CODBS: A cascading oblivious search protocol optimized for real-world relational database indexes

Rogério Pontes*‡, Bernardo Portela*‡, Manuel Barbosa*§, Ricardo Vilaça†‡
*University of Porto, †University of Minho, ‡INESC TEC, §NOVA LINCS

***Abstract*—Encrypted databases systems and searchable encryption schemes still leak critical information (e.g.: access patterns) and require a choice between privacy and efficiency. We show that using ORAM schemes as a black-box is not a panacea and that optimizations are still possible by improving the data structures.**

We design an ORAM-based secure database that is built from the ground up: we replicate the typical data structure of a database system using different optimized ORAM constructions and derive a new solution for oblivious searches on databases. Our construction has a lower bandwidth overhead than state-of-the-art ORAM constructions by moving client-side computations to a proxy with an intermediate (rigorously defined) level of trust, instantiated as a server-side isolated execution environment.

We formally prove the security of our construction and show that its access patterns depend only on public information. We also provide an implementation compatible with SQL databases (PostgresSQL). Our system is 1.2 times to 4 times faster than state-of-the-art ORAM-based solutions.

I. Introduction

Symmetric searchable encryption (SSE) schemes are used to address the problem of outsourcing private databases to an untrusted third-party. These schemes enable a client to store encrypted data in a third-party and evaluate queries remotely over ciphertexts. Conceptually, SSE schemes create an encrypted index that maps keywords to a set of documents identifiers, with each identifier pointing to an actual document. The index as well as the documents are encrypted by the client and stored on the remote server. The client can query the index by generating cryptographic tokens for a specific keyword. Given a token as an input, the server can search the index and find the set of documents that contain the queried keyword without having to decrypt any data. However, SSE schemes still disclose confidential information, for example the access patterns revealed by a query, which can be exploited by statistical analysis attacks [1], [2].

One approach to address the leakage of SSE schemes is to store the server-side index and document storage in an Oblivious Random Access Machine (ORAM) scheme [3]. An ORAM scheme is protocol that enables a client to store and fetch a data block from an array structure that can store at most N data blocks. For each operation, the protocol ensures that the remote server does not learn neither the client operation nor the real location of the blocks. However, ORAM schemes have a few drawbacks. First the client has to maintain a position map (pmap) that tracks the location of blocks and a stash to hold temporary blocks. Secondly a remote access has a high bandwidth blowup, i.e., for every real access to the remote server, multiple blocks are transferred to the client [4], [5].

The overhead of ORAM schemes can be minimized by using an isolated execution environment (IEE) [6] co-located with the encrypted index on the server-side. An IEE is a trusted hardware technology that enables the execution of arbitrary (verifiable) computations in a clean slate. The internal state of an IEE is assumed to be isolated from other co-located processes, including operating systems and hypervisors. Intel's Software Guard Extensions (SGX) [7] is a prominent instance of an IEE that is widely used to develop novel solutions due to its ubiquity and accessibility in commodity hardware.

Previous Work. Combining ORAM primitives with trusted hardware mitigates some overhead of ORAM schemes but it's still not sufficient to create efficient private databases. Existing work proposes new search algorithms and oblivious primitives that lower even further query latency and the bandwidth used in the client-server communication. *Wang et al.* [8] initially proposed an oblivious data structure (ODS) to search data inside an ORAM scheme. This construction stored a binary search tree in an ORAM scheme and was able to search for a value with $\mathcal{O}(N \cdot \log(N))$ bandwidth blowup. Additionally, this construction minimizes the client side state of ORAM schemes by storing the position map in the remote server alongside the search tree.

The idea of ODS has been further used and developed by different systems. More concretely, Oblix [9] uses an oblivious search tree to index the keywords of a database and POSUP [10] uses an oblivious linked list to store the keywords as well as the documents. In fact, both of these systems improve on the early work of oblivious data structures (ODS) proposed by *Wang et al.* [8]. Besides these previous examples that focused only on SSE systems, there are also fully-fledged oblivious database solutions such as Opaque [11] and ObliDB [12]. Nonetheless, existing systems often use ORAM algorithms as black-boxes and do not optimize the internal data structures.

Motivation. Our goal is to create a novel oblivious search scheme tailored-fit for relational databases that has minimal bandwidth usage and client-side state. To achieve this goal we dive into ORAM schemes instead of using them as black-boxes and propose a new search scheme from the ground up that is optimized for database indexes.

Contributions. We propose a novel oblivious search scheme inspired by *Wang et al.* tree-based ODS [8]. The search scheme improves over existing work in several key aspects. In comparison to POSUP it does not require auxiliary data structures to keep a relation between keywords and ORAM addresses. Furthermore, our scheme searches over keywords

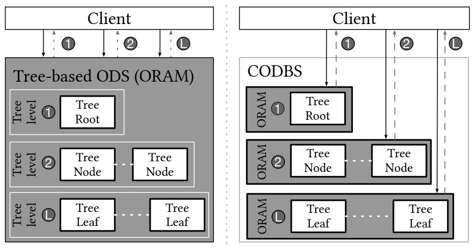


Fig. 1: Search tree with L levels stored in two different ORAM constructions: a tree-based ODS as proposed by *Wang et al.* [8] and the CODBS scheme presented in this paper.

and reduces the position maps to a small constant. Our system is also closely related to Oblix [9] tree-based ODS but our construction has a lower bandwidth blowup. The proposed scheme is only a subcomponent of our new oblivious relational database system architecture. We designed this system to outsource all processing load and storage from the client to a thin proxy, an Intel SGX IEE co-located with the database engine. We prove the security of our solution with a classical game-based proof and measure its performance. We make the following contributions:

- We propose CODBS, a novel tree-based oblivious search scheme to store database indexes. This scheme originated from the observation that an oblivious search on a tree-based ODS touches every tree level once in the same order. As such, it is clear that a balanced tree-based ODS only needs to hide which node is accessed in a level and not the level accessed. From this insight, we split the search tree into L smaller ORAM instances, rather than a large ORAM, where L is the search tree height. This modification reduces the bandwidth blowup of *Wang et al.* tree-based ODS from $\mathcal{O}(N \cdot \log(N))$ to $\mathcal{O}(L^2/2)$, here N is the number of data blocks (depicted in Figure 1).
- We propose Forest ORAM, an optimized ORAM construction to store database tables. This construction reduces the bandwidth blowup of OblivStore's partition framework [13].
- We present an optimized oblivious database architecture and implemented a complete solution on top of PostgreSQL.
- We measure the average system throughput, latency and resource usage of our solution with YCSB [14]. With the evaluation we validated the asymptotic improvements of our construction and shown a $\sim 2\times$ to $\sim 4\times$ performance improvement over state-of-the-art constructions that leverage Path ORAM and oblivious data structures [8], [9].

II. Problem definition

A. Database System Model

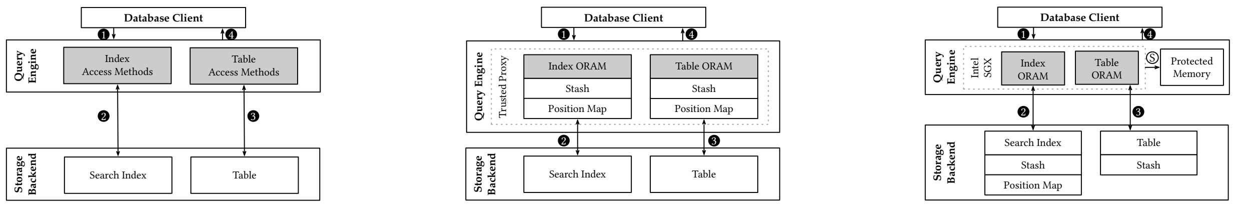
Databases have multiple data structures and query operations to select, filter and join data. We focus on minimizing the information disclosed by a fundamental operator of relational databases, the *index scan*. By protecting index scans, other operators can inherit its security guarantees. To understand an index scan operation, we present a high-level model of a database architecture in Figure 2a; we consider three main components: a *Database Client*, a *Query Engine* and a *Storage Backend*. In this model, the *Database Client* is a remote application that connects users to the *Query Engine*. The actual query processing is handled by the *Query Engine*, the most computationally intensive component. This component is stateless and stores block-based data structures on the *Storage Backend*. The *Storage Backend* abstracts the underlying storage and contains two data structures, a *Search Index* and a database *Table*. We consider the index a B^+-tree [15] that maps keywords to table records. The database *Table* is a linked list of blocks with each block holding a subset of records.

The execution of an index scan starts with the *Database Client* sending a query to the *Query Engine* (Figure 2a-❶). The input query is intercepted by the *Query Engine* which generates a query plan describing the database tables and indexes that must be accessed and the order of the accesses. The *Query Engine* executes an index scan by searching a tree-based index (Figure 2a-❷). This index search results in a subset of table pointers that satisfy an input query. For each pointer, the *Query Engine* retrieves its matching table record (Figure 2a-❸) and stores it in a result set. The execution flow between *Query Engine* and the *Storage Backend* is repeated until every relevant record is accessed and the complete result set is sent to the *Database Client* (Figure 2a-❹).

B. Leakage sources

The execution of a database query has two sources of leakage. The first are the access patterns of the query engine during its accesses the database storage (Figure 2a-❷,❸). Every access from the *Query Engine* to the *Storage Backend* consists of either reading or writing a data block in one of the databases' data structures. The sequence of blocks accessed during the tree transversal define a unique path that identifies a small subset of data records. The set of possible results is shortened even further by the identity of the blocks accessed on the table storage, as each table block contains a limited number of database tuples. Besides the access patterns, an adversary can also learn critical information just from the number of accesses from the table index to the table storage. The information disclosed by these accesses is captured by the second leakage considered in our model, the volume leakage. As demonstrated previously, volume leakage is sufficient to compromise an encrypted database [16], [17].

Our approach to mitigate these leakage sources is to propose ORAM-based solution optimized for relational databases that are capable of fully leveraging a *Trusted Proxy* deployed at an intermediate level of trust. As depicted in Figure 2b, in a naive solution the *Trusted Proxy* can be thought of as an interactive oblivious protocol that manages two position-based ORAM constructions and keeps all of the client side state inside the protected environment (stash and position map). One of the ORAM constructions stores the database index, while the other



(a) Plaintext Database (b) Naive Oblivious Database (c) Optimized Oblivious Database

Fig. 2: System models of a plaintext database, a naive oblivious solution and an optimized oblivious system.

stores the indexed table. We detail the trust model considered in the paper and our optimizations to the naive approach next.

C. *Trust Model*

We consider a semi-honest adversary that can observe all communications and computation activity, with the exception of those occurring inside the *Database Client* and *Trusted Proxy*. Concretely, this implies knowledge of: i.) messages exchanged between client and proxy (Figure 2c-❶,❹); ii.) proxy interactions with external memory (Figure 2c-Ⓢ); and iii.) proxy interactions with the storage (Figure 2c-❷,❸).

We assume that client-to-proxy interaction (Figure 2c-❶,❹) is preceded by a key exchange protocol, to establish a secure channel. This allows our system to rely on standard cryptographic techniques to protect the confidentiality of messages exchanged between client and proxy. Instrumenting IEE-enabled code in this way is a common requirement, and has been shown to be achievable securely with minimal performance overhead [18]. However, secure channels still disclose the size, direction and number of messages.

An underlying issue of IEE-enabled systems is the information disclosed in proxy interactions with external memory (Figure 2c-Ⓢ). We assume the trusted hardware only protects the memory contents [19], [20], but not the access patterns. Our protocol tackles this issue with constant-time implementations [9], [21]. The leakage that remains are the access patterns (Figure 2c-❷,❸) in the proxy-to-storage interface. We assume the adversary has full knowledge of the data blocks accessed in the external storage.

D. *Optimization approach*

We now refine the high-level model and detail the system architecture used in this paper, along with an overview of our optimizations. Our system is a relational database outsourced to a third-party infrastructure, as depicted in Figure 2c. The *Trusted Proxy* is hosted in an Intel SGX enclave that supports the creation of genuine IEEs that can be successfully authenticated with an attestation service. The *Trusted Proxy* and the *Query Engine* are co-located on the same third-party server, effectively lowering the inter-component latency to a minimum. We are agnostic with respect to the *Storage Backend* but we assume that it provides a standard I/O POSIX interface. In our model the *Query Engine* manages client connections, reroutes input client requests to the *Trusted Proxy* and provides an interface to read/write blocks from the database storage. The *Trusted Proxy* executes the search queries and keeps an internal secret state with the secret keys used to encrypt/decrypt blocks from the database storage.

Using an Intel SGX enclave as a *Trusted Proxy* is challenging as enclaves have a limited pool of protected memory available. Current technology is restricted to 128 MiB but only 93 MiB can actually be used to store and read application data. We address this limitation with two optimizations. First, we keep the enclave as thin as possible by moving the stash and position maps of ORAM schemes to the *Storage Backend*. Secondly, we do not simple use ORAM schemes as black-boxes and instead use CODBS, our novel oblivious search scheme which has multiple composable ORAMs that reduce the client (here proxy-side) storage to a constant factor and enables the protected proxy to function with a small local memory, while guaranteeing leakage-free storage access.

III. Definitions

In this section we present the notation and security definitions used throughout this work. The security parameter is denoted by λ in unary (i.e., 1^λ). A negligible function in the security parameter is denoted as $\mathsf{negl}(\lambda)$. We consider an adversary $\mathcal{A}$ and a simulator S to be polynomial time algorithms. Our constructions rely on notions of a variable-length-input pseudo-random function (PRF) and a symmetric encryption schemes secure against chosen plaintext attacks (IND-CPA) [22]. The secret keys are uniformly sampled from $\{0,1\}^\lambda$.

Databases. We denote a plaintext database as a set of data records indexed by a search key $DB = \{(key_1, data_1)\dots(key_n, data_n)\}$. We abstract the search keys as keywords from the set of all finite strings $\mathcal{W} \subseteq \{0,1\}^*$ and the data records $data_i \in \{0,1\}^B$ as binary data blocks of fixed length B. A database query $\tau : \mathcal{W} \to \{0,1\}$ is a predicate that consists of keywords in the domain $\mathcal{W}$ that satisfies a boolean formula. Given an input query τ a database search $DB(\tau) = \{data_i : \tau(key_i) = 1\}$ returns all data records that satisfy the query.

The database keys and data records are stored in a pair of data structures where $\mathcal{I}$ denotes a tree-based index that stores the database search keys and $\mathcal{T}$ denotes a *Table Heap* with the data records. The *Table Heap* is defined as a collection of N table blocks $\mathcal{T} = \{(a_1, data_1), \dots, (a_n, data_n)\}$ associated with a unique address $a_i \in \mathbb{Z}$. The tree-based index $\mathcal{I}$ abstracts the search tree indexes of databases as a collection of L levels, each one storing multiple tree nodes. A tree node in a tree level is defined by a list of tuples (key, a) where key is a search key and a is an address to either another node in a tree level or to a table block in a *Table Heap*. We denote access to the data structures with array notation where a *Table Heap* access returns a table block $data \leftarrow \mathcal{T}[a]$ and a *Table Index* access at level l and address a returns the list of pointers $(key, a) \leftarrow \mathcal{I}[l][a]$.

A. Oblivious Index Scan

Using the previous notation, the database operations are captured by the Oblivious Index Scan (*OIS*) scheme, which is realized by our main construction. This primitive uses ORAM schemes as building blocks to store the database data structure and hide the access patterns of search queries. Intuitively, an OIS scheme starts with an empty data storage, which the *Database Client* fills by outsourcing a plaintext database structure via the *Trusted Proxy* with an initialization algorithm. After this initial step, the *Database Client* sends queries to the database engine to retrieve the database records.

Definition 1. (Oblivious Index Scan) An oblivious index scan scheme OIS consists of the following two algorithms:

- Init(1^λ, $\mathcal{I}$, $\mathcal{T}$, $prms$)$\rightarrow$(st, $\tilde{\mathcal{I}}$, $\tilde{\mathcal{T}}$): Initialization algorithm that takes as input a *Table Index* $\mathcal{I}$, a *Table Heap* $\mathcal{T}$ and the public database parameters $prms$: (number of blocks N, tree-based index height L, and tree fanout d). The algorithm returns an internal state st, an oblivious search tree $\tilde{\mathcal{I}}$ and an oblivious table $\tilde{\mathcal{T}}$. The oblivious data structures preserve the indexing relation between the input data structures. The internal state is kept securely within the *Trusted Proxy* and it contains the internal state of multiple ORAMs, a secret key for a symmetric encryption scheme and a secret key of a PRF. The oblivious data structures are stored in the *Storage Backend*.
- Search(st, $\tilde{\mathcal{I}}$, $\tilde{\mathcal{T}}$, τ)$\rightarrow$(st', $\tilde{\mathcal{I}}'$, $\tilde{\mathcal{T}}'$, $data$): Search algorithm that takes as input the current state st, an oblivious *Table Index* $\tilde{\mathcal{I}}$, a oblivious *Table Heap* $\tilde{\mathcal{T}}$ and an input query τ. The algorithm filters the records that satisfy the query with an Oblivious Index Scan and returns an updated state st', a shuffled oblivious *Table Index* $\tilde{\mathcal{I}}'$, a permuted oblivious *Table Heap* $\tilde{\mathcal{T}}'$ and the resulting $data$ record.

B. Oblivious RAM

We follow the classical definition of position-based ORAMs [4], [8] where a client (e.g.: local machine) remotely accesses data blocks in a server (e.g.: block storage) but modify it in two ways. First, instead of providing a single Access method that reads data from the server, shuffles the blocks and flushes them back, we divide these processes in two distinct functions. Secondly, we explicitly require an external position map δ to be passed as input for every oblivious access. Similar to internal position maps, the external position map keeps track of the current location of blocks. However, the external position map also determines the next location where a block must be stored after an oblivious access. As such, the responsibility of correctly book-keeping the location of the blocks is shifted to the ORAM client.

Definition 2. (Oblivious RAM) An oblivious RAM scheme consists of the following three algorithms:

- Build(N)$\rightarrow$(st, $\tilde{\mathcal{D}}$): Initialization algorithm that takes as input a maximum number of blocks N and outputs an internal state st and an initialized data structure $\tilde{\mathcal{D}}$.
- Read((st, δ), $\tilde{\mathcal{D}}$, a)$\rightarrow$(st', $data$): Access operation that takes as input an internal ORAM state st, an external position map δ, the external data structure $\tilde{\mathcal{D}}$ and a block address a. It returns an updated state st' and the external block $data$. This operation does not evict the ORAM internal state nor modifies the external data structure.
- Write((st, δ), $\tilde{\mathcal{D}}$, a, $data$)$\rightarrow$(st', $\tilde{\mathcal{D}}'$): Eviction operation that takes as input an internal state st, an external pmap δ, a data structure $\tilde{\mathcal{D}}$, a block address a and the new block $data$. It evicts stashed blocks, writes $data$ to offset a and returns an updated state st' and data structure $\tilde{\mathcal{D}}'$.

IV. Oblivious Cascading Scans

Overview. In this section we present our Cascading Oblivious Database Search (CODBS) construction. Intuitively, the scheme captures the interaction between the *Trusted Proxy* and the *Storage Backend*. The client starts by issuing an initialization query to the *Trusted Proxy* in order to outsource a local plaintext *Table Heap* and a plaintext *Table Index* to a sequence of $L + 1$ levels of independent ORAMs. Our construction stores the database blocks across each level by following the pattern that emerges naturally from the tree-based indexes in databases such as B^+-trees. As such, every node of a *Table Index* at level l is stored on the ORAM level l. The last ORAM level is reserved for the table blocks. The underlying ORAMs are devoid of an internal pmap and instead we explicitly provide a pmap for every ORAM access. In fact, the locations of the blocks B in an ORAM at level $l + 1$ are stored in its parent node A at level l. As defined in Section III, each plaintext tree node has a list of tree points (key, a) which is enhanced during the initialization process with an additional counter (key, a, ctr). These counters keep the access to the ORAM levels correct and secure.

After the initialization, the client proceeds to issue search queries to the *Trusted Proxy*. With the multiple ORAM levels, a query search consists of cascading from level to level and choosing the next node to access at each step. In the last level the *Trusted Proxy* returns to the client a single *Table Heap* block that satisfies the input query. To gain an intuition on how search moves from level to level, consider the following example of an access to a block A at level l. Before moving to a level $l + 1$, the scheme seeks in the block A a pointer (key, a, ctr) to a child node that satisfies its query. If a match is found, the location of the next block to access is calculated with a PRF by providing as input the ORAM level $l + 1$, the address a and the counter ctr. However, before moving to the next level $l + 1$, the counter of the matching pointer is updated and block A is shuffled back in level l to a new position. This combination of independent ORAM levels with PRFs is a significant improvement over state-of-the-art tree-based ODS [8] that can only flush the tree nodes after iterating over the entire search tree. With CODBS, a tree search only needs to do a single pass over the entire tree and the tree nodes can be flushed to the ORAM immediately after being fetched.

CODBS in detail. Given a PRF $\mathbf{F} : \{0,1\}^{\lambda} \times \{0,1\}^{*} \to \{0,1\}^{\lambda}$, an IND-CPA symmetric encryption scheme $\Theta = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ and a position-based ORAM scheme Φ with an external pmap, CODBS is defined by the initialization Algorithm 1 and the search Algorithm 2. In this section we abstract the ORAM implementation, but we later present an optimized construction in Section IV-B. In CODBS the location of a block is provided by *location tokens*, i.e.: two outputs sampled from a PRF. The block location is defined by a tuple $(F(m), F(m'))$ containing a token for its current location and a token for its eviction location. These tokens are used by the ORAM scheme to move a block from its original address to a new address after an oblivious access. For instance, assuming the underlying ORAM scheme is a construction similar to Path ORAM [4] the tokens are used to compute uniformly random leaves in the server's binary tree.

Algorithm 1: CODBS Init protocol

```
1  Function Init($1^{\lambda}$, $\mathcal{I}$, $\mathcal{T}$, $N$, $L$, $d$)
2      $\mathsf{sk}_F \leftarrow F.\mathsf{KGen}(1^{\lambda})$;$\mathsf{sk}_E \leftarrow \Theta.\mathsf{KGen}(1^{\lambda})$
3      $(\mathsf{st}_{\tilde{\mathcal{I}}}, \tilde{\mathcal{I}}) \leftarrow \mathsf{InitSearchTree}(\mathcal{I}, \mathrm{L}, \mathrm{d}, \mathsf{sk}_F, \mathsf{sk}_E)$
4      $(\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow \Phi.\mathsf{Build}(N)$
5      for $a \in \{0, \ldots, N\}$ do
6          $\delta \leftarrow (F(\mathsf{sk}_F, L+1||a||0),\ F(\mathsf{sk}_F, L+1||a||1))$
7          $c \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, \mathcal{T}[a])$
8          $(\mathsf{st}_{\tilde{\mathcal{T}}}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a)$
9          $(\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a, c)$
10     return $((\mathsf{sk}_F, \mathsf{sk}_E, 1, \mathsf{st}_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}}), (\tilde{\mathcal{I}}, \tilde{\mathcal{T}}))$
11 Function InitSearchTree($\mathcal{I}$, $L$, $d$, $\mathsf{sk}_F$, $\mathsf{sk}_E$)
12     $\tilde{\mathcal{I}} \leftarrow []$; $\mathsf{st}_{\tilde{\mathcal{I}}} \leftarrow []$
13     for $l \in \{0, 1, \ldots, L\}$ do
14         $(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}_l) \leftarrow \Phi.\mathsf{Build}(d^l)$
15         for $a \in \{0, 1, \ldots, d^l\}$ do
16             data $\leftarrow \mathcal{I}[l][a]$
17             data' $\leftarrow$ []
18             for $i \in \{0, 1, \ldots, |data|\}$ do
19                 $(key, a') \leftarrow data[i]$
20                 $data'[i] \leftarrow (key, a', 1)$
21             $\delta \leftarrow (F(\mathsf{sk}_F, l||a||0),\ F(\mathsf{sk}_F, l||a||1))$
22             $c \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, data')$
23             $(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a)$
24             $(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}_l) \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a, c)$
25         $\tilde{\mathcal{I}}[l] \leftarrow \tilde{\mathcal{I}}_l$; $\mathsf{st}_{\tilde{\mathcal{I}}}[l] \leftarrow \mathsf{st}_{\tilde{\mathcal{I}}_l}$
26     return $(\tilde{\mathcal{I}}, \mathsf{st}_{\tilde{\mathcal{I}}})$
```

Initialization algorithm. The Init algorithm outsources a plaintext database to a pair of oblivious data structures stored in an untrusted server. The goal of this algorithm is to initialize the *Trusted Proxy* internal state and ensure the database is ready to process client queries. The algorithm starts with the generation of secret keys (line 2) and then proceeds to create two additional oblivious structures, an oblivious search tree $\tilde{\mathcal{I}}$ (line 3) and an oblivious table $\tilde{\mathcal{T}}$ (line 4-9). The oblivious search tree $\tilde{\mathcal{I}}$ is the result of the algorithm $\mathsf{InitSearchTree}$. This tree initialization algorithm traverses the plaintext database tree level by level, creates an ORAM for each level l with capacity for d^l blocks and stores the blocks of a tree level in the respective ORAM. The resulting data structure consists of L ORAMs that keep an identical structure to an input tree-based index. Each tree level is assigned to a single ORAM that stores all the of the level's nodes. Before a tree node is written to an ORAM, its internal structure is updated and a unique access counter is added for every pair of (key, ptr). It's important to note that the pointer ptr in a node at level i points to a node offset a at level $i+1$. As blocks are written to an ORAM for the first time, the cryptographic token used by the Φ.Read function starts with a counter set to 0 and the eviction token increments the counter by a single unit. At the end of the index initialization function every parent node can compute the location of its children.

After the index initialization, the Init function creates an additional level to store the *Table Heap* blocks. The initialization process starts with the allocation of an oblivious table $\tilde{\mathcal{T}}$ (line 4) filled with N dummy blocks. Afterwards, the algorithm scans the *Table Heap* block by block (line 5) and generates two cryptographic tokens for each block (line 6). The initial location of a block is computed by providing $\mathbf{F}$ with a unique message composed by the total number of levels $L+1$, the block address a and an initial counter set to 0 (line 6). The eviction token is computed with a similar message, but the counter is incremented by 1. The syntax of the message ensures that each block's location i is independent from previous locations and every other block. During the table scan, every block a is encrypted and stored in the oblivious data structure at a uniform random location defined by its location tokens δ (line 7-9). The function returns the internal state.

Search algorithm. We now describe CODBS' oblivious search algorithm. During this first look at the algorithm we are not concerned with volume leakage and assume that for every input query τ the protocol returns a single *Table Heap* block. This assumption implies that every indexed record is unique and there are no range queries. We address this limitation in Section IV-A. With this simplification the algorithm cascades from the first ORAM level to the last, selecting a single node at each level. In detail, a query consists of the following steps:

1.) Oblivious tree search. (line 4-13): In this step, the algorithm traverses the L levels of the tree-based index (line 3), fetching a node from each level until it reaches a tree leaf. At every level of the tree scan, the algorithm accesses the tree node at address a stored on an oblivious location defined by the counter ctr. These variables are initially set to the tree root (line 2) and similarly to the initialization algorithm, the current block location tokens are calculated with a PRF (line 5). The accessed tree node (line 8) is processed by the Next function which selects a new child node address a' and a counter ctr'

Algorithm 2: CODBS Search protocol

```
1  Function Search($st$, $\tilde{\mathcal{I}}$, $\tilde{\mathcal{T}}$, $\tau$)
2      $(\mathsf{sk}_F, \mathsf{sk}_E, ctr_r, \mathsf{st}_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}}) \leftarrow st$; $a \leftarrow 0$; $\text{ctr} \leftarrow ctr_r$
3      for $l \in \{0, 1, \ldots, L\}$ do
4          $\mathsf{st}_{\tilde{\mathcal{I}}_l} \leftarrow \mathsf{st}_{\tilde{\mathcal{I}}}[l]$; $\tilde{\mathcal{I}}_l \leftarrow \tilde{\mathcal{I}}[l]$
5          $\delta \leftarrow (\mathsf{F}(\mathsf{sk}_F, l||a||ctr), \mathsf{F}(\mathsf{sk}_F, l||a||ctr + 1))$
6          $(\mathsf{st}'_{\tilde{\mathcal{I}}_l}, \text{c}) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, \text{a})$
7          $data \leftarrow \Theta.\mathsf{Dec}(\mathsf{sk}_E, c)$
8          $(data', a', ctr') \leftarrow \mathsf{Next}(data, \tau)$
9          $c' \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, data')$
10         $(\mathsf{st}''_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}'_l) \leftarrow \Phi.\mathsf{Write}((\mathsf{st}'_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a, \text{c'})$
11         $\text{a} \leftarrow a'$; $ctr \leftarrow ctr'$; $\mathsf{st}_{\tilde{\mathcal{I}}}[l] \leftarrow \mathsf{st}''_{\tilde{\mathcal{I}}_l}$; $\tilde{\mathcal{I}}[l] \leftarrow \tilde{\mathcal{I}}'_l$
12     $\delta \leftarrow (\mathsf{F}(\mathsf{sk}_F, L + 1||a||ctr),$
         $\mathsf{F}(\mathsf{sk}_F, L + 1||a||ctr + 1))$
13     $(\mathsf{st}'_{\tilde{\mathcal{T}}}, c) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a)$
14     $data \leftarrow \Theta.\mathsf{Dec}(\mathsf{sk}_E, c)$
15     $c' \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, data)$
16     $(\mathsf{st}''_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}') \leftarrow \Phi.\mathsf{Write}((\mathsf{st}'_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a, \text{c'})$
17     $st' \leftarrow (\mathsf{sk}_F, \mathsf{sk}_E, ctr_r{+}1, \mathsf{st}''_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}})$
18     return $(st', \tilde{\mathcal{I}}, \tilde{\mathcal{T}}', data)$
1  Function Next ($data$, $\tau$)
2      for $i \in \{0, 1, \ldots, |data|\}$ do
3          $(key, a, c) \leftarrow data[i]$
4          if SelectChild$(key, \tau)$ then
5              data[i] $\leftarrow (key, a, c + 1)$
6              return $(data, a, c)$
```

to be accessed in the next tree level.

2.) Node selection. (Function Next): This operation selects a single tree child node address from an input parent node. A tree node is a set of tuples (key, a', c') where each tuple consists of a node address a', a location counter c' and a predicate key. We scan over every tuple (line 3) and check if a tuple key matches an input query τ (line 4). As the choice of which child nodes satisfies an input query depends on the underlying index we abstract this process with the function SelectChild that takes as input a query τ and the current child key. This function returns a boolean result bit b that is set to true if the key satisfies the query. When a child node is found the function increments the counter of the target child node (line 6). This update is made ahead of time, before the child node is accessed, to ensure that the parent node keeps a consistent pointer before it's shuffled back to the ORAM external storage. The function ends by returning the updated accessed node as well as the current location of the next node, defined by the address a and the old counter value c.

3.) Table heap access. (Line 14-19): Finally, after scanning the search tree and reaching a leaf node, the algorithm obtains a single *Table Heap* block pointer. With this information, the current location of the block is calculated with a PRF function (line 6) and the block is accessed and evicted (line 13-14). At the end of the algorithm, the *Trusted Proxy* internal state is updated (line 15) and the resulting block returned to the client.

Multi-User setting. Our protocol mainly considers a single-user setting, but can be extended to the multi-user setting. We can follow an approach similar to POSUP [10] and store an access control list (ACL) on the *Trusted Proxy*. This meta-data is created by the data owner and outsourced to the remote server during the database initialization process. Given an input query, the *Trusted Proxy* authenticates the request using the user credentials and validates the user's permissions. If the authentication is successful, then the *Trusted Proxy* searches the database obliviously, as defined in Algorithm 2.

A. Oblivious Query Stream

Until this point, the Search algorithm was a 1-to-1 function that returned a single table block for every input query. However, the database must support range queries and equality queries that return multiple results. We address this limitation with the insight that any query with multiple resulting records can be unfolded into a sequence of multiple queries with a single result. Additionally, queries can be composed one after the other to obtain an oblivious stream of client requests and database results. Next, we provide a concise description of our solution assuming that the search keys are defined in a continuous domain fully known to the client. This assumption matches existing work in state-of-the-art oblivious data structures [8] and can easily be dropped at the cost of additional server-side bookkeeping.

With this observation, the CODBS client is implemented as an algorithm that maintains a constant rate r of requests/responses with the *Trusted Proxy*. The algorithm starts by opening an authenticated channel with the *Trusted Proxy* and proceeds to send queries on a loop at a rate r. The first query starts by searching for the first element in a subrange of the search key domain. The request is processed by *Trusted Proxy* which scans every ORAM level with the Search algorithm. The resulting database block is stashed by the client which keeps sending queries with the consecutive elements in the key domain. A search query ends when the *Trusted Proxy* returns a dummy element that does not satisfy the client query. This query stream is crucial to hide one of the main sources of leakage of search queries over ORAMs, the volume leakage. To address this leakage the query stream remains active by the client, even if there are no new queries to search. In this case, a dummy query is sent to the *Trusted Proxy* and its result is ignored by the client. With this approach, the volume leakage of the system no longer depends on the size of the result set of a query but rather on the rate of requests made to the proxy. This rate is public information that can be adjusted depending on the workload. Regardless of the request rate, the access patterns no longer depend on private data.

B. Forest ORAM

We now instantiate the underlying ORAM construction used for each level. A major concern that arises from storing the *Table Heap* in an ORAM is the bandwidth blowup — number of blocks transferred per access — of an oblivious access.

Even though *Table Index* size is proportional to the number of blocks in a *Table Heap*, the cost of a database search is dominated by the access to the last level. In our experimental evaluation we verified that there are 50 times more blocks in a *Table Heap* than in a *Table Index* for a small dataset and this difference only increases as the dataset grows. To address this issue, we propose Forest ORAM a simple optimization to Path ORAM that scales with the number of blocks.

Forest ORAM leverage OblivStore's partition framework [23] to lower the bandwidth blowup of Path ORAM. Conceptually, the OblivStore framework splits a single ORAM into multiple independent partitions. Each partition stores only a subset of data blocks and a single oblivious access fetches blocks from one partition. After an access, blocks are shuffled to a new partition to hide the access patterns. Each partition also has an individual client side stash that temporarily stores blocks until they are evicted in background process. The classic OblivStore construction instantiates each partition as an hierarchical ORAM with $\mathcal{O}(\log(N))$ bandwidth blowup. In Forest ORAM, we swap the hierarchical ORAMS with Path ORAM constructions. As such, instead of storing N blocks in a single Path ORAM, blocks are distributed among P partitions with height $L = \log(N) - \log(P)$ which results in a bandwidth blowup of $\mathcal{O}(\log(N) - \log(P))$ and a $\mathcal{O}(\log(P))$ upper bounded stash. We provide a detailed description of the algorithm in the extended version of our paper [24].

C. Security Analysis

CODBS is secure as stated in Theorem 1 and the proof is presented in our extended version of this paper [24].

Theorem 1. The CODBS construction defined by Algorithm 1 and Algorithm 2 is a secure Oblivious Index Scan according to Definition 1 if Φ is an Oblivious RAM scheme, Θ is an IND-CPA symmetric encryption scheme and F is a PRF.

Proof sketch. The security analysis of CODBS hinges on the composition of multiple black-box ORAMs. Intuitively, every query processed by the *Trusted Proxy* consists of a sequence of L requests to independent ORAMs. From the adversary perspective, the accesses to each ORAM generates an arbitrary sequence of storage accesses to encrypted data blocks. Furthermore, the sequence of requests to each ORAM is deterministic and independent from the input query. As such, the adversary observes a sequence of arbitrary accesses to the external storage and does not learn any additional information.

We prove our security intuition with five indistinguishably games, from a real world to an ideal world. The first two hops consist of syntactic changes to the real-world, adding ideal structures that simplify the next hops in the proof. In the third hop we require position-based ORAMs accesses to be indistinguishable from random accesses, by replacing the PRFs used to generate location tokens by a true random function. The fourth hop captures the intuition that our scheme is secure as long as blocks are encrypted with IND-CPA schemes. Finally, the last game shows that the ORAM accesses to every level are independent of the input query by replacing the input address of an ORAM level with a dummy address. Overall, the security games prove that an adversary cannot distinguish between the real world and ideal world by using the security definition of PRFs, IND-CPA and ORAM.

V. Experimental Evaluation

We implemented CODBS as a PostgreSQL server-side extension that supports equality and range queries. We build upon on a widely used open-source database management systems to ensure that our system design choices are based on realistic assumptions and the evaluation results are comparable to industry standard databases. The complete solution has roughly 12K lines of C code and is composed by an ORAM library, a *Trusted Proxy* engine and a database wrapper.

A. System Implementation

Our system currently supports two ORAM constructions: Path ORAM and Forest ORAM. We implemented both constructions in a general-purpose ORAM library, open-source for any application that needs to hide its access patterns [25].

We implemented CODBS as the database component that replaces the *Trusted Proxy* and provides an input API similar to the definition in Section III-A. Additionally, this component has an output API to access the external database storage. The component is deployed within an Intel SGX enclave collocated with the database. Currently, the extension supports a B^{+}-tree as the index data structure. We leverage the LibSodium [26] library v1.0.5 to instantiate the cryptographic primitives as it provides constant-time implementations. Concretely, we instantiate the PRF F as a SHA256-HMAC and Θ encryption scheme as an AES block cipher with CBC mode. The *Trusted Proxy* is connected to the database with a wrapper component implemented as Foreign Data Wrapper (FDW), a PostgreSQL module that enables developers to extend the database server without modifying the core source code.

B. Methodology

We measure the performance of our system to answer the following questions: *1)* How does CODBS scale with increasingly larger datasets; *2)* What is the overhead in comparison to a plaintext database for different types of queries; *3)* How does size of the result set of a range query impact the overall system the database performance. In the evaluation we compare our construction to a system `Baseline` which consists of database that stores the *Table Index* and *Table Heap* in a single Path ORAM construction. For a fair comparison, the `Baseline` also uses an oblivious query stream.

Micro & Macro Settings. We divided our system evaluation in two distinct settings, a micro setting and a macro setting. Both settings use a synthetic dataset and workload. The micro setting measures the performance of Forest ORAM construction and Path ORAM constructions isolated from the CODBS scheme and the PostgreSQL engine. In this setting each construction read/writes blocks of $B = 8$ KiB from/to the main memory at randomly sampled positions. The data blocks written to memory are also sampled from a uniform

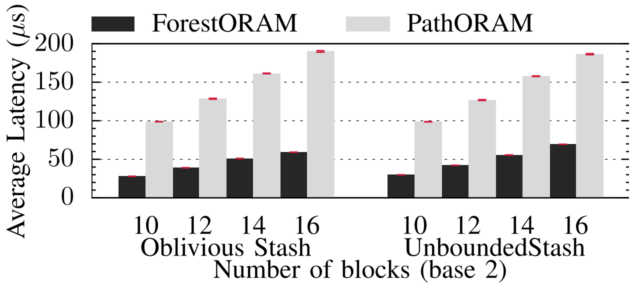


Fig. 3: Forest ORAM and Path ORAM comparison. X-axis measures number of blocks and errors bars the 95% CI.

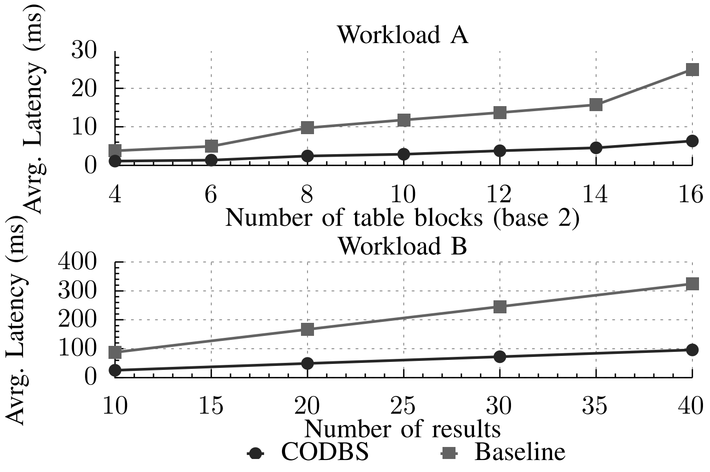


Fig. 4: Avg. latency of YCSB workloads. Workload A X-axis measures the numbers of blocks. Workload B X-axis measures a query resulting records.

distribution $\{0,1\}^B$. In the macro setting we use the YCSB benchmark v0.18 [14]. In the benchmark the database has a single table with two columns. The first column Key is indexed and stores unique keywords. The second column stores JSON objects containing randomly sampled data. Each table record has the same size as a database block 8 KiB. We configured the benchmark to generate two workloads over the indexed column: **Workload A)** Equality queries that search for keywords sampled from a uniform random distribution; **Workload B)** Range queries that start on a randomly sampled keyword and search for at most k values where k is uniformly sampled from $[1..X]$. The first benchmark is designed with a one-to-one match between a database record and database table block to enable a linear analysis of the expected database performance as the table size increases.

For both benchmarks we performed 5 runs for each combination of deployment, configuration, workload and database size. The number of runs is the maximum necessary to calculate an accurate confidence intervals (CI) [27] with the measured standard deviation. Each run lasted for 40 minutes with a 10-minute warm up period and a 2-minute cool down period between each run. Furthermore, we ensure that each run is an independent observation by clearing all persistent data.

Collected Metrics. In the micro benchmark we measure the mean and the percentile latencies of a read and write operations for every run. With the YCSB benchmark we collect the mean and percentile latencies as well as the system throughput for every run. The samples mean are calculated within an 95% CI with the Student's t-distribution [27]. We collected CPU, memory and disk usage of each system.

Experimental Setup. The system was deployed in a private infrastructure. Each computational node had an Intel Core i3-7100 CPU with a clock rate of 3.90 GHz and 2 physical cores in hyper-threading. The main memory was a 16 GiB DDR3 RAM and the solid-state storage a Samsung PM981 NVME with 250 GB. The machines had Intel SGX SDK v2.0 installed. Nodes were connected by a 10 GiB network switch.

C. *Micro Benchmark*

Figure 3 depicts the results of the micro benchmark. The workload starts with an empty oblivious data structure and measures the latency of an oblivious access request, either a read or a write. The benchmark measures the average latency of a request for Forest ORAM and Path ORAM. It also measures the latency of both constructions with two distinct stashes, an unbounded stash where a stash access stops as soon as it finds an element and a double-oblivious stash with fixed size upper bounded at $\log(N)$. The number of blocks stored on the ORAMs increases from 2^{10}(65 MiB) to 2^{16} (2 GiB).

As can be observed, the performance difference between both stashes is almost non-existent. This is expected as position-based ORAM constructions are designed to utilize as much as possible the stash. In more detail, on an oblivious stash a Forest ORAM request takes on average 27 μs for the smallest data set while in the largest takes 59 μs. The Path ORAM has a higher latency, with 99 μs for an oblivious request in the smallest data set and 190 μs for the largest data set. This difference represents at least a $\sim 2.6\times$ speedup. In the unbounded stash, the most significant difference is noticed on Forest ORAM in the 2^{14} dataset where there is a an average performance decrease of 2.5%. As the dataset increases, the performance of both systems degrades at a similar rate with Forest ORAM latency increasing by $\sim$ 15% and Path ORAM by $\sim$ 17%. At the 90th percentile, both systems performances degrade considerably, with Forest ORAM latency increasing at most by $\sim$ 17% and Path ORAM by 7%. Even with these outliers Forest ORAM latency is at least $\sim 1.6\times$ lower than Path ORAM.

This benchmark shows that the asymptotic difference between Forest ORAM and Path ORAM has a practical impact. The average latency as well as the 90th and 99.9th percentiles are smaller than Path ORAM. This difference is attributed to the partition framework which scales the number of partitions and the tree height of each individual partition as the data set increases. In fact, the number of partitions depends on parameter that can be adjusted to increase even further the performance of Forest ORAM at the cost of additional client-side storage. The only unexpected result is the 99.9th percentile maximum performance degradation of $\sim$ 120% when compared

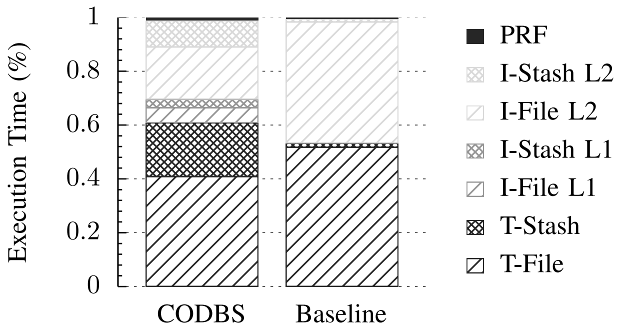


Fig. 5: Breakdown of time spent during query execution.

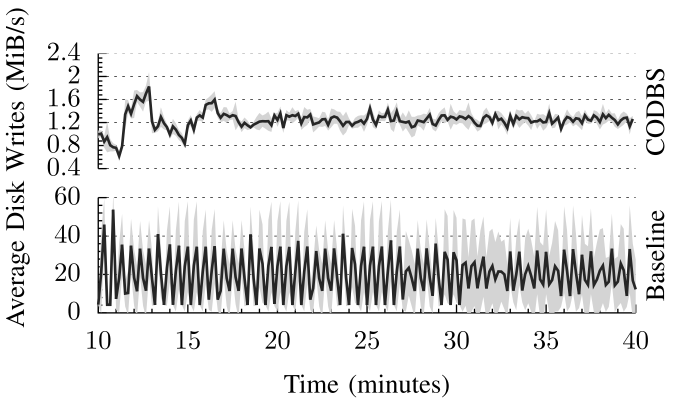


Fig. 6: Avg. disk writes over time on workload A on data set with 2^{16} records. The light gray represents the 95% CI.

to Path ORAM. However, this difference only occurs in the smallest data sizes and stabilizes in both protocols at $\sim 40\%$.

D. Macro Benchmark

We now present the performance of CODBS and compare it to `Baseline`. The `Baseline` solution uses a Path ORAM construction to store the database data and does not divide the *Table Index* in multiple ORAM levels. Instead, the *Table Index* is stored in a single ORAM and accessed as an oblivious data structure similar to the one used in Oblix and proposed by *Wang et. al* [8], [9]. However, it still calculates the block addresses using a PRF to keep both systems comparable. With this approach, the `Baseline` provides clear understanding on the practical performance improvements of our cascade solution. Additionally, we also contrast both solutions to a plaintext PostgreSQL database. The evaluation consists on measuring the throughput and latency of increasingly larger database databases until a saturation point is reached and the systems cannot provide a practical throughput (> 1 op/s).

Figure 4 presents the macro results. In workload A, the database size starts with 2^4 *Table Heap* records and a *Table Index* with a single tree level (a total of 47 MiB) and is increased until 2^{16} *Table Heap* blocks with a *Table Index* of two levels (a total 2.1 GiB). Across this range, CODBS maintains an average latency below 10 ms which corresponds to 886 ops/s for the smallest data set and 158 ops/s for the largest. The average maximum throughput of every run in `Baseline` is 264 ops/s (smallest dataset) and the average latency surpasses CODBS at just 2^8 *Table Heap* records. Its highest average latency is 25 ms, corresponding to a throughput of 40 ops/s, meaning that CODBS has approximately a $4\times$ speedup. There is a slight performance degradation of both systems on the 99th percentile in the largest dataset. CODBS has a 30% latency increase with an average latency of 13.34 ± 1.74 ms and the `Baseline` has an increase of 17% with an average latency of 29.33 ± 12.95 ms. In contrast to both solutions, a plaintext PostgreSQL has on average $\sim$ 7663 ops/s.

Workload B uses the dataset with 2^{16} *Table Heap* records to measure the latency of range scans, more specifically where clauses with a greater than operator. The number of resulting records ranges from 10 to 40, with larger ranges becoming impractical. Similar to workload A, the `Baseline` system has the highest average latency of 324 ms and a throughput of 3 ops/s. CODBS has a $\sim 3\times$ speedup with the lowest throughput of 10 ops/s and an average latency of 96 ms.

Figure 5 provides an analysis of the multiple query processing stages in both systems. It breakdowns the execution between the database data structures, *Table Index* (I-File, I-Stash) and the *Table Heap* (T-File, T-Stash) and PRF computation. Each structure is divided even further by the time spent in the ORAM stash and external access to store (I-File, T-File). The CODBS breakdown also accounts for the time spend at each index level (L2 and L1). The overhead of block encryption is measured within the accesses to the external files. As depicted, CODBS spends most of the time (60%) accessing the *Table Heap*, 8% accessing the first *Table Index* level and 28% accessing the second *Table Index* level and the remaining time calculating the PRFs. In contrast, the `Baseline` spends more time accessing the external file and the system throughput is dominated by the disk IO. This claim is further supported by Figure 6 which presents the average write requests to the external storage grouped in intervals of 10 seconds. As can be observed, the `Baseline` has a sustained rate 10 to 40 MiB writes per second while CODBS is constantly below 2 MiB/s.

E. Discussion

Across every benchmark and workload, CODBS displays an overall performance that exceeds that of the `baseline` system. These speedups are the result of combining the cascade approach with the Forest ORAM construction. This combination results in an asymptotic decrease of $\log(\log(N))$ bandwidth blowup compared to state-of-the-art oblivious data structures as shown in Section V-C. This seemingly small difference has a significant impact.

In the YCSB benchmark on workload A with a tree height of just two levels there is a $4\times$ speedup instead of just a $2.6\times$ speedup as might otherwise be expected from the micro benchmarks. This difference is the result of spending less time accessing the storage and a 95% lower number of disk writes on average than the `baseline`. Regarding workload B the performance gains of CODBS in comparison to xthe `baseline` are less significant. With just a $2\times$ speedup, the main bottleneck in this workload seems to be the size of the data exchanged in the oblivious query stream. Across the different result set size, CODBS has on average a write rate of ~ 1.3 MiB/s and the `baseline` writes at most ~ 347 KiB/s.

VI. Related Work

Position-based ORAMs. Position-based ORAMs were first proposed by *Shi et al.* [28] to improve the lower bounds of classic constructions [3], [29]. The first solutions consisted of a framework that stores data blocks in a binary tree. However, the framework requires an eviction process that flushes the nodes down to their corresponding tree paths and has a bandwidth blowup of $\mathcal{O}(\log^3 N)$. Newer constructions based on this framework have mostly improved the eviction process [30]–[32] to lower the bandwidth blowup. Currently, Path ORAM is the most efficient tree-based solution with a blowup of $\mathcal{O}(2 \cdot \log N)$. A new class of algorithms surpass the ORAM lower bound in the "balls and bins" model by assuming computation on the server side. The computation can be homomorphic encryption schemes that have a significant overhead [33], [34] or, as proposed in Burst ORAM [35] and Circuit ORAM [32], be a simple compression with an XOR operator.

Oblivious Data Structures. *Wang et al.* [8] proposed the first work with general-purpose oblivious data structures. One of the main contributions is a pointer-based technique that removes the need for a recursive position map on Path ORAM. With this optimization, a search on an oblivious search tree went from $\mathcal{O}(\log^2 N)$ blowup to a $\mathcal{O}(\log N)$ blowup. Our work lowers even further the blowup of an oblivious search tree. Furthermore, our approach to store the position map in the oblivious search tree is non-interactive which enables accessed node to be shuffled after every accessed.

Volume leakage As stated by *Grubbs et al.* [16] and shown by novel research [17], [36], [37], hiding the access patterns of a database is not enough. The volume leakage of database queries must also be addressed. *Kallaris et al.* [36] was the first to create a formal model of encrypted database that focus on volume leakage. New attacks have been proposed [16].

VII. Conclusion

Our construction provides an efficient solution for secure database searches on tree-based indexes and heap table accesses with minimal bandwidth blowup, with a detailed theoretical analysis on the system security and experimental results pointing towards practical feasibility. We implemented the proposed construction as well as a novel construction Forest ORAM and measured its performance with industry-standard benchmarks. Comparatively to the state-of-the-art constructions, our solution is $1.2\times$ to $4\times$ faster.

Acknowledgments

This work is financed by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within the project PTDC/CCI-INF/31698/2017.

References

[1] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15.

[2] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in *19th Annual Network and Distributed System Security Symposium, NDSS*.

[3] O. Goldreich, "Towards a theory of software protection and simulation by oblivious rams," in *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*.

[4] E. Stefanov, M. V. Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: An Extremely Simple Oblivious RAM Protocol," *J. ACM*, vol. 65, no. 4, pp. 18:1–18:26, Apr. 2018.

[5] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, California, USA*, 2014.

[6] M. Barbosa, B. Portela, G. Scerri, and B. Warinschi, "Foundations of hardware-based attested computation and application to SGX," in *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*. IEEE, 2016, pp. 245–260.

[7] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Intel Software Guard Extensions (Intel SGX) Support for Dynamic Memory Management Inside an Enclave," in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*.

[8] X. S. Wang, K. Nayak, C. Liu, T.-H. H. Chan, E. Shi, E. Stefanov, and Y. Huang, "Oblivious data structures," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. NY, USA: ACM.

[9] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *2018 IEEE Symposium on Security and Privacy (SP)*, May.

[10] T. Hoang, M. O. Ozmen, Y. Jang, and A. A. Yavuz, "Hardware-supported ORAM in effect: Practical oblivious search and update on very large dataset," *PoPETs*, vol. 2019, no. 1, pp. 172–191, 2019.

[11] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*.

[12] S. Eskandarian and M. Zaharia, "Oblidb: Oblivious query processing for secure databases," *Proc. VLDB Endow.*, 2019.

[13] E. Stefanov, E. Shi, and D. X. Song, "Towards practical oblivious ram." in *NDSS*. The Internet Society, 2012.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*.

[15] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer, "Generalized search trees for database systems," in *Proceedings of the 21th International Conference on Very Large Data Bases*, ser. VLDB '95.

[16] P. Grubbs, M.-S. Lacharite, B. Minaud, and K. G. Paterson, "Pump up the volume: Practical database reconstruction from volume leakage on range queries," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, 2018.

[17] M. Lacharité, B. Minaud, and K. G. Paterson, "Improved reconstruction attacks on encrypted data using range query leakage," in *2018 IEEE Symposium on Security and Privacy (SP)*, May 2018, pp. 297–314.

[18] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "Vc3: Trustworthy data analytics in the cloud using sgx," in *2015 IEEE Symposium on Security and Privacy*.

[19] J. V. Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution," in *26th USENIX Security Symposium*.

[20] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 640–656.

[21] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *25th USENIX Security Symposium*.

[22] M. Bellare and P. Rogaway, "Introduction to modern cryptography," in *UCSD CSE 207 Course Notes*, 2005, p. 207.

[23] E. Stefanov and E. Shi, "Oblivistore: High performance oblivious cloud storage," in *2013 IEEE Symposium on Security and Privacy*.

[24] R. Pontes, B. Portela, M. Barbosa, and R. Vilaça, “CODBS: A cascading oblivious search protocol optimized for real-world relational database indexes,” Cryptology ePrint Archive, Report 2021/917, 2021, https://eprint.iacr.org/2021/917.

[25] R. Pontes, “CODBS ORAM Library,” 2021. [Online]. Available: https://github.com/rogerioacp/oram

[26] F. Denis, “The sodium cryptography library,” Fev 2020. [Online]. Available: https://download.libsodium.org/doc/

[27] R. Jain, *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling.*, ser. Wiley professional computing. Wiley, 1991.

[28] E. Shi, T. H. H. Chan, E. Stefanov, and M. Li, “Oblivious RAM with O((logN)3) Worst-Case Cost,” in *Advances in Cryptology – ASIACRYPT 2011*.

[29] O. Goldreich and R. Ostrovsky, “Software protection and simulation on oblivious rams,” *J. ACM*, vol. 43, no. 3, pp. 431–473, May 1996.

[30] C. Gentry, K. A. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs, “Optimizing oram and using it efficiently for secure computation,” in *Privacy Enhancing Technologies*. Springer Berlin Heidelberg.

[31] K. Chung, Z. Liu, and R. Pass, “Statistically-secure ORAM with õ(log^2 n) overhead,” in *ASIACRYPT (2)*.

[32] X. Wang, H. Chan, and E. Shi, “Circuit oram: On tightness of the goldreich-ostrovsky lower bound,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*.

[33] T. Mayberry, E. Blass, and A. H. Chan, “Efficient private file retrieval by combining ORAM and PIR,” in *NDSS*. The Internet Society, 2014.

[34] I. Abraham, C. W. Fletcher, K. Nayak, B. Pinkas, and L. Ren, “Asymptotically Tight Bounds for Composing ORAM with PIR,” in *Public-Key Cryptography – PKC 2017*, S. Fehr, Ed.

[35] J. Dautrich, E. Stefanov, and E. Shi, “Burst ORAM: Minimizing ORAM response times for bursty access patterns,” in *23rd USENIX Security Symposium (USENIX Security 14)*.

[36] G. Kellaris, G. Kollios, K. Nissim, and A. O’Neill, “Generic attacks on secure outsourced databases,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.

[37] Z. Gui, O. Johnson, and B. Warinschi, “Encrypted databases: New volume attacks against range queries,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*.