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

Rogério Pontes[1,3], Bernardo Portela[1,4], Manuel Barbosa[1,3], and Ricardo Vilaça[2,3]

[1]University of Porto
[2]University of Minho
[3]INESC TEC
[4]NOVA LINCS

July 6, 2021

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.

Contents

1 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 [8, 22].

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 [16]. 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 [32, 33].

The overhead of ORAM schemes can be minimized by using an isolated execution environment (IEE) [4] 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) [27] 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.* [37] 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 [28] uses an oblivious search tree to index the keywords of a database and POSUP [21] 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.* [37]. Besides these previous examples that focused only on SSE systems, there are also fully-fledged oblivious database solutions such as Opaque [39] and ObliDB [14]. 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 [37]. 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

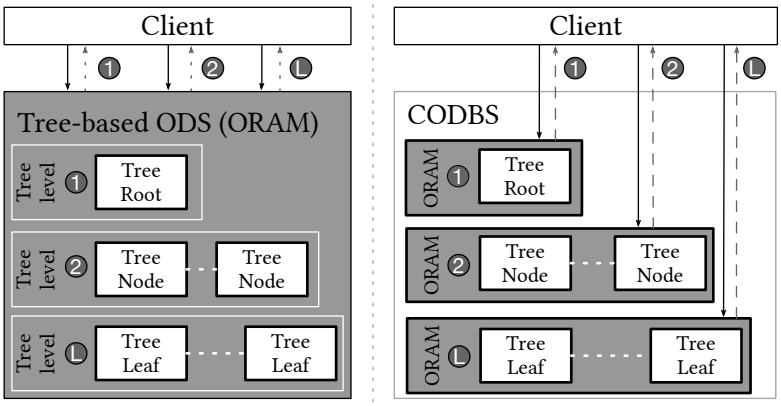


Figure 1: Search tree with L levels stored in two different ORAM constructions: a tree-based ODS as proposed by *Wang et al.* [37] and the CODBS scheme presented in this paper.

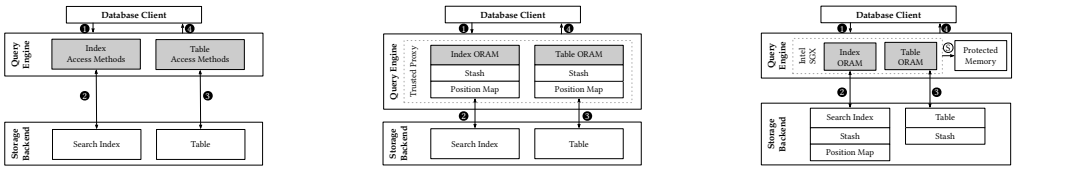
ORAM addresses. Furthermore, our scheme searches over keywords and reduces the position maps to a small constant. Our system is also closely related to Oblix [28] tree-based ODS but our construction has a lower bandwidth blowup. The proposed scheme is only a subcomponent of our new oblivious relational 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 [35].
- 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 [10]. 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 [37, 28].

2 Problem Definition

2.1 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



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

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

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 [20] 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).

2.2 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 [18, 25].

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

2.3 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 [30]. 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 [7, 38], but not the access patterns. Our protocol tackles this issue with constant-time implementations [2, 28]. 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.

2.4 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.

3 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 pseudorandom function (PRF) and a symmetric encryption schemes secure against chosen plaintext attacks (IND-CPA) [5]. 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 $\mathcal{T} = \{(a_1, data_1), \ldots, (a_n, data_n)\}$ 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 $(key, a) \leftarrow \mathcal{I}[l][a]$.

Ideal Storage. We capture the access patterns of a database execution using an idealized storage $\mathcal{M}$ that abstracts an untrusted external storage. This storage is a sequence of K words indexed by a logical address space $[K] = \{1, \ldots, K\}$. Each word is a data block of size B that can be individually accessed with a block-based API defined as follows:

- $\mathsf{Alloc}(N) \rightarrow \mathcal{D}$: reserves a empty data structure $\mathcal{D}$ that consists of a subset of storage words of size $N \in [0, \ldots, K]$ from the storage address pace.
- $\mathsf{Read}(\mathcal{D}, a) \rightarrow block$: returns a block of $\mathcal{D}$ at address a.
- $\mathsf{Write}(\mathcal{D}, a, block) \rightarrow \mathcal{D}$: updates the data structure $\mathcal{D}$ with a new block on the storage address a.
- $\mathsf{Addrs}() \rightarrow \mathcal{X}$: Returns the access patterns trace $\mathcal{X}$ of every API invocation on the ideal storage.

The ideal storage is available to any algorithm at any point of its execution and registers every API request in a public access pattern $\mathcal{X}$. More specifically, an access pattern $\mathcal{X} = \{I_1, \ldots, I_N\}$ is a sequence of instructions $I_i = (op, addr, data)$ where each instruction has an operation defined by the public interface $op \in \{\mathsf{Alloc}, \mathsf{Read}, \mathsf{Write}\}$, a target address within the ideal storage range $addr \in [K]$ and a binary string that is either written to or read from the storage $data \in \{0,1\}^B$. In case of a read instruction the data is initially empty and is filled with the content of the logical address in the storage. In case of an allocation, the requested storage size is specified in the address and the data field is left empty. We denote by $out \leftarrow \mathcal{A}^{\mathcal{M}}(prms)$ the execution of an algorithm $\mathcal{A}$ with access to the ideal storage $\mathcal{M}$ given parameters $prms$. The access pattern of the algorithm execution can be obtained by $\mathcal{X} \leftarrow \mathcal{M}.\mathsf{Addrs}()$.

3.1 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.

Correctness. An oblivious index scan scheme is correct if for every security parameter λ, every plaintext database DB, every pair of oblivious data structures initialized $\tilde{\mathcal{I}}$ and $\tilde{\mathcal{T}}$ initialized by the Init algorithm and every query τ, the set of the data records of a plaintext database search $DB(\tau)$ with size N is equal to the set of records returned after a sequence of N query searches Search with probability $1 - negl(N)$.[1] As such, correctness is defined by:

$$DB(\tau) = \{\mathsf{Search}(st, \tilde{\mathcal{I}}, \tilde{\mathcal{T}}, \tau_0), \ldots, \mathsf{Search}(st, \tilde{\mathcal{I}}, \tilde{\mathcal{T}}, \tau_N)\}$$

Security The security of an OIS construction is defined in the simulation-based real/ideal paradigm. Our security game consists of an adversary that sends a plaintext database of it choosing to the experiment and afterwards sends a sequence of queries. For each query, the adversary receives the access patterns $\mathcal{X}$ of the database to an external storage and a pair of encrypted data structures. In both games, the access pattern includes the addresses of blocks accessed, the instructions (read or write) and the blocks encrypted with a symmetric encryption scheme. We consider an adaptive adversary that can change its attack strategy during the game depending on the access patterns returned by the experiment. Intuitively, an OIS construction is secure if an adversary cannot distinguish if the access patterns were generated by a real-world execution or a simulator that only does arbitrary accesses. In both worlds, the access patterns are captured by an ideal storage $\mathcal{M}$ that is used internally by the ORAM constructions to read/write data blocks. However, in the real-world the ORAM accesses depend on the input queries, while in the ideal world the simulators are only given the databases public parameters, number of blocks, the tree-based index height and fanout. As such, the simulators use the ORAMs as black-boxes to access arbitrary storage addresses. This security definition follows a similar approach to simulation-based definitions of ORAM constructions [29, 3].

Definition 2. Let OIS = (Init, Search) be an oblivious index scan scheme. For every stateful algorithm $\mathcal{A}$ (the adversary) and S (the simulator), consider the following security game:

- $\mathbf{Real}_{\mathcal{A}}^{OIS}(1^\lambda)$: The adversary $\mathcal{A}$ sends the experiment a pair of plaintext database structures $\mathcal{I}$ and $\mathcal{T}$ as well as public parameters: number of *Table Heap* blocks N, *Table Index* height L and fanout d. The experiment initializes a pair of oblivious data structures $(\tilde{\mathcal{T}}, \tilde{\mathcal{I}}) \leftarrow \mathsf{Init}(1^\lambda, \mathcal{I}, \mathcal{T})$ and returns them to the adversary alongside the initialization algorithm access patterns $\mathcal{X}_I$. The adversary follows with a polynomial number of adaptive search queries τ and the experiment outputs updated oblivious structures and the access patterns $\mathcal{X}_S$ of the $\mathsf{Search}(\tilde{\mathcal{I}}, \tilde{\mathcal{T}}, \tau)$ function. In the end, the adversary returns a bit b which becomes the experiment result.

- $\mathbf{Ideal}_{S,\mathcal{A}}^{OIS}(1^\lambda)$: The adversary $\mathcal{A}$ sends to the experiment a pair of plaintext database structures $\mathcal{I}$ and $\mathcal{T}$ and public parameters. The game initializes a pair of dummy oblivious data structures $(\tilde{\mathcal{T}}, \tilde{\mathcal{I}}) \leftarrow \mathsf{S}(1^\lambda, N, L, d)$ and returns them to the client alongside a simulated access pattern $\mathcal{X}_I$. The adversary evaluates a polynomial number of adaptive search queries τ and the experiment returns the updated oblivious structures and simulated access patterns generated by $\mathsf{S}(N, L, d)$. Here the simulator is stateful and the crux is that it does not see the raw data or queries. At the end, the adversary returns a bit b which becomes the experiment result.

[1] This negligible chance of failure matches the probability of failure of a single underlying ORAM scheme. In our proposed construction, the probability is $1 - negl(N) \cdot L$, as L ORAMs are used.

OIS is secure if for all $\mathcal{A}$ there exists a simulator S such that:

$$\left| \Pr\left[\mathbf{Real}_{\mathcal{A}}^{OIS}(1^\lambda) = 1\right] - \Pr\left[\mathbf{Ideal}_{S,\mathcal{A}}^{OIS}(1^\lambda) = 1\right] \right| < \mathsf{negl}(\lambda)$$

The formal security definition is presented in Section 6.1.

3.2 Oblivious RAM

We follow the classical definition of position-based ORAMs [32, 37] 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 3. (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 $\tilde{\mathcal{D}}'$.

Security. In the classical ORAM indistinguishably definition, an ORAM scheme is secure if it generates access patterns independent of the client real accesses. Intuitively, an oblivious access pattern cannot disclose which data the client is accessing, when a data block was last accessed, or if a real access was a read or write operation.

Definition 4. (ORAM security) Let a data request sequence of a client to an external server be denoted by:

$$\vec{y} = ((op_M, a_M, data_M), \ldots, (op_1, a_1, data_1))$$

where M is the sequence size and each op_i denotes a **read**(a_i) or a **write**(a_i, $data_i$) operation where $1 \leq i \leq M$. More specifically, the block read/written is uniquely identified by address a_i, and $data_i$ denotes the data being read/written. Note that request sequences are only sent to the server after the client initializes an ORAM with the Buld algorithm and a **read** or a **write** operation corresponds to the evaluation of the Read algorithm followed by the Write algorithm. Furthermore, the offset 1 corresponds to the most recent operation while the offset M corresponds to the oldest operation. Additionally, Let $\mathcal{X}[\Phi](\vec{y})$ denote the access patterns (possibly randomized) generated by an ORAM construction Φ when accessing a remote storage server. An ORAM scheme Φ is secure if: *1)* for any two data request sequences $\vec{y_1}$ and $\vec{y_2}$ have the same length, their accesses patterns $\mathcal{X}[\Phi](\vec{y_1})$ and $\mathcal{X}[\Phi](\vec{y_2})$ also have the same length and are indistinguishable (computational or statistically); *2)* the data returned on input $\vec{y}$ is correct with probability $>=$ $1 - negl(|\vec{y}|)$.

We present a security definition that captures ORAM construction with an external position map. Intuitively, an ORAM construction is secure according to Definition 5 if the ORAM client generates the position map independently of the input request sequence.

Definition 5. (External position map ORAM security.) Let $\vec{\delta} = (\delta_M, \ldots, \delta_1)$ denote a sequence of position map states such that δ_i determines the access pattern for request op_i. Additionally, we denote by $\mathcal{X}[\Phi; \vec{\delta}](\vec{y})$ the access patterns (possibly randomized) generated by an ORAM construction Φ when accessing a remote storage server given a sequence of position maps $\vec{\delta_y}$.

Given a data request sequence $\vec{y}$, we say that an ORAM construction with an external pmap $\mathcal{X}[\Phi; \vec{\delta}](\vec{y})$ is secure if it satisfies Definition 4 and there exists an algorithm $\mathcal{O}$ that upon activation outputs a pmap with an identical distribution to that produced by the construction. Algorithm $\mathcal{O}$ has access to the database size as a public parameter. Note that $\mathcal{O}$ is not restricted to random algorithms, as an ORAM scheme can be deterministic and simply access the same sequence of addresses (e.g.: full scan of the storage blocks).

4 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 3, 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 [37] 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 5. 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 [32] the tokens are used to compute uniformly random leaves in the server's binary tree.

Algorithm 1: CODBS Init protocol

1 **Function** $\texttt{Init}(1^\lambda, \mathcal{I}, \mathcal{T}, N, L, d)$
2 $\quad \mathsf{sk}_F \leftarrow F.\mathsf{KGen}(1^\lambda); \mathsf{sk}_E \leftarrow \Theta.\mathsf{KGen}(1^\lambda)$
3 $\quad (\mathsf{st}_{\tilde{\mathcal{I}}}, \tilde{\mathcal{I}}) \leftarrow \mathsf{InitSearchTree}(\mathcal{I}, \mathrm{L}, \mathrm{d}, \mathsf{sk}_F, \mathsf{sk}_E)$
4 $\quad (\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow \Phi.\mathsf{Build}(N)$
5 $\quad$ **for** $a \in \{0, \ldots, N\}$ **do**
6 $\quad\quad \delta \leftarrow (F(\mathsf{sk}_F, L+1||a||0), F(\mathsf{sk}_F, L+1||a||1))$
7 $\quad\quad c \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, \mathcal{T}[a])$
8 $\quad\quad (\mathsf{st}_{\tilde{\mathcal{T}}}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a)$
9 $\quad\quad (\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a, c)$
10 $\quad$ **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** $\texttt{InitSearchTree}(\mathcal{I}, L, d, \mathsf{sk}_F, \mathsf{sk}_E)$
12 $\quad \tilde{\mathcal{I}} \leftarrow []; \mathsf{st}_{\tilde{\mathcal{I}}} \leftarrow []$
13 $\quad$ **for** $l \in \{0, 1, \ldots, L\}$ **do**
14 $\quad\quad (\mathsf{st}_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}_l) \leftarrow \Phi.\mathsf{Build}(d^l)$
15 $\quad\quad$ **for** $a \in \{0, 1, \ldots, d^l\}$ **do**
16 $\quad\quad\quad$ data $\leftarrow \mathcal{I}[l][a]$
17 $\quad\quad\quad$ data' $\leftarrow []$
18 $\quad\quad\quad$ **for** $i \in \{0,1, \ldots, |data|\}$ **do**
19 $\quad\quad\quad\quad (key, a') \leftarrow data[i]$
20 $\quad\quad\quad\quad data'[i] \leftarrow (key, a', 1)$
21 $\quad\quad\quad \delta \leftarrow (F(\mathsf{sk}_F, l||a||0), F(\mathsf{sk}_F, l||a||1))$
22 $\quad\quad\quad c \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, data')$
23 $\quad\quad\quad (\mathsf{st}_{\tilde{\mathcal{I}}_l}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a)$
24 $\quad\quad\quad (\mathsf{st}_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}_l) \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a, c)$
25 $\quad\quad \tilde{\mathcal{I}}[l] \leftarrow \tilde{\mathcal{I}}_l; \mathsf{st}_{\tilde{\mathcal{I}}}[l] \leftarrow \mathsf{st}_{\tilde{\mathcal{I}}_l}$
26 $\quad$ **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 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 keeps an identical structure to an input tree-based index. Each tree level is assigned to a single ORAM that stores all the of the levels 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 matches 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 $\Phi.\mathsf{Read}$ function starts with a counter set to 0 and the eviction token increments the counter by a single unit. At the end of the algorithm 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 **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).

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 4.1. 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:

Algorithm 2: CODBS Search protocol

1 **Function** `Search`(st, $\tilde{\mathcal{I}}$, $\tilde{\mathcal{T}}$, τ)
2 $(\mathsf{sk}_F, \mathsf{sk}_E, ctr_r, \mathsf{st}_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}}) \leftarrow st$; a $\leftarrow 0$; 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 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`($data$, τ)
2 **for** $i \in \{0, 1, \ldots, |data|\}$ **do**
3 $(key, a, c) \leftarrow data[i]$
4 **if** SelectChild*(key, τ)* **then**
5 data[i] $\leftarrow (key, a, c+1)$
6 **return** *(data, a, c)*

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' 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 the single-user setting, but can also be extended to the multi-user setting. We can follow an approach similar to POSUP [21] and store an access control list (ACL) on the *Trusted Proxy*, as well as a list of user credentials. 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.

4.1 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 [37] and can easily be dropped at the cost of additional server-side bookkeeping, as is standard in relational databases.

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.

5 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 an extension to Path ORAM that scales with the number of blocks.

Forest ORAM is based on OblivStore [35], a ORAM partition framework that divides a single ORAM construction in multiple, independent P partitions. Each partition is protected with an ORAM scheme and the cost of accessing a single block depends on the number of blocks in a partition and not on the total number of blocks stored on the external storage. However, the initial OblivStore's constructions was proposed before tree-based constructions became standard and thus uses a hierarchical ORAM scheme for each partition, resulting in a worst-case bandwidth blowup of $\mathcal{O}\sqrt{N}$. In Forest ORAM we replace the hierarchical framework with Path ORAM and optimize the number of partitions to lower the bandwidth overhead of accessing a single Path ORAM and prevent a partition from overflowing. Forest ORAM has $\mathcal{O}(\log(N) - \log(P))$ bandwidth blowup and a $\mathcal{O}(\log(P))$ upper bounded stash.

The security of OblivStore, as with most tree-based ORAM algorithms, is based on the assumption that the stash is stored securely by the client. However, if the stash is stored inside an IEE's secure memory, every access to the stash is disclosed to the adversary. Specifically, an adversary can learn the stash offset accessed and track the movement of blocks within the stash. This leakage can result in linkability attacks [35]. This problem also affects OblivStore as it keeps an individual stash for each partition. The issue becomes clear with the following example; after an oblivious access, OblivStore moves an accessed block from a partition A to a partition B. Without any modification to the original construction, the movement between partitions is simply a matter of removing a block stored in partition A and writing it to the partition B stash. However, if the adversary can trace this exchange of blocks, it learns the exact location where the block was stored. This information would have never been disclosed on a classical client-server deployment, which compromises OblivStore's security.

We addresses this issue with a single oblivious stash shared between all of the partitions. This oblivious stash stores blocks from every partition and has a fixed size equal to the upper bound of excepted blocks, i.e., $\mathcal{O}(\log(P))$ blocks. We denote the oblivious stash as `OS` and we use the standard set notation to denote any access to the stash. To ensure a uniform access pattern, all operations to the stash implicitly touch every element and any conditional logic, such as if conditions, or assignments are executed with constant time operators. We also considered that the Forest ORAM algorithms executed inside an IEE are implemented with constant time guarantees to prevent any access patterns from being disclosed trough side-channel attacks. However, we refrain from explicitly presenting low-level detailed algorithms with constant time operators as there are several standard approaches that can be applied [28, 21, 2].

The Forest ORAM construction is defined by the algorithms presented in Algorithm 3. Whereas state-of-the-art position based ORAM algorithms provide a single Access method, we split this method in two main functions Read and Write. Furthermore, we also explicitly define the initialization algorithm Build that allocates an external memory structure and setups the construction parameters. Our algorithms are mostly similar to Path ORAM definition [32] and we explicitly highlight our modifications in blue, which mostly include the division of blocks in multiple partions as done by OblivStore.

Intuition. We first start with an intuitive overview of the protocol. Following OblivStore's partition framework, the untrusted server storage is divided into P individual partitions. The N outsourced blocks are uniformly distributed between the partitions, with each partition storing about N/P blocks each. A partition stores the data in a binary tree, similar to Path ORAM. The nodes in the binary tree are known as buckets and each one stores up to Z data blocks.

Main Invariant. Forest ORAM has one main invariant. Every block is mapped to a uniformly random partition p and a uniformly random leaf l in a partition tree. If a block a is mapped to a partition p and a leaf l, the block can either be found on a client cache or on a bucket at partition p in the path from the root to the leaf l.

To access (read or write request) a block from the server, Forest ORAM uses the external position map to obtain the location of the block. Given a partition p and a leaf l, the entire path from the tree root in partition p to the tree leaf l is transferred to the client stash. The client shuffles the blocks and flushes a new set of blocks to the path accessed. To prevent any number of subsequent accesses to the same block from being disclosed to an adversary, blocks are re-assigned to new locations after each access.

When a block is re-assigned, its partition as well as its tree leaf are sampled from a uniform random distribution. The blocks stay on the client-side stash until a client request can flush it back to the external storage. Similar to OblivStore, we prevent the stash from overflowing with a periodic eviction process that flushes blocks from the stash to the partitions. This process is independent of the client requests and the client cache size.

External position map. The external position keeps track of the blocks locations and is managed by an application using Forest ORAM. Intuitively, this position map could be as simple as an inverted index that maps a block address a to a tuple with the block location (x, y) where x denotes a tree leaf and y a partition. Additionally, the position map is updated after every access to move blocks to new uniform random position. In Forest ORAM we abstract the details of the implementation of the external position map and define it as a tuple of location tokens $\delta = (\delta_t, \delta_{t+1})$. Each location token is a bitstring $\{0,1\}^D$ with size D. Every Forest ORAM request (Read/Write) on address a receives a location token where δ_t is the current location of a and δ_{t_1} is a new location of a after an access. We denote by $\tau : \delta \rightarrow (x, y)$ a function that takes as input a location token and returns its associated coordinates, leaf x and partition y.

Server Storage. The server storage is divided into P partitions of Path ORAMs such that $P = \log(N)$. Each partition contains $2^{\log(N/P)+1} - 1$ blocks of fixed size B. The partitions are independent Path ORAM constructions with a binary tree of height $L = \{0, 1, \ldots, 2^{\log(N/P)}\}$ and each tree node is a bucket containing Z data blocks. The blocks are structured as a tuple (δ, a, $isDummy$, $data$) such that each block contains its $data$, a bit $isDummy$ that defines if the $data$ is real or free with dummy data, the real block offset a and its current location token δ. It's important to note that we assume that blocks are implicitly encrypted before being stored on the external storage to hide its information from an adversary.

Path. Consider $x \in \{0, 1, \ldots, 2^{L-1}\}$ a leaf node in a tree on partition $y \in \{0, 1, \ldots, P\}$. Each leaf node x in y defines a unique tree path that starts on the tree root and ends on the leaf. We denote by $\mathcal{P}(x, y)$ the set of buckets in partition y on the unique path defined by x. We further denote by $\mathcal{P}(x, y, l)$ a bucket at level l in $\mathcal{P}(x, y)$. Finally, we denote by $\mathcal{L}(y, l)$ the set of all the buckets in partition y at tree level l.

Client Storage. The client's storage consists single oblivious stash `OS` shared between every partition. The stash is a temporary holding place for blocks accessed on a partition that have not been evicted to the server. A subset of the stashed blocks is flushed after a partition access, while the remaining blocks are evicted by a background eviction process. As proven in OblivStore [34] the size of the stash is upper-bounded by the number of partitions $\mathcal{O}(\log N)$.

Initialization. Forest ORAM is initialized by the Build function defined in Algorithm 3. This function allocates an external memory structure $\tilde{\mathcal{D}}$ for P partitions, each with a Path ORAM tree with height L and Z blocks. The allocated blocks are overwritten by dummy data blocks. The algorithm returns the client state with the internal parameters and an empty stash OS.

Algorithm 3: Forest ORAM

```
1  Function Build(N)
2      P ← log(N) L ← log(N/P)
3      D̃ ← M.Alloc(P · 2^L · Z)
4      Bucket ← []
5      for b ∈ {0, 1, ..., Z} do
6          Bucket[b] ← ({0}*, {0}*, 1, {0}*)
7      for i ∈ {0, 1, ... P · 2^L} do
8          M.Write(D̃, i, Bucket)
9      return (([], P, L), D̃)

1  Function Read(st, D̃, a)
2      ((OS, P, L), δ) ← st; (δ_t, _) ← δ
3      (x, y) ← τ(δ_t)
4      for l ∈ {0, 1, ..., L} do
5          OS ← S ∪ M.Read(D̃, P(x, y, l))
6      data ← Read block a from S
7      return ((OS,P,L), data)

1  Function Write(st, D̃, a, data*)
2      ((OS, P, L), δ) ← st; (δ_t, δ_{t+1}) ← δ
3      (x, y) ← τ(δ_t)
4      (x_{t+1}, y_{t+1}) ← τ(δ_{t+1})
5      OS ← (OS − {(δ, a, 0, data)}) ∪ {(δ_{t+1}, a, 0, data*)}
6      for l ∈ {L, L − 1, ..., 0} do
7          S' ← {(δ', a', 0, data) ∈ OS :
             P(x, y, l) = P((τ(δ'), l)}
8          S' ← Select min(|S'|, Z) blocks from S'.
9          OS ← OS − S'
10         D̃' ← M.Write(D̃, P(x,y,l), S')
11     return ((OS,P,L), D̃')
```

In Forest ORAM, a block access to an address a is done with a Read request followed by a Write request. First, the client downloads a block from the server with the function Read, updates the block in case of a write operation and flushes the updated block back to the server with the Write function. The functions are described by the following two steps:

1.) Block Access (Function Read): The function generates the location of offset a from the location token (line 3), reads from the remote server the blocks in path $\mathcal{P}(x, y, l)$ defined by the leaf x, partition y and tree level l (line 4-6). The blocks are stored on the stash and the requested block is returned to the client (line 7-8).

2.) Block Flush (Function Write): The function generates the current location of offset a as well as its new position from the location's tokens (line 3-4). The stash is updated with new block $data^*$ and the new location of address δ_{t+1} (line 5). The block eviction process (line 6-11) scans a tree path level by level, from the leaf to the root. At each level, the function selects blocks from the stash with a path $\mathcal{P}(\tau(\delta'), l)$ that intercepts the path $\mathcal{P}(x, y, l)$ accessed on the Read function. The selected blocks are trimmed to limit the tree node's capacity to Z (line 8). Finally, the resulting blocks are removed from the stash (line 9) and evicted to the external storage (line 9).

5.1 Background Eviction

A core component of OblivStore's construction is the background eviction process that prevents the client-side cache from overflowing. However, the algorithm used by OblivStore's is not applicable to Forest ORAM as it is bounded to hierarchical ORAMs. Furthermore, simply evicting blocks from the stash after an oblivious access can disclose the new location of block. We present a new eviction algorithm that addresses the two main challenges of a background eviction:

Minimizing stash size. The background process runs at an eviction rate $c \cdot v$, such that $c > 0$ is the eviction rate and v is the clients request rate, meaning that for every oblivious access there are c background evictions executed. The eviction process maximizes the number of real blocks in a partition and attempts to replace every dummy block with a real stashed block.

Algorithm 4: Background eviction

1 **Function** `Evict`(st, $\tilde{D}$, a)
2 $(\texttt{OS}, \text{P}, \text{L}) \leftarrow \text{st}$
3 $y \leftarrow \mathsf{Random}(1, \ldots, P)$
4 **for** $l \in \{0, 1, \ldots, L\}$ **do**
5 $\texttt{OS} \leftarrow S \cup \mathcal{M}.\textbf{Read}(\mathcal{L}(y, l))$
6 **for** $x \in \{0, \ldots, 2^{(l+1)} - 1\}$ **do**
7 $S' \leftarrow \{(\delta, a, 0, data) \in \texttt{OS} : \mathcal{P}(x, y, l) = \mathcal{P}(\tau(\delta), l)\}$
8 $S' \leftarrow$ Select $\min(|S'|,\ Z * 2^{(l)} - 1)$ blocks from S'.
9 $\texttt{OS} \leftarrow \texttt{OS} - S'$
10 $\mathcal{M}.\textbf{WriteBucket}(\mathcal{L}(y, l), S')$

Oblivious access pattern. The access patterns of the partitions selected for eviction have to be independent of the data access patterns of client requests and from the stashed blocks. Furthermore, a partition must always be evicted if chosen, even when there are no stashed blocks to evict.

The eviction background process is presented in Algorithm 4. The protocol resembles the Path ORAM eviction function but has a few differences to ensure that blocks are written to partitions independently of the data access patterns. The algorithm transverses the binary tree level by level, from the root to the leaf, reading every block and flushing stashed blocks to the server. Conceptually, Forest ORAM Write eviction is a vertical operation on a tree branch while the background eviction is an horizontal process. We slightly abuse the notation of $\mathcal{M}$.Read and $\mathcal{M}$.Write to denote that every node in tree level is read/write from/to the server.

The algorithm is described by the following 4 steps: **1.) (line 3)** choose a random partition y to evict; **2.) (line 4-5)** Start to iterate the tree level by level $\mathcal{L}(y, l)$, and at every level read the nodes from the server on to the stash; **3.) (line 6-10)** Choose from the stash the set of blocks that can be stored on the current level. This selection filters from the stash the blocks with a path that belong to the selected partition and can be written to the current level; **4.) (line 1)** Evict the batch of selected blocks S' to the current tree level. At each tree level l, blocks are written with a deterministic access pattern from the first node to the last $\{0, 1, \ldots, 2^l\}$.

5.2 Asymptotic Analysis

The Forest ORAM bandwidth blowup per access is similar to Path ORAM, requiring $2 \cdot Z \log(N)$ blocks to be transferred from the server to the client. The major difference is the partition of the tree in multiple subtrees, each with only $\log(N/P)$ blocks leading to a total bandwidth blowup of $2 \cdot Z(\log N - \log P)$.

In a CODBS search, we can further decrease the online bandwidth by pushing some I/O to a background process. The CODBS search algorithm, as defined in algorithm 2, accesses the levels of a *Table Index* with a sequence of Read and Write functions, in this same order. However, note that each level is an independent ORAM and that the levels grow by a factor of 2 from the roots to the leaf. As such, an oblivious access to level i has a lower bandwidth blowup than an access to level $i+1$. Furthermore, consider an execution model where a main thread processes the CODBS search and an additional background thread can evaluate any function. If we evaluate the function Write (Algorithm 4) in the background thread, the main thread can fetch a block from level i and move on to access the next level $i+1$. The eviciton of the block in level i is done by the background thread which flushes the blocks before the main thread fetches all blocks from level $i+1$. By induction, we can apply this process to every level and the CODBS search only processes Read functions, effectively decreasing the only bandwidth blowup to $Z \log(\log N - \log P)$.

$\mathbf{Real}_{\mathcal{A}}^{OIS}(1^\lambda)$	$\mathbf{Ideal}_{S,\mathcal{A}}^{OIS}(1^\lambda)$
$(\mathcal{I}, \mathcal{T}, N, L, d, st_{\mathcal{A}}) \leftarrow \mathcal{A}_1(1^\lambda)$	$(\mathcal{I}, \mathcal{T}, N, L, d, st_{\mathcal{A}}) \leftarrow \mathcal{A}_1(1^\lambda)$
$(st, \tilde{\mathcal{I}}, \tilde{\mathcal{T}}) \leftarrow Init^{\mathcal{M}}(1^\lambda, \mathcal{I}, \mathcal{T}, N, L, d)$	$(\tilde{\mathcal{I}}, \tilde{\mathcal{T}}) \leftarrow Sim_{Init}^{\mathcal{M}}(1^\lambda, N, L, d)$
$\mathcal{X} \leftarrow \mathcal{M}.Addrs()$	$\mathcal{X} \leftarrow \mathcal{M}.Addrs()$
$(\tau, st_{\mathcal{A}}) \leftarrow \mathcal{A}_2(st_{\mathcal{A}}, \tilde{\mathcal{I}}, \tilde{\mathcal{T}}, \mathcal{X})$	$(\tau, st_{\mathcal{A}}) \leftarrow \mathcal{A}_2(st_{\mathcal{A}}, \tilde{\mathcal{I}}, \tilde{\mathcal{T}}, \mathcal{X})$
while $\tau \neq \perp$	**while** $\tau \neq \perp$
$(st, \tilde{\mathcal{I}}, \tilde{\mathcal{T}}, d) \leftarrow Search^{M}(st, \tilde{\mathcal{I}}, \tilde{\mathcal{T}}, \tau)$	$(\tilde{\mathcal{I}}, \tilde{\mathcal{T}}, d) \leftarrow Sim_{Search}^{\mathcal{M}}(\tilde{\mathcal{I}}, \tilde{\mathcal{T}})$
$\mathcal{X} \leftarrow \mathcal{M}.Addrs()$	$\mathcal{X} \leftarrow \mathcal{M}.Addrs()$
$(\tau, st_{\mathcal{A}}) \leftarrow \mathcal{A}_3(st_{\mathcal{A}}, \tilde{\mathcal{I}}, \tilde{\mathcal{T}}, \mathcal{X})$	$(\tau, st_{\mathcal{A}}) \leftarrow \mathcal{A}_3(st_{\mathcal{A}}, \tilde{\mathcal{I}}, \tilde{\mathcal{T}}, \mathcal{X})$

Figure 3: OIS Real and Ideal game definition

6 Security Analysis

Theorem 1. The CODBS construction defined in 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 four 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.

6.1 Security Model

We now present our game-based security model and the security proof of the CODBS protocol. We present in Figure 3 the game-based security definition of an oblivious index (Definition 2). During the execution of the game, an ideal storage $\mathcal{M}$ is available to ORAM primitives in both worlds. In the real game $\mathbf{Real}_{\mathcal{A}}^{OIS}(1^\lambda)$ the adversary $\mathcal{A}$ starts by interacting with the CODBS protocol by providing a plaintext database to be encrypted and stored on oblivious data structures. The adversary is given the complete sequence of accesses to the external storage $\mathcal{M}$ after the initialization algorithm and the pair of resulting data structures. The adversary follows by adaptively requesting new search queries and for each one receives the respective access patterns. In the ideal game $\mathbf{Ideal}_{S,\mathcal{A}}^{OIS}(1^\lambda)$, the interaction between the adversary and the simulator S follows the same flow of interactions. However, the sequence of access patterns and oblivious data structures generated by the simulator does not take into account the adversary plaintext database

Init($\mathcal{I}$, $\mathcal{T}$, N, L, d)

$\mathsf{sk}_F \leftarrow F.Gen(1^\lambda); \mathsf{sk}_E \leftarrow \Theta.Gen(1^\lambda)$
$\tilde{\mathcal{I}} \leftarrow []; \mathsf{st}_{\tilde{\mathcal{I}}} \leftarrow []$
for $l \in \{0, 1, \ldots, L\}$ **do**
  $(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}_l) \leftarrow \Phi.\mathsf{Build}(d^l)$
  for $a \in \{0, 1, \ldots, d^l\}$ **do**
    $\delta \leftarrow (\mathsf{F}(sk_F, l||a||0), \mathsf{F}(sk_F, l||a||1))$
    $d' \leftarrow \mathsf{InitNode}(\mathcal{I}[l][a])$
    $c_d \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, d')$
    $(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a)$
    $o \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a, c_d)$
    $(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}_l) \leftarrow o$
  $\tilde{\mathcal{I}}[l] \leftarrow \tilde{\mathcal{I}}_l; \mathsf{st}_{\tilde{\mathcal{I}}}[l] \leftarrow \mathsf{st}_{\tilde{\mathcal{I}}_l}$
$(\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow \Phi.\mathsf{Build}(N)$
for $a \in \{0, \ldots, N\}$ **do**
  $\delta_c \leftarrow \mathsf{F}(sk_F, L+1||a||0)$
  $\delta_n \leftarrow \mathsf{F}(sk_F, L+1||a||1)$
  $\delta \leftarrow (\delta_c, \delta_n); c_d \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, \mathcal{T}[a])$
  $(\mathsf{st}_{\tilde{\mathcal{T}}}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a)$
  $(\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a, c_d)$
return $((\mathsf{sk}_F, \mathsf{sk}_E, 1, \mathsf{st}_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}}), \tilde{\mathcal{I}}, \tilde{\mathcal{T}})$

Search(st, $\tilde{\mathcal{I}}$, $\tilde{\mathcal{T}}$, τ)

$(\mathsf{sk}_F, \mathsf{sk}_E, ctr_r, \mathsf{st}_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}}) \leftarrow st$
$a \leftarrow 0; c \leftarrow ctr_r$
for $l \in \{0, 1, \ldots, L\}$ **do**
  $\mathsf{st}_{\tilde{\mathcal{I}}_l} \leftarrow \mathsf{st}_{\tilde{\mathcal{I}}}[l]; \tilde{\mathcal{I}}_l \leftarrow \tilde{\mathcal{I}}[l]$
  $\delta \leftarrow (\mathsf{F}(\mathsf{sk}_F, l||a||c), \mathsf{F}(sk_F, l||a||c+1))$
  $(\mathsf{st}'_{\tilde{\mathcal{I}}_l}, data_c) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a)$
  $data \leftarrow \Theta.\mathsf{Dec}(\mathsf{sk}_E, data_c)$
  $o_R \leftarrow \mathsf{Next}(data, \tau); (data', a', c') \leftarrow o_R$
  $data'_c \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, data')$
  $o_W \leftarrow \Phi.\mathsf{Write}((\mathsf{st}'_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a, data'_c)$
  $(\mathsf{st}''_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}'_l) \leftarrow o_W; a \leftarrow a'; c \leftarrow c'$
  $\mathsf{st}_{\tilde{\mathcal{I}}}[l] \leftarrow \mathsf{st}''_{\tilde{\mathcal{I}}_l}; \tilde{\mathcal{I}}[l] \leftarrow \tilde{\mathcal{I}}'_l$
$\delta_c \leftarrow \mathsf{F}(\mathsf{sk}_F, L+1||a||c)$
$\delta_n \leftarrow \mathsf{F}(\mathsf{sk}_F, L+1||a||c+1)$
$\delta \leftarrow (\delta_c, \delta_n)$
$(\mathsf{st}'_{\tilde{\mathcal{T}}}, data_c) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a)$
$data \leftarrow \Theta.\mathsf{Dec}(\mathsf{sk}_E, data_c)$
$data'_c \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, data)$
$(\mathsf{st}''_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}') \leftarrow \Phi.\mathsf{Write}((\mathsf{st}'_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a, data'_c)$
return $((\mathsf{sk}_F, \mathsf{sk}_E, c+1, \mathsf{st}''_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}}), \tilde{\mathcal{I}}, \tilde{\mathcal{T}}', data)$

Figure 4: Extended Real Game

$\mathbf{Sim}_{Init}(N, L, d)$

$\mathsf{sk}_E \leftarrow \Theta.Gen(1^\lambda) : g \leftarrow\!\!\$\ Func(D, R); c \leftarrow 0$
$\tilde{\mathcal{I}} \leftarrow []; \mathsf{st}_{\tilde{\mathcal{I}}} \leftarrow []; \tilde{\mathcal{T}}(\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow \Phi.\mathsf{Build}(N)$
for $l \in \{0, 1, \dots L\}$
  $(\mathsf{st}_{\tilde{\mathcal{I}}'}, \tilde{\mathcal{I}}') \leftarrow \Phi.\mathsf{Build}(d^l)$
  for $a \in \{0, 1, \dots d^l\}$
    $\delta \leftarrow (g(c), g(c+1))$
    $(\mathsf{st}_{\tilde{\mathcal{I}}'}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{I}}'}, \delta), \tilde{\mathcal{I}}', \{0\}^*)$
    $data \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, \{0\}^B)$
    $o \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{I}}'}, \delta), \tilde{\mathcal{I}}', \{0\}^*, data)$
    $(\mathsf{st}_{\tilde{\mathcal{I}}'}, \tilde{\mathcal{I}}') \leftarrow o; c \leftarrow c + 2$
  $\tilde{\mathcal{I}}[l] \leftarrow \tilde{\mathcal{I}}'; \mathsf{st}_{\tilde{\mathcal{I}}}[l] \leftarrow \tilde{\mathcal{I}}'$
for $a \in \{0, 1, \dots, N\}$
  $\delta \leftarrow (g(c), g(c+1))$
  $(\mathsf{st}_{\tilde{\mathcal{T}}}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, \{0\}^*)$
  $data \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, \{0\}^B)$
  $(\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, \{0\}^*, data)$
return $(\tilde{\mathcal{I}}, \tilde{\mathcal{T}})$

$\mathbf{Sim}_{Search}(\tilde{\mathcal{I}}, \tilde{\mathcal{T}})$

$(N, L, d, \mathsf{sk}_E, \mathsf{st}_{\tilde{\mathcal{I}}}, \mathsf{st}_{\tilde{\mathcal{T}}}, g, c) \leftarrow \mathsf{st}_{Sim}$
for $l \in \{0, 1, \dots L\}$
  $\mathsf{st}_{\tilde{\mathcal{I}}_l} \leftarrow \mathsf{st}_{\tilde{\mathcal{I}}}[l]; \tilde{\mathcal{I}}_l \leftarrow \tilde{\mathcal{I}}[l]$
  $\delta \leftarrow (g(c), g(c+1))$
  $(\mathsf{st}'_{\tilde{\mathcal{I}}_l}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, \{0\}^*)$
  $data \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, \{0\}^B)$
  $o \leftarrow \Phi.\mathsf{Write}((\mathsf{st}'_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, \{0\}^*, data)$
  $(\mathsf{st}'_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}'_l) \leftarrow o; c \leftarrow c + 2$
  $\mathsf{st}_{\tilde{\mathcal{I}}}[l] \leftarrow \mathsf{st}'_{\tilde{\mathcal{I}}_l}; \quad \tilde{\mathcal{I}}[l] \leftarrow \tilde{\mathcal{I}}'_l$
$\delta \leftarrow (g(c), g(c+1))$
$(\mathsf{st}_{\tilde{\mathcal{T}}}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, \{0\}^*)$
$data \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, \{0\}^B)$
$o \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, \{0\}^*, data)$
$(\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow o$
$\mathsf{st}_{Sim} \leftarrow (N, L, d, \mathsf{sk}_E, \mathsf{st}_{\tilde{\mathcal{I}}}, \mathsf{st}_{\tilde{\mathcal{T}}}, g, c + 2)$
return $(\tilde{\mathcal{I}}, \tilde{\mathcal{T}})$

Figure 5: OIS Ideal Simulators

or its queries. Instead, the access patterns are generated from public parameters.

Our security model focuses on a passive adversary that can observe every interaction between the *Trusted Proxy* and the external database storage. However, the adversary cannot create arbitrary instances of the protocol, or forge requests and data blocks. In fact, our proof can be extended to an active adversary in the IEE model [4] albeit at the cost of a more extensive proof that would detract from the main concern of our construction: the external database access patterns. Furthermore, note that an ORAM algorithm can be made trivially secure against an active adversary by using standard techniques such as message authentication codes (MAC) or authenticated data structures [31]. Nonetheless, we provide a brief description on the existing mechanisms that can be used to make CODBS secure against active attackers. Intuitively, our construction can be loaded in an IEE with an attested key exchange protocol that ensures that only a single instance of the protocol can establish a secret key between the database client and the *Trusted Proxy*. Furthermore, a secure communication channel between the client and the database prevents the adversary from providing valid inputs to either party. Finally, an authenticated encryption scheme and a sequence of numbers can be used to ensure that the database blocks in the external storage cannot be forged.

6.2 Game-Based Proof

The security proof of CODBS is described through a sequence of four games presented from Figure 6 to Figure 9. We denote by G_i the i-th game and by $Pr[G_i = 1]$ the probability that Game i outputs 1. Each game is a transition from the real game defined in Figure 4 with a slight modifications until the last game that is identical to the ideal game with the simulators defined in Figure 5.

Let $\Phi = (Build, Read, Write)$ be a constant-time position based ORAM with the security guarantees in Definition 3.2. Furthermore, let $\mathbf{F}$ be a PRF with domain D and output Range R

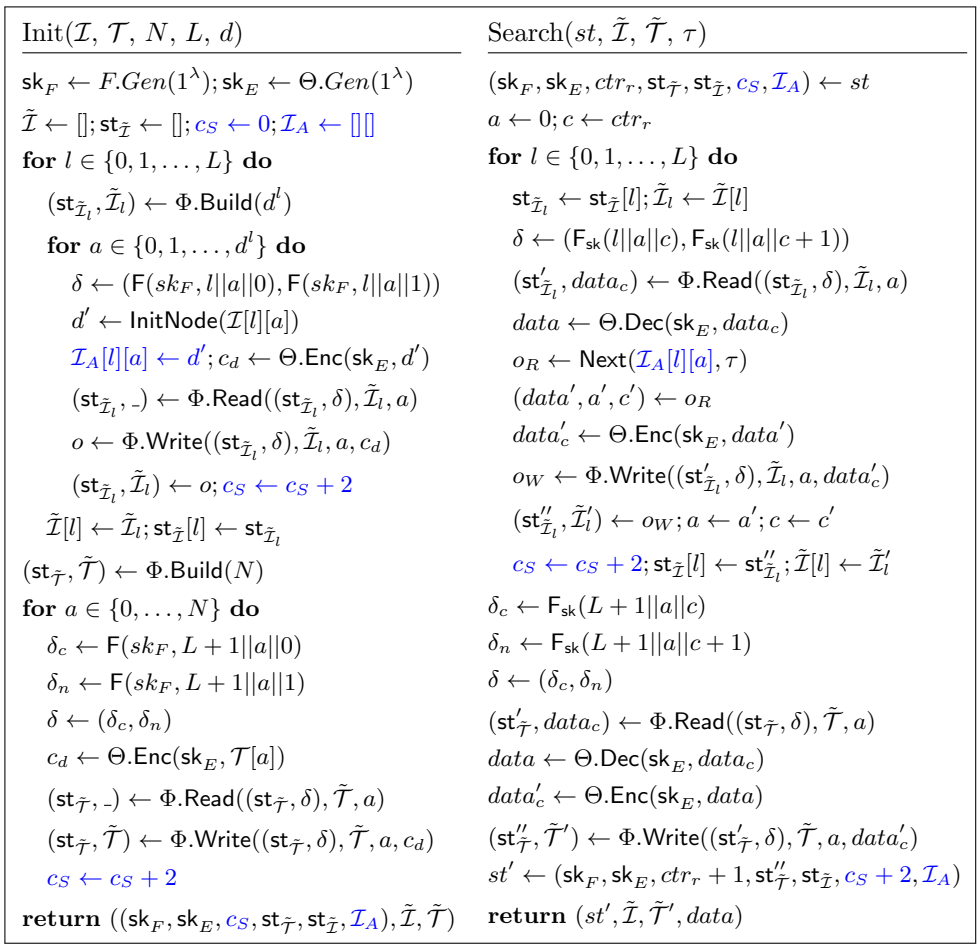


Figure 6: Game 1 hop

with prf-security as defined by *Bellare and Rogway* [5]. Furthermore. let $\Theta = (Gen, Enc, Dec)$ be an IND-CPA encryption scheme according to *Shoup and Boneh* [6]. CODBS is secure according to Definition 3.1 as proven by the following games:

Game G_0. G_0 is defined by the real security game $\mathbf{Real}_{\mathcal{A}}^{OIS}(1^\lambda)$ (Figure 3) instantiated with the CODBS construction which results in the extended real game presented in Figure 4. This extension inlines the InitSearchTree function and encapsulates the addition of a child counter to the tree nodes in the function InitNode.

$$Pr[\mathbf{Real}_{\mathcal{A}}^{OIS}(1^\lambda) = 1] = Pr[G_0 = 1]$$

Game G_1. The following game G_1 makes two modifications on the real world game, which simplify the next steps. First, it adds a global counter to the protocols internal state. The counter does not modify the protocol but provides a unique, non-repeatable value. The counter is incremented twice after every pair of Φ.Read/Φ.Write function, i.e., when the initialization protocol stores a database block in on the external data structures and after a block is accessed during a query search. The second modification is the addition of an ideal structure $\mathcal{I}_A$ that stores the tree-based nodes generated by InitNode. The data stored on this structure is identical to the data blocks stored on the L ORAM levels and does not modify it in any way. As this modification does

Init($\mathcal{I}$, $\mathcal{T}$, N, L, d)	Search(st, $\tilde{\mathcal{I}}$, $\tilde{\mathcal{T}}$, τ)
$\mathsf{sk}_F \leftarrow F.Gen(1^\lambda); g \leftarrow\!\!\$\ Func(D, R)$	$(g, \mathsf{sk}_E, \mathsf{st}_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}}, c_S, \mathcal{I}_A) \leftarrow st$
$\tilde{\mathcal{I}} \leftarrow []; \mathsf{st}_{\tilde{\mathcal{I}}} \leftarrow []; c_S \leftarrow 0; \mathcal{I}_A \leftarrow [][]$	$a \leftarrow 0;$
for $l \in \{0, 1, \ldots, L\}$ **do**	**for** $l \in \{0, 1, \ldots, L\}$ **do**
$(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}_l) \leftarrow \Phi.\mathsf{Build}(d^l)$	$\mathsf{st}_{\tilde{\mathcal{I}}_l} \leftarrow \mathsf{st}_{\tilde{\mathcal{I}}}[l]; \tilde{\mathcal{I}}_l \leftarrow \tilde{\mathcal{I}}[l]$
for $a \in \{0, 1, \ldots, d^l\}$ **do**	$\delta \leftarrow (g(c_S), g(c_S + 1))$
$\delta \leftarrow (g(c_S), g(c_S + 1))$	$(\mathsf{st}'_{\tilde{\mathcal{I}}_l}, data_c) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a)$
$d' \leftarrow \mathsf{InitNode}(\mathcal{I}[l][a])$	$data \leftarrow \Theta.\mathsf{Dec}(\mathsf{sk}_E, data_c)$
$\mathcal{I}_A[l][a] \leftarrow d'; c_d \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, d')$	$(data', a', c') \leftarrow \mathsf{Next}(\mathcal{I}_A[l][a], \tau)$
$(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a)$	$data'_c \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, data')$
$o \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a, c_d)$	$o_W \leftarrow \Phi.\mathsf{Write}((\mathsf{st}'_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a, data'_c)$
$(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}_l) \leftarrow o; c_S \leftarrow c_S + 2$	$(\mathsf{st}''_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}'_l) \leftarrow o_W; a \leftarrow a'; c_S \leftarrow c_S + 2$
$\tilde{\mathcal{I}}[l] \leftarrow \tilde{\mathcal{I}}_l; \mathsf{st}_{\tilde{\mathcal{I}}}[l] \leftarrow \mathsf{st}_{\tilde{\mathcal{I}}_l}$	$\mathsf{st}_{\tilde{\mathcal{I}}}[l] \leftarrow \mathsf{st}''_{\tilde{\mathcal{I}}_l}; \tilde{\mathcal{I}}[l] \leftarrow \tilde{\mathcal{I}}'_l$
$(\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow \Phi.\mathsf{Build}(N)$	$\delta \leftarrow (g(c_S), g(c_S + 1))$
for $a \in \{0, \ldots, N\}$ **do**	$(\mathsf{st}'_{\tilde{\mathcal{T}}}, data_c) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a)$
$\delta \leftarrow (g(c_S), g(c_S + 1)); c_S \leftarrow c_S + 2$	$data \leftarrow \Theta.\mathsf{Dec}(\mathsf{sk}_E, data_c)$
$c_d \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, \mathcal{T}[i])$	$data'_c \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, data)$
$(\mathsf{st}_{\tilde{\mathcal{T}}}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a)$	$(\mathsf{st}''_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}') \leftarrow \Phi.\mathsf{Write}((\mathsf{st}'_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a, data'_c)$
$(\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a, c_d)$	**return** $((g, \mathsf{sk}_E, \mathsf{st}''_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}}, c_S + 2, \mathcal{I}_A), \tilde{\mathcal{I}}, \tilde{\mathcal{T}}', data)$
return $((g, \mathsf{sk}_E, c_S, \mathsf{st}_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}}, \mathcal{I}_A), \tilde{\mathcal{I}}, \tilde{\mathcal{T}})$	

Figure 7: Game 2 hop

not change the game execution it is clear that the adversary gains no additional advantage in this hop.

$$Pr[G_0 = 1] = Pr[G_1 = 1]$$

Game G_2. This game is a two-step hop that alters the process of generating location tokens for every ORAM access. First, every function F is replaced with a real randomly sampled function g. Secondly, the input messages to the functions are replaced by the current value on the global counters. Since these input messages are unique by construction, we are exchanging unique values by unique values, enabling us to use the global counter in the PRF. For every ORAM access either a tree level l, a node offset a or access counter c is different. As the secret key sk_F is outside of the adversary control, an adversary that can distinguish between G_1 and G_2 can also be used to distinguish F from a truly random function. As such, we upper bound the distance between these two games by building an adversary $\mathcal{B}_1$ against the prf-security experiment such that

$$Pr[G_1 = 1] - Pr[G_2 = 1] = \mathsf{Adv}^{\mathrm{prf}}_{\mathsf{F},\mathcal{B}_1}(\lambda)$$

Adversary $\mathcal{B}_1$ simulates the game G_2 as follows. For every requested location token the adversary issues a new call to the $\mathsf{prf}_{\mathsf{F},\mathcal{B}_1}$ oracle. Furthermore, the output bit of G_2 is forwarded as the resulting bit of the prf-security experiment. As the difference between both games is the location token generated either by a $\mathsf{F}(l||a||c)$ or $\mathsf{g}(\mathsf{c_S})$, and both input messages are unique, then the probability of distinguishing between game G_2 and G_1 is the same as prf-security.

Game G_3. With game G_3 the encrypted data blocks are replaced by dummy message with a constant length B. As the contents of the table blocks are never disclosed to the adversary accord-

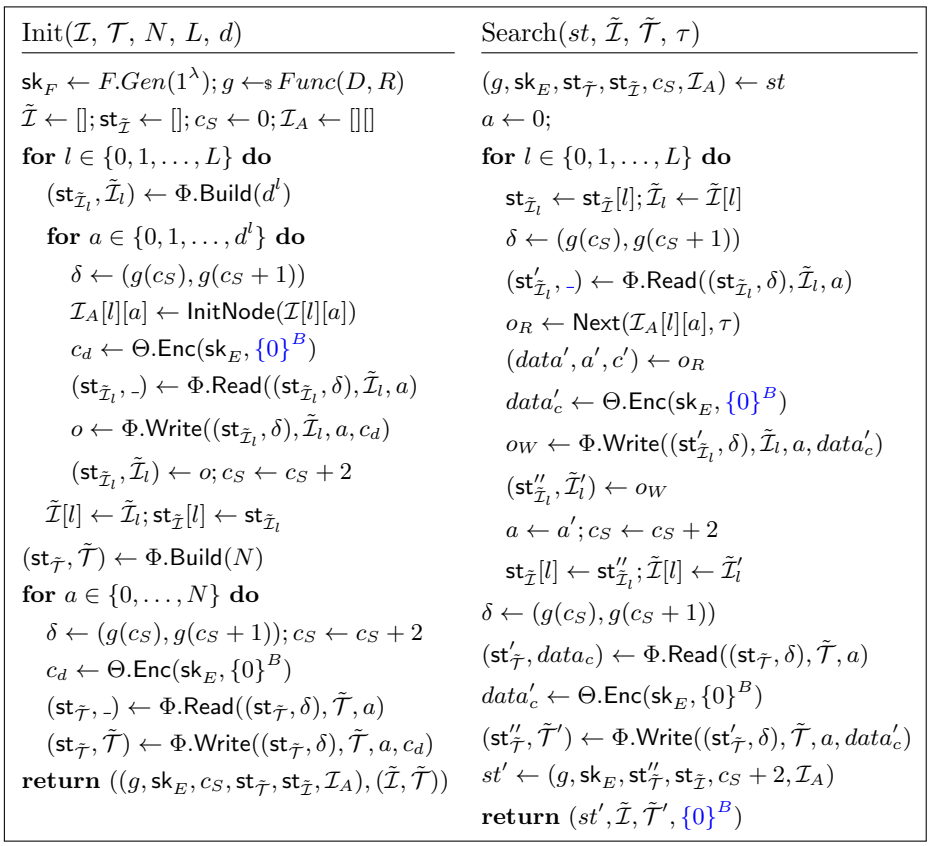


Figure 8: Game 3 hop

ing to the security model and the access counters stored within the tree nodes have been replaced by the global counter, the blocks contents are no longer relevant for the protocol execution. Furthermore, as the adversary $\mathcal{A}$ does not have access to the secret key sk_E the upper bound between game G_4 and game G_3 is defined by an adversary $\mathcal{B}_2$ in an IND-CPA experiment $\mathsf{Adv}^{\text{IND-CPA}}_{\Theta,\mathcal{B}_2}(\lambda)$.

Adversary $\mathcal{B}_2$ simulates the game G_3 by forwarding for every encryption request a pair of input message $(data, \{0\}^B)$ to the $\mathbf{IND\text{-}CPA}_{\theta,\mathcal{B}_2}$ experiment. After the requests, the protocol continues to be executed with the ciphertext returned from the experiment. Once G_3 terminates its resulting bit is returned as the guessing bit of the experiment. Since the difference between both games is the same as presenting the encryption of one of the input messages, then the probability that $\mathcal{A}$ distinguish between G_2 and G_3 is the same as the IND-CPA security.

$$Pr[G_2 = 1] - Pr[G_3 = 1] = \mathsf{Adv}^{\text{IND-CPA}}_{\Theta,\mathcal{B}_2}(\lambda)$$

Game G_4. In this game we replace every block address a in an ORAM read and write requests by the fixed address 0. With this modification, game G_3 has a data request sequence $\overrightarrow{y_3}$ that depends on the input query and database contents while G_4 has a data request sequence $\overrightarrow{y_4}= ((\cdot, 0, \cdot), \ldots, (\cdot, 0, \cdot))$ that always accesses the same address. Since both requests have the exact same length then according to the security definition of an ORAM construction the access pattern generated by both requests are indistinguishable $\mathcal{X}[\Phi](\overrightarrow{y_3}) \approx \mathcal{X}[\Phi](\overrightarrow{y_4})$. In fact, our constructions leverage an ORAM scheme with an external pmap which determines the access patterns generated. Since in both games the pmap sequence $\overrightarrow{\delta}$ is generated by an oracle $\mathcal{O}$ instantiated as a random function g that outputs a unique message independent of the accessed addressed then

Init($\mathcal{I}$, $\mathcal{T}$, N, L, d)	Search(st, $\tilde{\mathcal{I}}$, $\tilde{\mathcal{T}}$, τ)
$\mathsf{sk}_F \leftarrow F.Gen(1^\lambda); g \leftarrow_\$ Func(D, R)$	$(N, L, d, g, \mathsf{sk}_E, \mathsf{st}_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}}, c_S) \leftarrow st$
$\tilde{\mathcal{I}} \leftarrow []; \mathsf{st}_{\tilde{\mathcal{I}}} \leftarrow []; c_S \leftarrow 0; (\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow \Phi.\mathsf{Build}(N)$	**for** $l \in \{0, 1, \ldots, L\}$ **do**
for $l \in \{0, 1, \ldots, L\}$ **do**	$\mathsf{st}_{\tilde{\mathcal{I}}_l} \leftarrow \mathsf{st}_{\tilde{\mathcal{I}}}[l]; \tilde{\mathcal{I}}_l \leftarrow \tilde{\mathcal{I}}[l]$
$(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}_l) \leftarrow \Phi.\mathsf{Build}(d^l)$	$\delta \leftarrow (g(c), g(c+1))$
for $a \in \{0, 1, \ldots, d^l\}$ **do**	$(\mathsf{st}'_{\tilde{\mathcal{I}}_l}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, \{0\}^*)$
$\delta \leftarrow (g(c), g(c+1))$	$data'_c \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, \{0\}^B)$
$c_d \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, \{0\}^B)$	$o_W \leftarrow \Phi.\mathsf{Write}((\mathsf{st}'_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, \{0\}^*, data'_c)$
$(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, \{0\}^*)$	$(\mathsf{st}''_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}'_l) \leftarrow o_W; c_S \leftarrow c_S + 2$
$o \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, \{0\}^*, c_d)$	$\mathsf{st}_{\tilde{\mathcal{I}}}[l] \leftarrow \mathsf{st}''_{\tilde{\mathcal{I}}_l}; \tilde{\mathcal{I}}[l] \leftarrow \tilde{\mathcal{I}}'_l$
$(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}_l) \leftarrow o; c_S \leftarrow c_S + 2$	$\delta \leftarrow (g(c), g(c+1))$
$\tilde{\mathcal{I}}[l] \leftarrow \tilde{\mathcal{I}}_l; \mathsf{st}_{\tilde{\mathcal{I}}}[l] \leftarrow \mathsf{st}_{\tilde{\mathcal{I}}_l}$	$(\mathsf{st}'_{\tilde{\mathcal{T}}}, data_c) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, \{0\}^*)$
for $a \in \{0, \ldots, N\}$ **do**	$data'_c \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, \{0\}^B)$
$\delta \leftarrow (g(c), g(c+1))$	$o \leftarrow \Phi.\mathsf{Write}((\mathsf{st}'_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, \{0\}^*, data'_c)$
$c_d \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, \{0\}^B)$	$(\mathsf{st}''_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}') \leftarrow o$
$(\mathsf{st}_{\tilde{\mathcal{T}}}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, \{0\}^*)$	$st' \leftarrow (N, L, d, g, \mathsf{sk}_E, \mathsf{st}''_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}}, c_S + 2)$
$(\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, \{0\}^*, c_d)$	**return** $(st', \tilde{\mathcal{I}}, \tilde{\mathcal{T}}', \{0\}^B)$
return $((N, L, d, g, \mathsf{sk}_E, c_S + 2, \mathsf{st}_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}}), \tilde{\mathcal{I}}, \tilde{\mathcal{T}})$	

Figure 9: Game 4 hop

the access patterns generated are indistinguishable from an access pattern generated by an ORAM construction with an internal pmap. We upper bound the distance between both games with an hybrid argument [6] of an adversary $\mathcal{B}_3$ that plays against an ORAM experiment $\mathsf{Adv}^{\text{ORAM}}_{\Phi,\mathcal{B}_3}(\lambda)$.

In this game we apply a standard hybrid argument where and adversary $\mathcal{B}_3$ has to distinguish between a sequence of $L+1$ hops such that each hop is denoted as $G_{(3,i)}$ where $0 \leq i \leq L+1$. In the first hop $G_{(3,0)}$ everything in the game is identical to G_3, but in the hop $G_{(3,1)}$ the access to the first ORAM level is made to a fixed address 0. As everything remains equal, distinguishing between these two hops is the same as distinguishing between two access patterns generated by an ORAM construction. This argument can be applied recursively in the subsequent levels by changing one level at each step, from $G_{(3,i)}$ to $G_{(3,(i+1))}$ where $i \in [0..L]$. In the last game, $G_{(3,(L+1))}$ is exactly equal to G_4. As such, the advantage of an adversary $\mathcal{A}$ distinguishing between G_4 and G_3 is the same as $\mathcal{B}_3$ distinguishing the access patterns generated of just one of the $L+1$ ORAM levels in the sequence of from $G_{(3,0)}$ to $G_{(3,(L+1))}$. The upper bound is given by:

$$Pr[G_3 = 1] - Pr[G_4 = 1] = (L+1) \cdot \mathsf{Adv}^{\text{ORAM}}_{\Phi,\mathcal{B}_3}(\lambda)$$

Let,

$$\mathsf{Adv}^{\text{OBLIVS}}_{OIS,S,\mathcal{A}}(\lambda) = \left| \Pr\left[\mathbf{Real}^{OIS}_{\mathcal{A}}(1^\lambda) = 1\right] - \Pr\left[\mathbf{Ideal}^{OIS}_{S,\mathcal{A}}(1^\lambda) = \right] \right|$$

then Theorem 1 follows from

$$\mathsf{Adv}^{\text{OBLIVS}}_{OIS,S,\mathcal{A}}(\lambda) = \sum_{i=0}^{4} |Pr[G_i = 1] - Pr[G_{i+1} = 1]| \leq$$

$$\mathsf{Adv}^{\text{prf}}_{\mathsf{F},\mathcal{B}_1}(\lambda) + \mathsf{Adv}^{\text{IND-CPA}}_{\Theta,\mathcal{B}_2}(\lambda) + L + 1 \cdot \mathsf{Adv}^{\text{ORAM}}_{\Phi,\mathcal{B}_3}(\lambda) \leq \mathsf{negl}(\lambda)$$

■

7 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.

7.1 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[2].

We implemented CODBS as the database component that replaces the *Trusted Proxy* and provides an input API similar to the definition in Section 3.1. 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 [12] 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.

7.2 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 distribution $\{0,1\}^B$. In the macro setting we use the YCSB benchmark v0.18 [10]. 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) [23] with the measured standard deviation. Each run lasted

[2] `https://github.com/rogerioacp/oram`

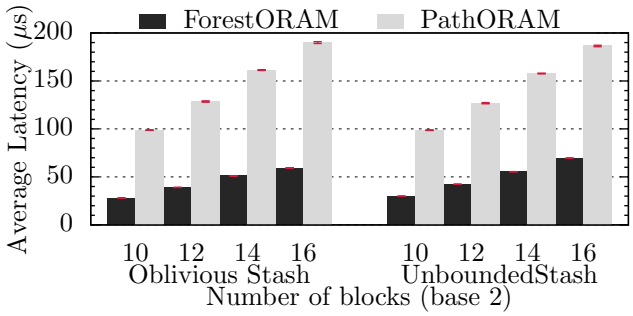


Figure 10: Forest ORAM and Path ORAM comparison. X-axis measures number of blocks and errors bars the 95% CI.

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 the systems caches and deleting any 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 [23]. We collected CPU, memory and disk usage of each system and ORAM construction using Dstat v0.7.3.

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. Additionally, the nodes were interconnected by a 10 GiB network switch.

7.3 Micro Benchmark

Figure 10 depicts the results of the micro benchmark. The workload in this benchmark with an initially empty oblivious data structure and measures the latency of an oblivious access request, either a read or a write. With this workload we measure the average latency of a request for the Forest ORAM and Path ORAM constructions. We also measure 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

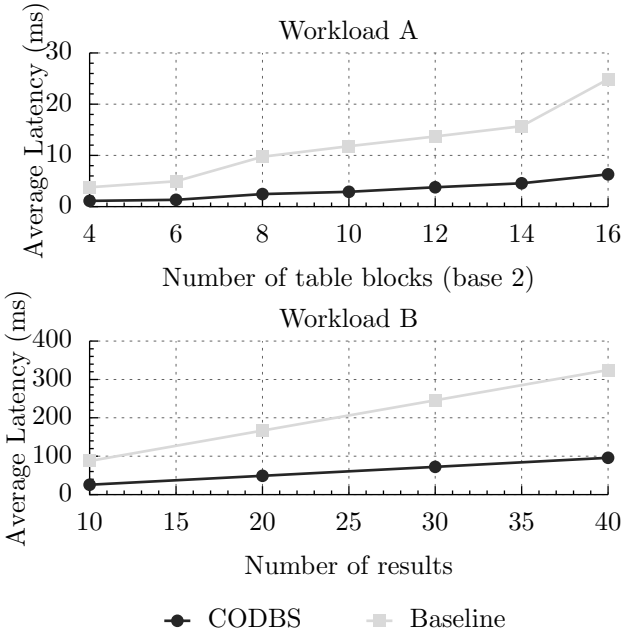


Figure 11: Avg. latency of YCSB workloads. Workload A X-axis measures the numbers of blocks. Workload B X-axis measures a query resulting records.

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

7.4 Macro Benchmark

We now present the performance of a complete CODBS deployment 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* [28, 37]. 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 11 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

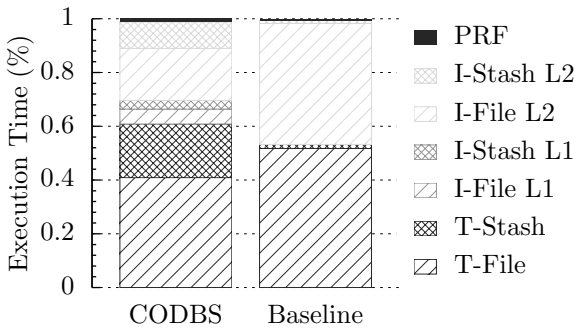


Figure 12: Breakdown of time spent during query execution.

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 ∼ 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 ∼ 3× speedup with the lowest throughput of 10 ops/s and an average latency of 96 ms. The plaintext PostgreSQL has the highest performance decrease of ∼ 26% on the largest dataset.

Figure 12 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 13 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.

7.5 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 7.3. 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× speedup instead of just a 2.6× 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 the `baseline` are less significant. With just a 2× 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.

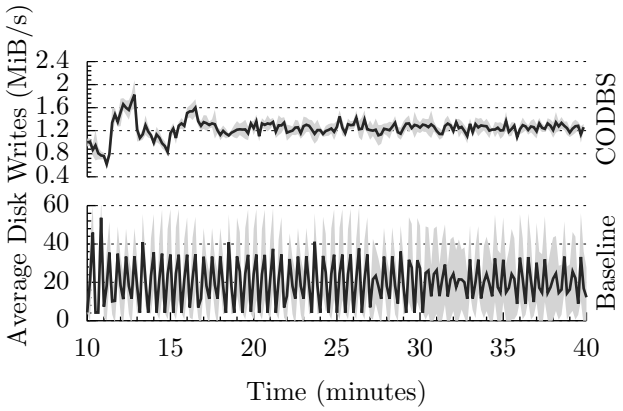


Figure 13: Avg. disk writes over time on workload A on data set with 2^{16} records. The light gray represents the 95% CI.

8 Related Work

Position-based ORAMs. Position-based ORAMs were first proposed by *Shi et al.* [31] to improve the lower bounds of classic constructions [16, 17]. 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 [15, 9, 36] 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 [13, 26, 1] or, as proposed in Burst ORAM [11] and Circuit ORAM [36], be a simple compression with an XOR operator.

Oblivious Data Structures. *Wang et al.* [37] 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.* [18] and shown by novel research [24, 25, 19], hiding the access patterns of a database is not enough. The volume leakage of database queries must also be addressed. *Kallaris et al.* [24] was the first to create a formal model of encrypted database that focus on volume leakage. New attacks have been proposed [18].

9 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× to 4× faster and only requires a small constant size storage that can be deployed within an Intel SGX enclave.

Acknowledgements

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

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

[3] Asharov, G., Komargodski, I., Lin, W., Nayak, K., and Shi, E. Optorama: Optimal oblivious RAM. *IACR Cryptology ePrint Archive 2018* (2018), 892.

[4] Barbosa, M., Portela, B., Scerri, G., and Warinschi, B. 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* (2016), IEEE, pp. 245–260.

[5] Bellare, M., and Rogaway, P. Introduction to modern cryptography. In *UCSD CSE 207 Course Notes* (2005), p. 207.

[6] Boneh, D., and Shoup, V. A graduate course in applied cryptography. p. 818.

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

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

[9] Chung, K., Liu, Z., and Pass, R. Statistically-secure ORAM with $\tilde{o}(\log^2 n)$ overhead. In *ASIACRYPT (2)*.

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

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

[12] Denis, F. The sodium cryptography library, Fev 2020.

[13] Devadas, S., van Dijk, M., Fletcher, C. W., Ren, L., Shi, E., and Wichs, D. Onion oram: A constant bandwidth blowup oblivious ram. In *Theory of Cryptography*, E. Kushilevitz and T. Malkin, Eds.

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

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

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

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

[18] Grubbs, P., Lacharite, M.-S., Minaud, B., and Paterson, K. G. 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* (2018), CCS '18.

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

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

[21] Hoang, T., Ozmen, M. O., Jang, Y., and Yavuz, A. A. Hardware-supported ORAM in effect: Practical oblivious search and update on very large dataset. *PoPETs 2019*, 1 (2019), 172–191.

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

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

[24] Kellaris, G., Kollios, G., Nissim, K., and O'Neill, A. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM.

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

[26] Mayberry, T., Blass, E., and Chan, A. H. Efficient private file retrieval by combining ORAM and PIR. In *NDSS* (2014), The Internet Society.

[27] McKeen, F., Alexandrovich, I., Anati, I., Caspi, D., Johnson, S., Leslie-Hurd, R., and Rozas, C. 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*.

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

[29] Patel, S., Persiano, G., Raykova, M., and Yeo, K. Panorama: Oblivious ram with logarithmic overhead. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)* (Oct 2018), pp. 871–882.

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

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

[32] Stefanov, E., Dijk, M. V., Shi, E., Chan, T.-H. H., Fletcher, C., Ren, L., Yu, X., and Devadas, S. Path oram: An extremely simple oblivious ram protocol. *J. ACM 65*, 4 (Apr. 2018), 18:1–18:26.

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

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

[35] Stefanov, E., Shi, E., and Song, D. X. Towards practical oblivious ram. In *NDSS* (2012), The Internet Society.

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

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

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

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