high overhead in computation.

In most database scenarios, data records are queried based on attributes. This creates the need for indexes to efficiently perform relational database operations like prefix or range selections. We have implemented PATCONFDB with both a B-tree index structure and a bitmap index structure.

4.2 PATCONFDB with a B-Tree Index Structure

Since an encrypted B-tree index structure for databases can have a similar size to the size of the actual data, the index structure needs to be outsourced as well. When outsourcing the index structure, we have to be very careful to still maintain Access Pattern Confidentiality (APC). This section will explain the use of a B-tree index structure for PATCONFBDB and will explain how APC is preserved when outsourcing the index structure. To query

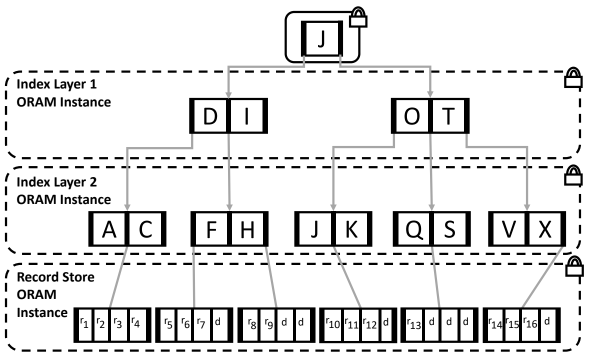


Figure 3: Hierarchy of the ORAM instances in PATCONFDB (Example with two index layers).

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 3. For the remainder of this paper we call the ORAM instances that contain the index structure *index layers* and the ORAM instance that contains the actual data records *record store*. Note that the used ORAM instances (one for each layer) keep their local index structure or position map as it is. The B-tree index structure, however, spans over all used ORAM instances.

4.2.1 Performance Optimization of B-tree Index

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 4. It can be seen that the potentially large Position Maps are stored on the mediator. The locks symbolize an entity (ORAM instance and also the single root DC) of which the contents can be retrieved without access patterns being leaked. The retrieval of the root DC does not leak any information since every query starts with the retrieval of the root DC, except for re-encrypting the root DC with a random seed, no further security mechanisms as seen in ORAM are necessary.

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 [17]).

The overhead for mapping fixed data block IDs on ORAM instance locations can be avoided

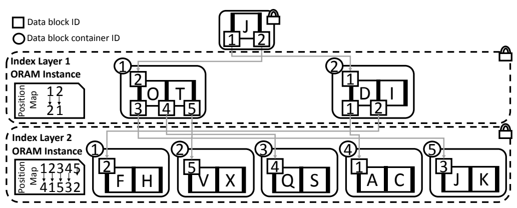


Figure 4: ORAM index layers with Position Maps outsourced on the SP.

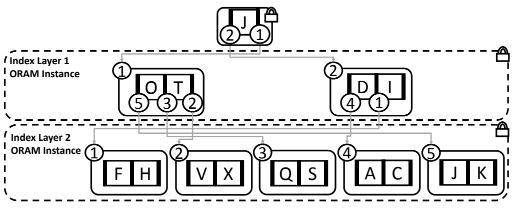


Figure 5: Improved ORAM index layers with integrated Position Maps outsourced on the SP.

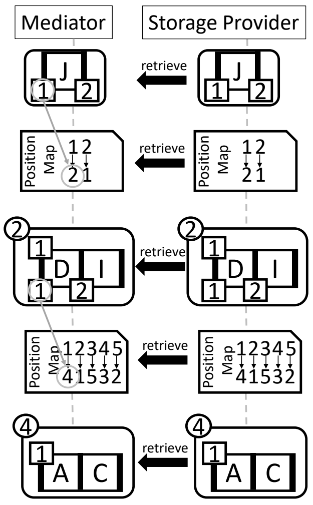


Figure 6: Retrieval sequence of trivial outsourcing of the Position Map.

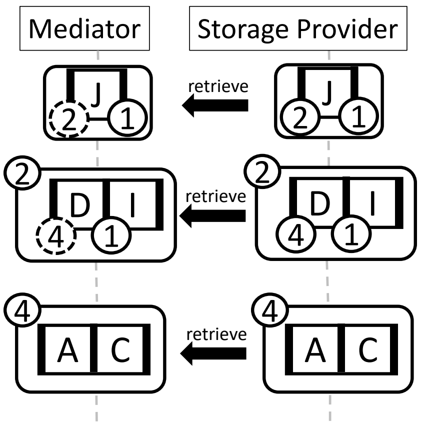


Figure 7: Retrieval sequence of improved outsourcing of the Position Map. The dotted circles mark the ID of the next DC to be retrieved.

in the PATCONFDB case. As shown in Figure 5, 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. The process of a query, which is performed in this optimized way, compared to the naive way, can be seen in Figure 6 and Figure 7. Since, in the optimized approach, the position map of a lower layer ORAM instance has already been stored in the ORAM layer above, there is no need to retrieve the position map of each layer separately, reducing the number of retrievals by $(n-1)/2$ for $n =$ number of ORAM instances in PATCONFDB. When the root DC is retrieved, the mediator can encrypt it and directly learn the position and the ID of the DC, that needs to be retrieved in the next lower layer ORAM instance.

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 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.

In the following, we describe how PATCONFDB in a setting with the improved B-tree index structure 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 4.2.2.

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.

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.

4.2.2 Security Analysis of Outsourced B-tree Index

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 4.2, 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 (APC). We therefore have shown that a composition of ORAM instances, which within themselves enforce APC, also enforces APC.

4.3 PATCONFDB with Bitmap Index Structure

We also implemented a bitmap index structure into PATCONFDB to compare its performance to a B-tree index structure. Since bitmap index structures are most efficient in scenarios with low-cardinality attributes, it is not useful to outsource the bitmap index structure because it comes with low enough storage requirements for the mediator to just always be stored locally. Therefore there is also no need for encryption of the bitmap index structure, because the mediator, which it is stored on, is considered a trusted instance. On the SP only one ORAM instance is stored, the record store. The Position Map is again integrated into the index structure, as it was the case in Section 4.2.1 for the improved B-tree index structure.

In Figure 8 an exemplary bitmap index structure for 8 data records in a database table of the health insurance status of people is shown for the attribute values *uninsured, insured, private, federal*. Additionally to the Record-ID, also the DC-ID of the DC that contains the data record has to be stored, because the DC will be retrieved by a query. For every data record there is a row in a bitmap index structure. For every attribute that this data record fulfills, there is a '1' in the bitmap index structure, and a '0' otherwise. For any selection a bit-vector is created based on the queried attributes. In our example the bit-vector for a query returning all people with a federal health insurance is [0, 1, 0, 1]. A query can also skip attribute values. The bit-vector [0, 1, -, -] returns all people who have any kind of insurance.

4.3.1 Performance Optimization of Bitmap Index

The previous solution has one big drawback: It is neither necessary nor feasible to store the attributes of every single data record, because it is not single data records that are retrieved, but DCs, which usually contain many data records. Our test results have shown in 6, the best performance can be achieved when a DC contains between 10 and 100 data records. The improved bitmap index structure is set up in a way that the number of rows equals the number of DCs in the record store and the columns are *DC-ID, attribute value 1...attribute*

Record-ID	DC-ID	Uninsured	Insured	Private	Federal
1	1	1	0	0	0
2	1	0	1	0	1
3	3	1	0	0	0
4	1	0	1	0	1
5	4	1	0	0	0
6	4	0	1	1	0
7	2	0	1	1	0
8	3	1	0	0	0

Figure 8: Naive bitmap index structure for attributes of people based on single data records.

DC-ID	Uninsured	Insured	Private	Federal
1	1	1	0	1
2	0	1	1	0
3	1	0	0	0
4	1	1	1	0

Figure 9: Improved bitmap index structure for attributes of people based on all data records within a data block container.

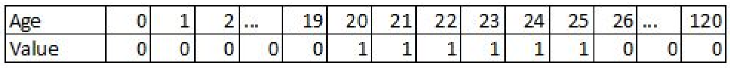
value n. The bitmap therefore indicates for every DC, if it contains a data record with a certain attribute value. The current number of data blocks stored in each DC is stored in a file on the mediator, so that the mediator can always find non-full DCs to store new data blocks in. In Figure 9 the improved bitmap index structure for the same example is shown. Note that the right bitmap index structure is directly derived from the one on the left by using an OR-operator for all data records that have the same DC-ID. In real world scenarios with DCs that contain 10 to 100 data records this reduction of bitmap index structure size also has a factor of 10 to 100. Since the position of the DCs inside the ORAM record store needs to be updated on every access, occasional updates in the bitmap index structure when data records are updated or shuffled do not induce a significant overhead query latency.

In the following, we describe how PATCONFDB in a setting with the improved bitmap index structure supports the database functionality that is described in Section 3. **Equality selections.** To retrieve all records that contain a specific attribute value, a bit-vector is created based on the queried attributes. Based on the matching DC-IDs, it can be determined which DCs have to be retrieved to retrieve all data records that match the queried attribute value.

If more than one DC-ID matches the query, the DCs are retrieved in consecutive 'get'-operations to the record store ORAM instance. The equality selection has to be evaluated again for each matching DC-ID before retrieving it from the record store, even if all DC-IDs are already known. Similarly to PATCONFDB with a B-tree index structure, this mechanism is needed to keep queries indistinguishable.

Prefix / range selections. For bitmap index structures, prefix and range selections are executed in the exact same way as equality selections. Yet both prefix and range selections are very rare for bitmap index structures, because they work directly against the principle of few distinct attribute values, which a bitmap index structure is optimized for. For an attribute value like 'Age', columns in the bitmap structure from '0' to '120' would be necessary. To perform a range selection for people aged '20' to '25', a bit-vector as shown in Figure 10 would be created and then executed like an equality selection.

Insert / delete / update. To insert a new data record, an equality selection for a random DC-ID is performed, which is not yet completely filled with data blocks. The data record is stored in the DC and the bitmap index structure row for this DC is updated accordingly. Deleting or updating a record works analogously, with the small change that at this time the DC which contains the data record that is updated or deleted is queried and the



Age	0	1	2	...	19	20	21	22	23	24	25	26	...	120
Value	0	0	0	0	0	1	1	1	1	1	1	0	0	0

Figure 10: Example: Bit-vector for a range selection of the attribute values ’20’ to ’25’ of the attribute ’age’.

bitmap index structure is updated accordingly. If many DC-IDs match the characteristics of the data record, that needs to be updated, the DCs are again queried consecutively as described above for equality elections until the data record that is updated or deleted has been retrieved.

4.3.2 Security Analysis of Bitmap Index

An additional security analysis is not necessary since the bitmap index structure remains on the trusted mediator, so the security guarantees of PATCONFDB with a bitmap index structure are the same as the ones of the underlying ORAM scheme.

4.4 Functional Comparison of PATCONFB and a Single ORAM Scheme

The PATCONFDB concept utilizes several single ORAM instances so that they can be used in a DaaS scenario. For databases with several attributes that require indexes, the necessary client side storage to store index tables for all of these indexes can become very large. In some scenarios the necessary client storage can even exceed the size of the database. PATCONFDB outsources index tables to reduce necessary client storage to a small constant value. PATCONFDB also provies a framework to deploy hierarchical ORAM instances and efficiently look up index tables, especially for prefix and range selections. It is agnostic to the underlying ORAM scheme and can even be used with different ORAM schemes in the same deployment. PATCONFDB is well suited for scenarios in which multiple indexes are present that are optimized for different ORAM schemes. If the amount of necessary client side storage due to index tables does not pose a problem because of scarce indexes in a certain scenario, the PATCONFDB concept should not be used, as it increases the network traffic overhead and query latency in comparison to a single ORAM scheme.

5 Shuffled B-Tree Approaches

In this section we look at shuffled B-tree approaches (see Section 2) as an alternative and for comparison to PATCONFDB. The extended Shuffle Index implementation [21] fulfills the functionality requirements of Section 3 and, thus, represents a good candidate for comparison. This implementation hides the type of any database operation by serializing prefix or range queries to equality selections. We are, however, also interested in the performance gain when prefix or range queries are not serialized and, thus, are distinguishable from equality selections. Therefore, in this section we propose a variant of the Shuffle Index approach with increased performance for range and prefix selections. We argue that it might be reasonable to give up the indistinguishability of prefix/range and equality selections as follows.

Based on our functionality requirements, we have to consider five types of database operations: (1) Add a data record (2) Update a data record (3) Delete a data record (4) Query a single data record (5) Query multiple data records. Operations (2) and (4) are indistinguishable for most shuffled B-tree approaches already. The authors of [21] use random splitting of nodes during some queries to also make operations (1) and (3) indistinguishable from each other and from (2) and (4). To ensure that the operation (5) is also indistinguishable from all others, the authors process range or prefix selections as a series of equality selections. While this strategy makes it harder for attackers to determine if one or multiple data records were queried, is does not make it impossible.

Example: Consider a database that is queried infrequently, e.g. once every hour. If a single data record is queried, an attacker observes one 'Shuffle Index operation' performed on the database. This will be the only query the attacker observes within one hour. If a range query is executed, an attacker observes a multitude of 'Shuffle Index operations' performed consecutively, and then no other query for the rest of the hour. While these multiple operations all look like the retrieval of single data records, the probability of those queries being unrelated in an otherwise sparsely queried database is close to 0. The attacker was able to determine which operations retrieved a single data record and which operation was a range selection.

As this example shows, an attacker has the ability to distinguish operations (4) and (5). If databases are queried more frequently, all an attacker needs to do is to collect more data about usage patterns to be able to distinguish operations (4) and (5). One approach to make them indistinguishable is an intervall-based database that solely retrieves a single data record at the start of its intervall time. If there is no data record to query, a random data record will be retrieved. However, intervall-based databases greatly increase the network traffic overhead and are only useful in very specific scenarios.

Based on this conclusion, we can increase the performance of prefix and range queries by accepting that they will not be indistinguishable. We outline our proposed variant for a shuffled B-tree approach in Section 5.1. We like to emphasize that our variant is not better or worse than [21], but conceptually represents a different point on the 'performance versus security trade-off curve'. We actually also give up the indistinguishability of insert and delete operations.

Please note that shuffled B-tree approaches do not (yet) provide similarly strict APC guarantees than ORAM does. Please also note that shuffled B-tree approaches are fundamentally different to PATCONFDB. Even though both concepts work with a B-tree index structure, PATCONFDB uses provably secure ORAM instances to store the layers of a B-tree on th SP, whereas shuffled B-tree approaches store the nodes of the B-tree directly on the SP. To hide the relationship between nodes, shuffled B-tree approaches rely on shuffle operations and the retrieval of dummy nodes, which makes it more difficult, but not impossible, for an attacker to link any information. We discuss the security of our proposed solution in Section 5.2.

5.1 Shuffled B-tree Variant

Shuffled B-Tree approaches use a B-tree with a dynamical fan out. There are two categories of nodes. Navigation nodes, which contain a list of all their child nodes and current positions, and data nodes, which are the leaf nodes of the B-tree that contain the actual records. All data in the nodes is alphabetically sorted. All navigation nodes are padded to the navigation node size. All data nodes are padded to the data node size. The root node does not need to be padded, because any attacker can openly identify the root node as the first node retrieved with any operation on the database. To enhance access pattern confidentiality for the retrieval of a single record, three methods are used by existing shuffled

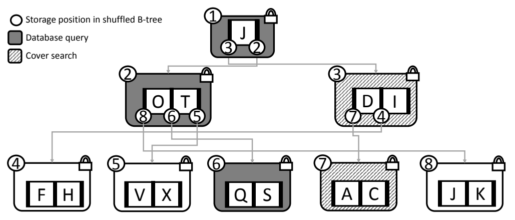


Figure 11: Query execution with one cover search

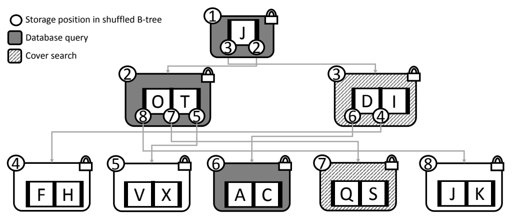


Figure 12: Shuffled B-tree after node shuffling

B-tree approaches [6, 7, 19]: 1) Cover searches, as seen in Figure 11, 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 12.

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 present our variant of shuffled B-trees.

Insert / delete. 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 13. 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

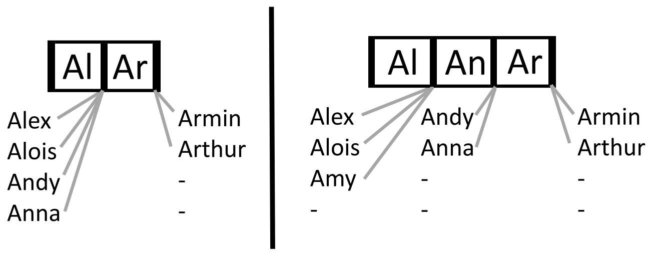


Figure 13: An excerpt of a Shuffle Tree before and after the insertion of the record 'Amy'.

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 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. When records in neighboring nodes get sparse through deletion, there is no direct way to merge nodes within an equality selection, since only one child node for each parent is retrieved, but we would need at least two child nodes to do a merge. To prevent having many nodes with only one or a few records inside, we have implemented a node merging algorithm that is triggered when neighboring nodes are retrieved within a prefix or range query. If two neighboring nodes are retrieved and both contain less records than the square root of the nodes' maximum capacity, the two nodes get merged into one and the references to the parent node are updated accordingly.

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. Instead of retrieving only one data node, every data node inside one navigation node that matches the query can be downloaded. Even though multiple 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 6.1.2.

5.2 Security of Shuffled B-Tree Approaches

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 nodes is shuffled, padded, and encrypted with a new nonce in every access. Yang et al. [19] 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 once 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. It therefore remains unclear if an attacker has a higher probability of guessing the plain text of any encrypted data record after any observed access or series of accesses.

Therefore, shuffled B-tree approaches do not provide strict security guarantees like ORAM schemes do. It is known that there is a considerable degeneration of information in terms of query link-ability due to mechanisms like node shuffling. However, it is not proven yet if the degeneration of information happens fast enough to prevent inference attacks from being possible before n is large enough to make the location of records indistinguishable. To further investigate this question, we ran an attack simulation on our shuffled B-tree implementation. Although we can not claim that we have implemented the best possible attacker, our impression is that the information degeneration is much faster than the possible information gain. This would make shuffled B-tree approaches secure enough for the use in real-world scenarios.

The type of attacker shuffled B-tree approaches protect against is somewhat unclear. The cache as a security mechanism as proposed in [6] only protects from an attacker that manages to actively query the database either as an authenticated user or as a delegator of a query which an authenticated user then executes. When this is the case, the cache increases the number of shuffles between two queries an attacker can link together, therefore better hiding the parent-child relationship of the tree nodes. If however an attacker can actively query the database, he is not only far more powerful than the assumed "honest-but-curious" attacker, he also has more and easier ways to retrieve plain text data through queries instead of inference attacks.

To use the Shuffle Index in relational databases, methods to **insert, update and delete** data had to be implemented. While the authors of [21] implemented an algorithm that splits nodes randomly on some operations, we propose a solution that does not try to hide the type of operation and therefore does not cause any additional overhead in query time or network traffic. The insertion or deletion of a data record could cause the Shuffle Tree to expand or shrink 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 5.1 introduces two 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.

6 Efficiency Evaluation

To evaluate the feasibility of access pattern confidentiality preserving databases in practice, we evaluate the performance of queries and the efficiency with regard to server storage and network traffic of PATCONFDB and the shuffled B-tree variant. As the results were generated for specific implementations, they cannot be considered as a final answer to the question whether preserving access pattern confidentiality is feasible. However, they do provide insights in the form of lower bounds, i.e., while more efficient schemes will be proposed in the future most likely, we are able to show that enforcing access pattern confidentiality is feasible in specific application scenarios. 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 of PATCONFDB, we used the state-of-the-art ORAM schemes **Path ORAM** [18] and **Burst ORAM** [5]. For the remainder of this section, we refer to our PATCONFDB implementation that uses Path ORAM as PathCONFDB and to our PATCONFDB implementation that uses Burst ORAM BurstCONFDB.

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 mapping of data blocks and their positions is stored in the *position map*. 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 and temporarily stored on the client *stash*. The necessary client storage is the size of the *position map* added to the size of the *stash*. After the operation a new random path number is assigned to the queried data record. This new path number is updated in the *position map*. All other retrieved data records are pushed further towards the leaf nodes of the binary tree if possible. This is the case whenever a node could be stored in a deeper node without breaking the invariant of Path ORAM. After the data records are put inside their new DCs and the index structure is updated, the DCs are padded to the same size, re-encrypted with a nonce and then re-uploaded to the SP. If a DC could not be placed on the path to its position because all of the DC (including the root DC) are full, the node remains in the client *stash* and another attempt to place it in the tree will be done during the next query.

Burst ORAM organizes the ORAM DCs in $log(n)$ levels and $\sqrt{n}$ partitions. It also uses an index table on the client to keep track of which data record is stored in which level and partition. When a data record is retrieved, the DC containing the data record plus one additional dummy DC of every other level are chosen. Then a block-level XOR operation is executed on the server over all chosen DCs to create one new DC, which is transferred to the mediator. The mediator can reconstruct the chosen dummy DCs to invert the XOR operation and retrieve the data record inside the retrieved DC. To make queries indistinguishable from updates and to clean up the tree, Burst ORAM also has a mandatory write-back operation. Opposed to other ORAM schemes, Burst ORAM does not need to execute the write-back operation right away. Insted it can keep querying data records until one level runs out of dummy DCs. Only then new Dummy DCs have to be created and all retrieved data record are put inside new DCs to prepare the write-back operation. Then the index structure is updated, the DCs (including dummy DCs) are padded to the same size and re-uploaded to the SP. This way, Burst ORAM can quickly perform a burst of equality selections, while taking idle times to do necessary write-back operations, which the authors call clean-up's. The performance of Burst ORAM worsens when there has been a large burst of queries, as explained in 6.1.1. After a burst the protocol performs a clean-up of writing new dummy DCs to the server. Before the start of every measurement, Burst ORAM was in a clean state with the full number of available dummy DCs. The performance of Burst ORAM during large bursts has been measured by performing prefix and range queries.

In PATCONFDB, there are global and local parameters to be set during the initialization of an outsourced database. Global parameters are set for the entire PATCONFDB setup, while local parameters are chosen for each ORAM instance. Local parameters can differ within the same PATCONFDB instance. For example: The parameter 'size of DC' will be smaller for ORAM index layers than for the ORAM record store, since index information is usually smaller than data records. The following parameters can be chosen for a PATCONFDB setup:

- **Number of ORAM index layers (global):** This parameter is directly dependent on the amount of storage available on the mediator. For best performance, the index structure is completely stored on the mediator, which would result in zero ORAM index layers. Since this is often not feasible in practice, we need to build hierarchical levels of ORAM index layers until the amount of storage needed for the location information of every DC in the uppermost ORAM index layer is smaller than the available storage on a mediator. In most scenarios one or two ORAM index layers are required.

- **Size of DC (local):** The size of a data block container (DC) determines how many data blocks fit into one DC. Depending on the ORAM instance it resides in, a data block can either contain a data record or index information about data records. Figure 14 shows the time it takes to insert a data block into a PathCONFDB record store depending on the size of the DCs. It can be seen that the best access times can be

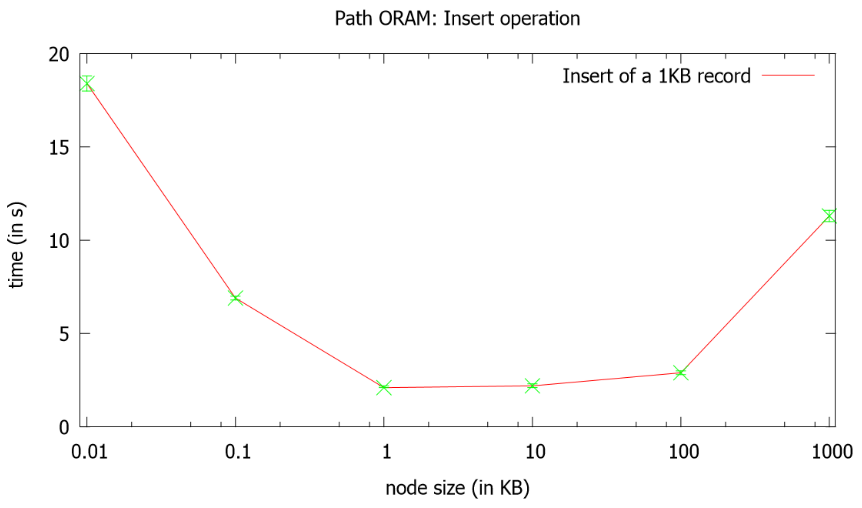


Figure 14: Measured time to insert a data record into a PathCONFDB record store depending on the size of DCs (node size).

achieved when the size of the DCs are set so that 1 to 100 data blocks fit into a DC. In combination with the measurements for the next parameter, we will argue that the optimal DC size for a PATCONFDB setup is reached when it can contain between 10 and 100 data blocks.

- **Number of levels / tree height (local):** Every ORAM scheme has some kind of data structure to store its DCs. In the case of schemes like Burst ORAM, DCs are stored within different levels, whereas schemes like Path ORAM store DCs in a binary tree. The total number of DCs that can be stored within an ORAM scheme is therefore dependent on the number of levels or the tree height accordingly. Figure 15 shows time it takes to insert the same data block into a PathCONFDB record store depending on the parameter 'tree height'. It can be seen that the time for an insert operation rises linear to the tree height. As a result of both measurements it can be seen that for an optimal result the size of DC should be set so that it fits close to 100 data blocks. Only when more storage space is needed, the number of levels or the tree height should be increased.

We instantiated both PathCONFDB and BurstCONFDB with two index layers and one record store. Our shuffled B-tree variant is implemented with a Shuffle Index [6] as described in Section 5.1 with two cover searches, a cache size of five and a Shuffle Tree height of three. In each tested scenario, all parameters are chosen to achieve the best performance while fulfilling all requirements for available database storage and maximum client storage. 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*.

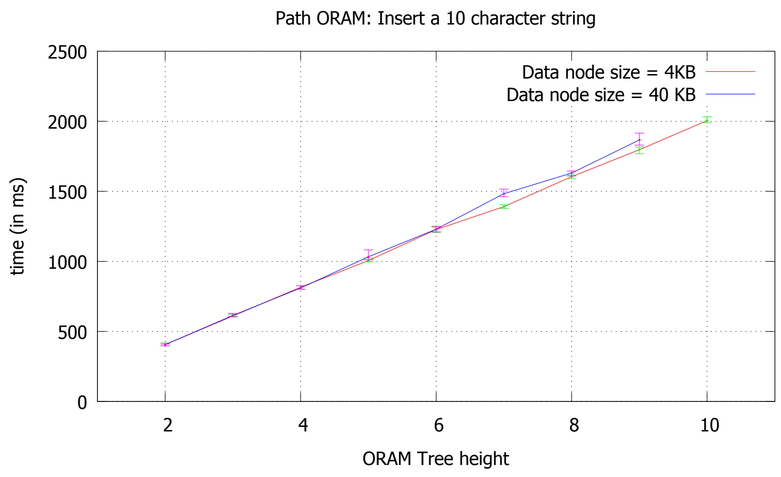


Figure 15: Measured time to insert a data block into a PathCONFDB record store depending on the tree height.

6.1 PATCONFDB with B-Tree Index Structure

We investigate the overheads that our proposed deployment concepts with a B-tree Index structure induce with regard to query latency (Section 6.1.1), network (Section 6.1.2) and storage overhead (Section 6.1.3).

6.1.1 Query Latency Overhead

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 data records that were used for evaluating the CPIs contain a string attribute and a number attribute. The number attribute 'age' is generated at random in an uniform distribution between the values 0 to 120. The string values have also been generated at random and equally distributed over the characters A to Z. That string was created until the data record size seen in Table 1 had been reached. 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.

Burst ORAM is an ORAM scheme which is designed to postpone the execution of its security mechanisms, which ensure APC, until an idle phase, in which no one is accessing the database. This is why for the remainder of this paper we categorize network traffic in *online I/O* and *offline I/O*. Online I/O is defined as the amount of network traffic that is needed until a client request can be answered. Offline I/O is defined as the amount of network traffic that is needed after a client access to execute security algorithms and to ensure APC. In a standard ORAM scheme like Path ORAM, the amount of Online I/O and the amount of Offline I/O are similar and they are both necessary for every single query. With the XOR-operation of Burst ORAM, its online I/O is usually only the size of one DC, whereas the Offline I/O needed to reinitialize all used dummy DCs is a lot higher. This makes it worthwhile to evaluate the performance of BurstCONFDB not only

	DB size	Records	Record size	Bandwidth and latency
Scenario 1	1 GB	1 mio	100 Byte	1 Gbit/s and 5 ms
Scenario 2	1 GB	1 mio	100 Byte	10 Mbit/s and 50 ms
Scenario 3	1 GB	10.000	10 KByte	1 Gbit/s and 5 ms
Scenario 4	1 GB	10.000	10 KByte	10 Mbit/s and 50 ms
Scenario 5	10 GB	10 mio	100 Byte	1 Gbit/s and 5 ms
Scenario 6	10 GB	10 mio	100 Byte	10 Mbit/s and 50 ms
Scenario 7	10 GB	100.000	10 KByte	1 Gbit/s and 5 ms
Scenario 8	10 GB	100.000	10 KByte	10 Mbit/s and 50 ms

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

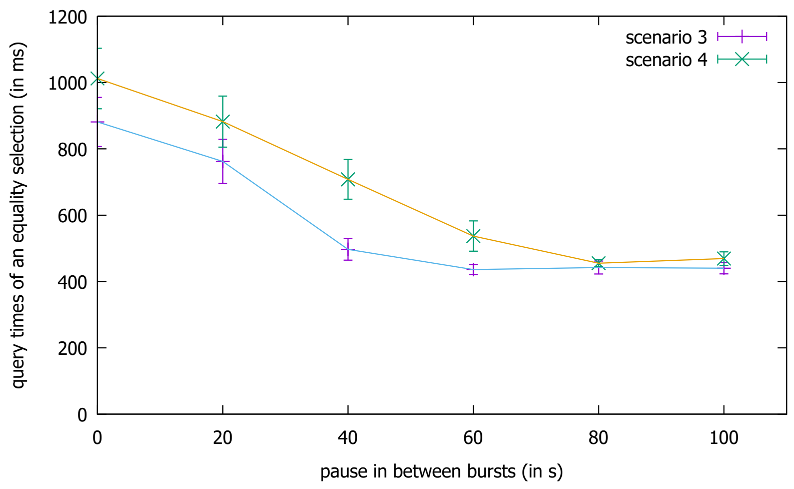


Figure 16: Measured query times of BurstCONFDB in scenarios 3 and 4 with different pauses between bursts.

with a steady stream of queries, but with bursts of queries of differing lengths and pauses in between queries. Figure 16 shows the average query latencies of 1000 queries after differing lengths of pauses between bursts. In both scenarios a local client storage of 500MB is used. Scenario 3 and 4 only differ in their network link. Whereas the network in scenario 3 has a throughput of 1GBit/s, the network link in scenario 4 only has 10Mbit/s. It can be seen that even after 40 seconds of pauses between bursts, the performance of Burst ORAM is still increasing due to a re-filling of the database with dummy DCs.

The query latency of equality selections measured over all scenarios is shown in Figure 17. 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 both PATCONFDB protocols 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 query attribute values. For both PATCONFDB protocols it takes a long time to sequentially query index layers which contain a large number of identifiers.

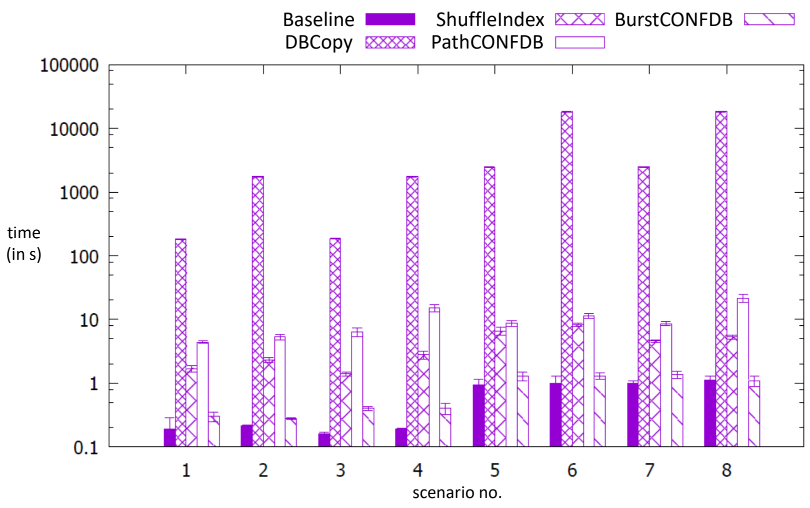


Figure 17: Measured query latency of equality selections over all scenarios on a logarithmic y scale.

The query latency of range selections measured over all scenarios is shown in Figure 18. Since more data is retrieved, the bandwidth of the network link influences the query latency significantly (scen. 1,3,5,7 vs. 2,4,6,8). Note that the confidence intervals for the queries executed with PathCONFDB are significantly larger than the ones of the other protocols. This is caused by different sized network overheads as explained in Section 6.1.2. For scenarios 6 and 8, PathCONFDB exceeded our time limit of 7h for one range query, while BurstCONFDB came close to this limit in scenarios 5 and 6. This indicates that BurstCONFDB has slower access times for range queries when there are a higher number of records, whereas PathCONFDB performs worse when there is a low network bandwidth available. Our Shuffle Index variant 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 both PATCONFDB protocols for range selections if more than about 0,1% off all records are queried in range selections. The results indicate that by performing an estimation of the number of queried records, it can be feasible to just download the whole database, rather than to execute Path or Burst ORAM. It can be seen that our Shuffle Index variant 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. This is the scenario in which our variant has a significantly lower query latency than the extension proposed by the authors of [21]. Since prefix and range selections are treated as a series of equality selections, the data throughput of their implementation remains the same when a range selection is performed. The data throughput of our solution is increased significantly, as shown in Section 6.1.2, specifically in Table 2.

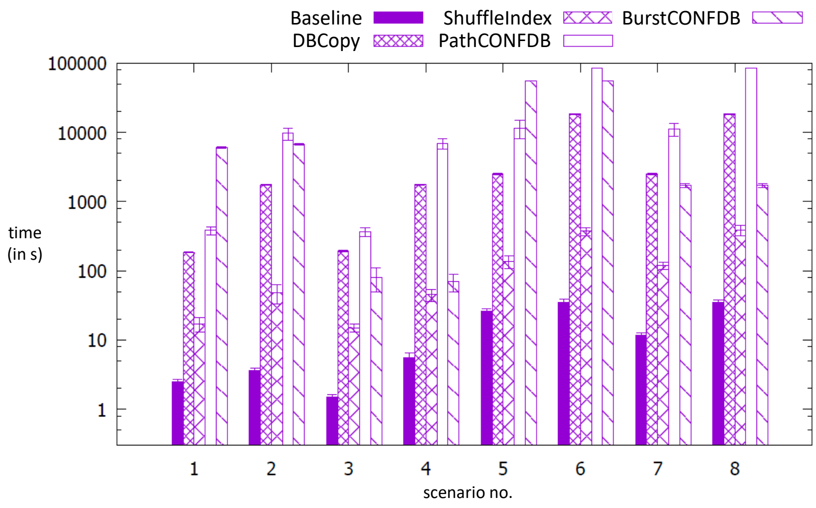


Figure 18: Measured query latency of range selections over all scenarios on a logarithmic y scale.

6.1.2 Network Overhead

In the following, we investigate the network overhead of PathCONFDB, BurstCONFDB 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 for PathCONFDB, BurstCONFDB and shuffled B-trees. Then we present and interpret the network overheads we measured for the scenarios that we introduced in Section 6.1.1.

The total network traffic (in Byte) N_P^s to retrieve a record from any **PATCONFDB** instance 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 denotes the number of index layers, N_r^s denotes the network traffic induced by the retrieval of a record from the record store depending on the ORAM scheme s, N^t denotes the size of the root node of the PATCONFDB B-tree in Byte, N_i^s denotes the network traffic in Byte induced by the retrieval of a record from an index layer depending on the ORAM scheme s.

For an implementation of PathCONFDB, 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.

For an implementation of BurstCONFDB, the retrieval of a single record causes a lot less network overhead. Through the XOR-technique only one DC from every Burst ORAM layer has to be retrieved. When we consider all necessary cleanup operations known as off-line I/O, the network overhead increases significantly, up to a factor of 25 for a single Burst ORAM layer [5].

Scenario	SI(EQ)	SI(PR)	PC(EQ)	PC(PR)	BC(EQ)	BC(PR)
1 + 2	562,0	6,56	7080	6557	147,9	9151
3 + 4	110,4	6,31	683,1	669,8	30,4	439,7
5 + 6	1460	6,57	5901	5761	132,1	7843
7 + 8	111,3	6,26	1025	971,3	28,1	647,2

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

The total network traffic N_S (in Byte) 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 denotes the height of Shuffle Tree, r denotes the root node size in Byte, n denotes the navigation node size in Byte, d denotes the data node size in Byte, c denotes the number of executed cover searches, a denotes the 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 6.1.1 are shown in Table 2 for the Shuffle Index, PathCONFDB and BurstCONFDB. 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 BurstCONFDB the equality selection is done so that there is no cleanup operation necessary, whereas the prefix query reflects the network overhead including the clean up process.

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 all schemes (scen. 1+2,5+6 vs. 3+4,7+8). Since the total network traffic for equality selections in all schemes remains low (smaller than 1,5MB), the large network overhead does not have a similarly large impact on the query latency measured in Section 6.1.1. In the case of prefix selections, the performance optimization proposed in Section 5.1 for our Shuffle Index variant reduces the relative network overhead significantly. Since the extended Shuffle Index as proposed in [21] executes prefix and range selections as a series of sequential equality selections, there is no significant reduction of the relative network overhead for this approach. This is also true for the PATCONFDB concept. 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.

6.1.3 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 records, 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 with Path ORAM 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. A PATCONFDB instance with Burst ORAM that is filled to its maximum capacity has a higher storage overhead because the Burst ORAM scheme requires at least half of the DCs to be dummy elements. So the overhead for record store and index layers increases up to a factor of 4,4 for a database filled to its maximum capacity. 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 lexicographically 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 all 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.

6.2 Comparison of B-Tree and Bitmap Index Structures

To ensure Access Pattern Confidentiality (APC), a large amount of overhead in computation or network traffic is induced, which significantly decreases the performance in certain scenarios. To achieve a more thorough understanding of which methods can be used to achieve a better performance by leveraging the structure of data (tunable security), we analyze the impact of different index structures depending on the outsourced data.

Since the bitmap index works fundamentally different than a B-tree index structure, we had to make some adjustments so that the two index structures are comparable while still using them like they would be used in real world scenarios. This leads to some requirements for our evaluation in Section 6.2:

1. Both index structures are only used for scenarios that contain low-cardinality attributes.

2. Both index structures are not outsourced nor encrypted; they always remain on the mediator, therefore there are no outsourced index layers.

3. Once the index is traversed all matching queries are executed sequentially to hide the type of query and the number of queried record sets.

To the best of our knowledge, there is no usage scenario for outsourcing the bitmap index structure. Outsourcing the index structure is only feasible, if it is too large to be stored on the mediator. If this is the case, it also can't be fully retrieved from the SP for every access, because then the mediator would need to hold available the same free storage space as if it would just store the whole index structure in the first place. So we need to establish some kind of hierarchy or distribution, which brings us back to a structure similar to a B-tree. It might however be feasible to store the index structures of some attributes in a bitmap index structure on the mediator to reduce access times. This is why we compare the index structures in this setting.

Figure 19 shows the average query latencies of a B-tree and a bitmap index structure measured in PathCONFDB and BurstCONFDB. The x-axis shows the number of distinct attribute values used for the measurement. The tests are set up as described in Section 4.2. It can be seen that for a low number of distinct attribute values a B-tree and a bitmap index structure have almost the same query latencies for both ORAM schemes. As the number of distinct attribute values rises, the query latency of a bitmap index structure rises significantly, whereas the query latency of the B-tree based ORAM schemes remains low. Even though the bitmap index structure performed slightly better for a small number of distinct values, there is almost no visible difference between the index structures for low numbers, since the index lookup only takes a small fragment of the query latency. Most of the query time results from network traffic and computation (encrypting and decrypting), as can be seen by the big difference in performance between the Burst ORAM and the Path ORAM implementations of both index structures. It is a good idea to not outsource small indexes but to keep them on the mediator, since they only take up little space and access times are orders of magnitude faster. In this case, the bitmap has significantly lower access latencies for indexes with a low number of distinct attribute values. However, these index lookup times are marginal compared to the much higher access times of data records in all PATCONFDB protocols. It can be concluded that it is not feasible to use a bitmap index structure in the PATCONFDB framework even for a very small number of distinct attribute values. An evaluation of the storage or network overhead is not feasible since the bitmap index structure was not outsourced onto the SP.

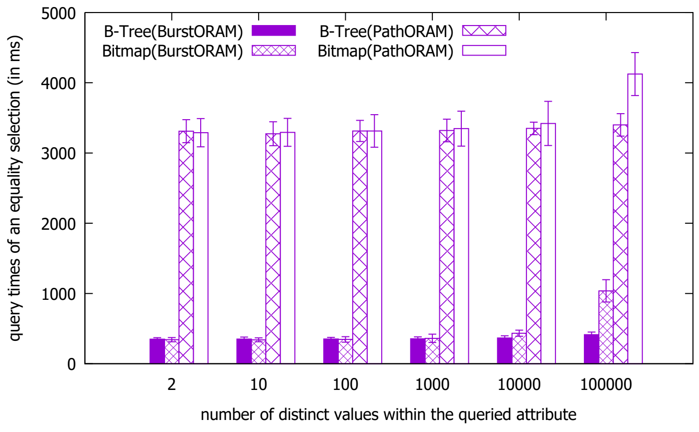


Figure 19: Measured query latency of a B-tree and bitmap index structure depending on the number of distinct attribute values.

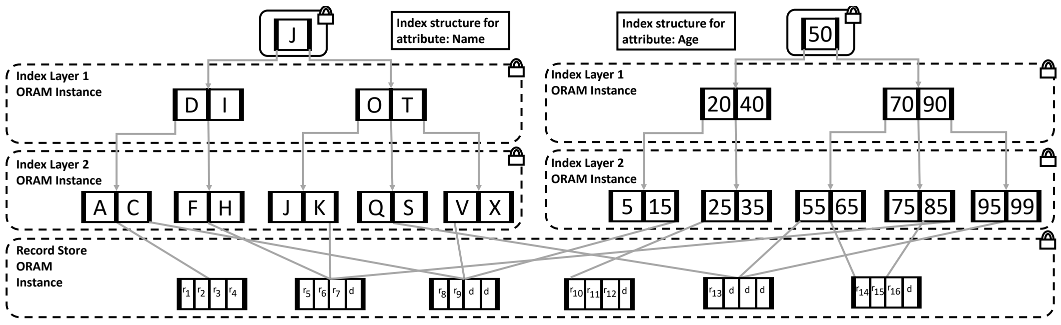

Index structure for attribute: Name
Index structure for attribute: Age
Index Layer 1 ORAM Instance
Index Layer 2 ORAM Instance
Record Store ORAM Instance
J
50
D I
O T
20 40
70 90
A C
F H
J K
Q S
V X
5 15
25 35
55 65
75 85
95 99

Figure 20: Example of a PATCONFB setup with the two index structures 'Name' and 'Age'.

7 Functionality Extensions

We address the challenges in complying to access pattern confidentiality in DaaS scenarios with **multiple types of indexes**, i.e. strings, numbers, and dates, for the use of the PATCONFDB approach. In Figure 20 an example of a scenario with two index structures for the attributes 'Name' and 'Age' is shown. The index structures are kept in separate ORAM instances, which both have pointers to records in the same record store.

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 updated, 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 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.

8 Conclusions & Future Work

To investigate the feasibility of access pattern confidentiality preserving index structures, we proposed PATCONFDB, an ORAM-based concept to achieve access pattern confidentiality, and provided variants of prefix and range selections for existing shuffled B-tree approaches. In particular, empirical measurements of these concepts showed that enforcing access pattern confidentiality through confidentiality-preserving index structures in relational databases only induces an overhead in query latency of factor 1.5 for evaluating equality conditions on a data set of up to 10 million records, compared to a plain text database. We showed that BurstCONFDB has lower query latencies for equality selections than the extended Shuffle Index approach, while at the same time delivering better security guarantees. Since there are many scenarios in which an equality selection to retrieve data records is sufficient, one can already profit from the security guarantees of PATCONFDB, while inducing a very reasonable overhead in query times and server storage. Our Shuffle Index variant, however, induces only a small combined overhead in query latency of about factor 5.2 (average over all test cases) for a setup with equality and range selections, but does not yet provide a strict and well-defined access pattern confidentiality guarantee. This means that shuffled B-tree approaches remain a viable alternative for scenarios in which large amounts of data is accessed frequently and the strict security guarantees are not required. Our performance evaluation showed to which extend the induced overhead in query latency depends on the network link to the SP, the structure of the data and the query workload of the outsourcing scenario. It can also be seen that an ORAM scheme like Burst ORAM can have a very low query latency for a single equality selection and is therefore very well usable in some scenarios, the benefit comes with the drawback of more network traffic and more storage overhead on client and on server.

The findings of this paper highlight multiple future research directions. It is worthwhile to aim for further efficiency improvements of ORAM-based index structure. The superior efficiency of shuffled B-tree approaches makes it also worthwhile to aim for an advanced understanding of their security guarantees. The availabilty of various implementations of shuffled B-tree schemes brings us closer to a tunable security scheme, in which users can choose the security requirements of specific database columns that are then outsourced with a different CPI scheme. Furthermore, we plan to investigate how both ORAM-based and shuffle-based schemes can be extended to provide access pattern confidentiality-preserving indexes also for the case of relational databases with multiple index types and queries that include more complex operations.

Acknowledgements

Special thanks to Jens Koehler and Jan Grashoefer for their help in the refinement process of this paper as well as to the anonymous reviewers of our previous paper [8] and of the recent one for the feedback, insights, and very helpful comments.

References

[1] E. Bacis, S. D. C. di Vimercati, S. Foresti, S. Paraboschi, M. Rosa, and P. Samarati. Distributed shuffle index in the cloud: Implementation and evaluation. In *Proc. of the IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)*, 2017.

[2] 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.

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

[4] 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.

[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] A. Degitz, J. Koehler, and H. Hartenstein. Access pattern confidentiality-preserving relational databases: Deployment concept and efficiency evaluation. In *Proc. of the 9th International Workshop on Privacy and Anonymity in the Information Society (PAIS)*, 2016.

[9] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs. Onion oram: A constant bandwidth blowup oblivious ram. In *Theory of Cryptography Conference*, pages 145–174. Springer, 2016.

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

[11] 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.

[12] 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.

[13] 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.

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

[15] L. Li and A. Datta. Write-only oblivious ram-based privacy-preserved access of outsourced data. *International Journal of Information Security*, 16(1):23–42, 2017.

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

[17] 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.

[18] 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.

[19] 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.

[20] J. Zhang, W. Zhang, and D. Qiao. Mu-oram: Dealing with stealthy privacy attacks in multi-user data outsourcing services. *IACR Cryptology ePrint Archive*, 2016:73, 2016.

[21] J. Zhang, W. Zhang, and D. Qiao. Shuffle Index: Efficient and Private Access to Outsourced Data *ACM Transactions on Storage (TOS) - Special Issue USENIX FAST 2015*, Volume 11 Issue 4, Article 19, November 2015.