An Optimal Relational Database Encryption Scheme

Seny Kamara [*] Brown University
Tarik Moataz[†] Aroki Systems
Stan Zdonik [‡] Brown University
Zheguang Zhao[§] Brown University

Abstract

Recently, Kamara and Moataz described the first encrypted relational database solution with support for a non-trivial fraction of SQL that does not make use of property-preserving encryption (*Asiacrypt*, 2018). More precisely, their construction, called SPX, handles the set of conjunctive SQL queries. While SPX was shown to be optimal for the subset of uncorrelated conjunctive SQL queries, it did not handle correlated queries optimally. Furthermore, it only handles queries in heuristic normal form. In this work, we address these limitations by proposing an extension of SPX that handles all conjunctive SQL queries optimally no matter what form they are in.

[*] seny@brown.edu
[†] tarik@aroki.com
[‡] sbz@cs.brown.edu
[§] zheguang.zhao@brown.edu

Contents

1 Introduction

End-to-end encrypted relational database management systems encrypt relational database in such a way that they can be be privately queried. This problem was first considered by Hacigümüs, Iyer, Li and Mehrotra [15] who used a quantization-based approach that leaked the range within which an item fell. In [1], Popa, Redfield, Zeldovich and Balakrishnan described a system called CryptDB that could support a non-trivial subset of SQL without quantization. CryptDB achieved this in part through the use of property-preserving encryption (PPE) schemes like deterministic and order-preserving encryption [2, 4, 5]. This approach was adopted by other systems including Cipherbase [3] and SEEED [23]. While these systems were efficient and legacy-friendly, it was shown by Naveed, Kamara and Wright [21], that they leaked a non-trivial amount of information.

An alternative approach to designing encrypted databases is to use structured encryption (STE) [9] which is a generalization of index-based searchable symmetric encryption [24, 11]. STE-based systems leak less than their PPE-based counterparts while achieving similar efficiency. Initial STE-based solutions, however, have had two major limitations. The first is that they are not legacy-friendly and require custom database management systems. The second is that they could only handle a limited fraction of SQL. For example, systems based on standard STE techniques like Blind Seer [22, 12] and ESPADA [7, 6] can handle filtering and range queries but not joins or projections. This limitation was addressed recently by Kamara and Moataz [16] who proposed an STE-based scheme called SPX that handles a non-trivial fraction of SQL queries; specifically the set of conjunctive SQL queries, which have the form,

$$\texttt{SELECT}\ \text{attributes}\ \texttt{FROM}\ \text{tables}\ \texttt{WHERE}\ \mathsf{att}_1 = a \wedge \mathsf{att}_2 = \mathsf{att}_3\,.$$

In addition to handling a large subset of SQL queries, the SPX construction was also shown to be efficient and even optimal for the subset of uncorrelated queries which, roughly speaking, are queries of the above form where the attributes are all distinct and from different tables. Though SPX is practical, it is not efficient enough to yield a system that is competitive with commercial plaintext database management systems (DBMS). This stems from several reasons which we now discuss.

Query processing and optimization. Database systems process SQL queries in a series of steps. First, a SQL query is converted into a *logical query tree* which is a tree-based representation of the query where each node is a relational algebra operator. Query trees are evaluated bottom up by evaluating the operators at the leaves on the appropriate database tables. The intermediate table that results from an operation is then passed on to its parent node until the final result table is output by the root. The initial query tree is then converted by a *query optimizer* to an equivalent but optimized query tree using various optimization techniques.

Query optimizers are one of the most important components of a DBMS and a large part of why commercial systems are so efficient. It follows then that for encrypted database systems to be competitive with commercial systems, they must support some form of query optimization. As described, however, the SPX construction does not allow for query optimization because it only handles queries in heuristic normal form (HNF) which is a very specific form of query tree.

An overview of SPX. We briefly recall how SPX works at a high-level. First, note that any conjunctive SQL query can be represented as an SPC query [8] which, in turn, can be represented as a query tree with select/filter, projection and cross product operations. SPX makes use of two kinds of encrypted data structures: encrypted multi-maps (EMM) which map encrypted labels to

encrypted tuples and encrypted dictionaries which map encrypted labels to encrypted values. A database $\mathsf{DB} = (\mathbf{T}_1, \ldots, \mathbf{T}_n)$ is encrypted as $\mathsf{EDB} = (\mathsf{EMM}_R, \mathsf{EMM}_C, \mathsf{EMM}_V, \mathsf{EDX})$ where EMM_R and EMM_C store encryptions of the rows and columns in the database, respectively; where EMM_V is used to process filter operations and where EDX is an encrypted dictionary that stores a set of encrypted multi-maps $\{\mathsf{EMM}_{\mathbf{c},\mathbf{c}'}\}_{\mathbf{c},\mathbf{c}' \in \mathsf{DB}}$ that are used to process joins between columns $\mathbf{c}$ and $\mathbf{c}'$. Given a query tree, SPX evaluates leaf operations by querying one of its EMMs directly and then uses various algorithms to process the internal operations on intermediate results. While the leaf operations are handled optimally thanks to the EMMs, internal operations are not necessarily handled in optimal or even sub-linear time.

Sub-optimality of correlated queries. Another source of SPX's sub-optimality is comes from how it handles correlated queries. Roughly speaking, a conjunctive SQL query is uncorrelated if the terms of its `WHERE` clause include attributes/columns that are in different tables. The query trees of uncorrelated queries are relatively simple: they have height 1 with leaves that are either join or filter operations and a root that is a Cartesian product. From the discussion above, one can see that SPX can handle these queries very efficiently since leaf operations are evaluated optimally by directly querying the EMMs. Correlated queries, on the other hand, have query trees of height 2 or more which means they have internal operations which, as discussed above, are not necessarily handled optimally.

Our contributions. In this work, we describe an extension of the SPX construction [16], called OPX, that supports query optimization and handles all conjunctive SQL queries optimally. It does this by using additional encrypted structures that are designed to optimally handle internal operations. These additional structures include an encrypted set structure to handle internal filters and an additional set of encrypted multi-maps to handle internal joins. These additional structures increase the storage overhead but only concretely; asymptotically-speaking OPX has the same storage overhead as SPX. The leakage profile of OPX is also similar to that of SPX. In addition to executing internal operations more efficiently, OPX has the advantage that it can handle any query tree; not just HNF trees. This is an important feature because it means that OPX can be used to query trees that have been optimized by standard query optimizers.

Related work. Since we already discussed related work on encrypted database schemes and systems, we omit a formal related work section.

s

2 Preliminaries

Notation. The set of all binary strings of length n is denoted as $\{0,1\}^n$, and the set of all finite binary strings as $\{0,1\}^*$. $[n]$ is the set of integers $\{1, \ldots, n\}$. We write $x \leftarrow \chi$ to represent an element x being sampled from a distribution χ, and $x \xleftarrow{\$} X$ to represent an element x being sampled uniformly at random from a set X. The output x of an algorithm $\mathcal{A}$ is denoted by $x \leftarrow \mathcal{A}$. Given a sequence $\mathbf{v}$ of n elements, we refer to its ith element as v_i or $\mathbf{v}[i]$. If S is a set then $\#S$ refers to its cardinality. If s is a string then $|s|$ refers to its bit length.

Basic cryptographic primitives. A private-key encryption scheme is a set of three polynomial-time algorithms $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ such that Gen is a probabilistic algorithm that takes a security parameter k and returns a secret key K; Enc is a probabilistic algorithm that takes a key K and a message m and returns a ciphertext c; Dec is a deterministic algorithm that takes a key K and a ciphertext c and returns m if K was the key under which c was produced. Informally, a private-key encryption scheme is secure against chosen-plaintext attacks (CPA) if the ciphertexts it outputs do not reveal any partial information about the plaintext even to an adversary that can adaptively query an encryption oracle. We say a scheme is random-ciphertext-secure against chosen-plaintext attacks (RCPA) if the ciphertexts it outputs are computationally indistinguishable from random even to an adversary that can adaptively query an encryption oracle.[1] In addition to encryption schemes, we also make use of pseudo-random functions (PRF) and permutations (PRP), which are polynomial-time computable functions that cannot be distinguished from random functions by any probabilistic polynomial-time adversary. We refer the reader to [20] for formal definitions of CPA-security, PRFs and PRPs.

Basic structures. We make use of several basic data types including dictionaries and multi-maps which we recall here. A dictionary DX of capacity n is a collection of n label/value pairs $\{(\ell_i, v_i)\}_{i \leq n}$ and supports get and put operations. We write $v_i := \mathsf{DX}[\ell_i]$ to denote getting the value associated with label ℓ_i and $\mathsf{DX}[\ell_i] := v_i$ to denote the operation of associating the value v_i in DX with label ℓ_i. A multi-map MM with capacity n is a collection of n label/tuple pairs $\{(\ell_i, \mathbf{t}_i)_i\}_{i \leq n}$ that supports get and put operations. Similarly to dictionaries, we write $\mathbf{t}_i := \mathsf{MM}[\ell_i]$ to denote getting the tuple associated with label ℓ_i and $\mathsf{MM}[\ell_i] := \mathbf{t}_i$ to denote operation of associating the tuple $\mathbf{t}_i$ to label ℓ_i. Multi-maps are the abstract data type instantiated by an inverted index. In the encrypted search literature multi-maps are sometimes referred to as indexes, databases or tuple-sets (T-sets).

Relational databases. A relational database $\mathsf{DB} = (\mathbf{T}_1, \ldots, \mathbf{T}_n)$ is a set of *tables* where each table $\mathbf{T}_i$ is a two-dimensional array with rows corresponding to an entity (e.g., a customer or an employee) and columns corresponding to attributes (e.g., age, height, salary). For any given attribute, we refer to the set of all possible values that it can take as its *space* (e.g., integers, booleans, strings). We define the *schema* of a table $\mathbf{T}$ to be its set of attributes and denote it $\mathbb{S}(\mathbf{T})$. The schema of a database $\mathsf{DB} = (\mathbf{T}_1, \ldots, \mathbf{T}_n)$ is then the set $\mathbb{S}(\mathsf{DB}) = \bigcup_i \mathbb{S}(\mathbf{T}_i)$. We assume the attributes in $\mathbb{S}(\mathsf{DB})$ are unique and represented as positive integers. We denote a table $\mathbf{T}$'s number of rows as $\|\mathbf{T}\|_r$ and its number of columns as $\|\mathbf{T}\|_c$.

We sometimes view tables as a tuple of rows and write $\mathbf{r} \in \mathbf{T}$ and sometimes as a tuple of columns and write $\mathbf{c} \in \mathbf{T}^\intercal$. Similarly, we write $\mathbf{r} \in \mathsf{DB}$ and $\mathbf{c} \in \mathsf{DB}^\intercal$ for $\mathbf{r} \in \bigcup_i \mathbf{T}_i$ and $\mathbf{c} \in \bigcup_i \mathbf{T}_i^\intercal$, respectively. For a row $\mathbf{r} \in \mathbf{T}_i$, its table identifier $\mathsf{tbl}(\mathbf{r})$ is i and its row rank $\mathsf{rrk}(\mathbf{r})$ is its position in $\mathbf{T}_i$ when viewed as a tuple of rows. Similarly, for a column $\mathbf{c} \in \mathbf{T}_i^\intercal$, its table identifier $\mathsf{tbl}(\mathbf{c})$ is i and its column rank $\mathsf{crk}(\mathbf{c})$ is its position in $\mathbf{T}_i$ when viewed as a tuple of columns. For any row $\mathbf{r} \in \mathsf{DB}$ and column $\mathbf{c} \in \mathsf{DB}^\intercal$, we refer to the pairs $\chi(\mathbf{r}) \stackrel{def}{=} (\mathsf{tbl}(\mathbf{r}), \mathsf{rrk}(\mathbf{r}))$ and $\chi(\mathbf{c}) \stackrel{def}{=} (\mathsf{tbl}(\mathbf{c}), \mathsf{crk}(\mathbf{c}))$, respectively, as their *coordinates* in DB. We write $\mathbf{r}[i]$ and $\mathbf{c}[i]$ to refer to the ith element of a row $\mathbf{r}$ and column $\mathbf{c}$. The coordinate of the jth cell in row $\mathbf{r} \in \mathbf{T}_i$ is the triple $(i, \mathsf{rrk}(\mathbf{r}), j)$. Given a column $\mathbf{c} \in \mathsf{DB}^\intercal$, we denote its corresponding attribute by $\mathsf{att}(\mathbf{c})$. For

[1] RCPA-secure encryption can be instantiated practically using either the standard PRF-based private-key encryption scheme or, e.g., AES in counter mode.

any pair of attributes $\mathsf{att}_1, \mathsf{att}_2 \in \mathbb{S}(\mathsf{DB})$ such that $\mathsf{dom}(\mathsf{att}_1) = \mathsf{dom}(\mathsf{att}_2)$, $\mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2}$ denotes the set of row pairs $\{(\mathbf{r}_1, \mathbf{r}_2) \in \mathsf{DB}^2 : \mathbf{r}_1[\mathsf{att}_1] = \mathbf{r}_2[\mathsf{att}_2]\}$. For any attribute $\mathsf{att} \in \mathbb{S}(\mathsf{DB})$ and constant $a \in \mathsf{dom}(\mathsf{att})$, $\mathsf{DB}_{\mathsf{att}=a}$ is the set of rows $\{\mathbf{r} \in \mathsf{DB} : \mathbf{r}[\mathsf{att}] = a\}$.

SQL. In practice, relational databases are queried using the special-purpose language called SQL. SQL is a declarative language and can be used to modify and query a relational DB. In this work, we only focus on its query operations. Informally, SQL queries have the form

`SELECT` attributes `FROM` tables `WHERE` condition,

where *attributes* is a set of attributes/columns, *tables* is a set of tables and *condition* is a predicate over the rows of *tables* and can itself contain a nested SQL query. More complex queries can be obtained using Group-by, Order-by and aggregate operators (i.e., max, min, average etc.) but the simple form above already captures a large subset of SQL. The most common class of queries on relational DBs are *conjunctive queries* [8] which have the above form with the restriction that *condition* is a conjunction of equalities over attributes and constants. In particular, this means there are no nested queries in *condition*. More precisely, conjunctive queries have the form

`SELECT` attributes `FROM` tables `WHERE` $(\mathsf{att}_1 = X_1 \wedge \cdots \wedge \mathsf{att}_\ell = X_\ell)$,

where att_i is an attribute in $\mathbb{S}(\mathsf{DB})$ and X_i can be either an attribute or a constant.

The SPC algebra. It was shown by Chandra and Merlin [8] that conjunctive queries could be expressed as a subset of Codd's relational algebra which is an imperative query language based on a set of basic operators. In particular, they showed that three operators *select*, *project* and *cross product* were enough. The *select* operator σ_Ψ is parameterized with a predicate Ψ and takes as input a table $\mathbf{T}$ and outputs a new table $\mathbf{T}'$ that includes the rows of $\mathbf{T}$ that satisfy the predicate Ψ. The *projection* operator $\pi_{\mathsf{att}_1,\ldots,\mathsf{att}_h}$ is parameterized by a set of attributes $\mathsf{att}_1, \ldots, \mathsf{att}_h$ and takes as input a table $\mathbf{T}$ and outputs a table $\mathbf{T}'$ that consists of the columns of $\mathbf{T}$ indexed by att_1 through att_n. The *cross product* operator $\times$ takes as input two tables $\mathbf{T}_1$ and $\mathbf{T}_2$ and outputs a new table $\mathbf{T}' = \mathbf{T}_1 \times \mathbf{T}_2$ such that each row of $\mathbf{T}'$ is an element of the cross product between the set of rows of $\mathbf{T}_1$ and the set of rows of $\mathbf{T}_2$. The query language that results from any combination of select, project and cross product is referred to as the *SPC algebra*. We formalize this in Definition 2.1 below.

Definition 2.1 (SPC algebra)**.** *Let* $\mathsf{DB} = (\mathbf{T}_1, \ldots, \mathbf{T}_n)$ *be a relational database. The SPC algebra consists of any query that results from the combination of the following operators:*

- $\mathbf{T}' \leftarrow \sigma_\Psi(\mathbf{T})$*: the* select *operator is parameterized with a predicate* Ψ *of form* $\mathsf{att}_1 = X_1 \wedge \cdots \wedge \mathsf{att}_\ell = X_\ell$*, where* $\mathsf{att}_i \in \mathbb{S}(\mathsf{DB})$ *and* X_i *is either a constant* a *in the domain of* att_i *(type-1) or an attribute* $\mathsf{x}_j \in \mathbb{S}(\mathsf{DB})$ *(type-2). It takes as input a table* $\mathbf{T} \in \mathsf{DB}$ *and outputs a table* $\mathbf{T}' = \{\mathbf{r} \in \mathbf{T} : \Psi(\mathbf{r}) = 1\}$*, where terms of the form* $\mathsf{att}_i = \mathsf{x}_j$ *are satisfied if* $\mathbf{r}[\mathsf{att}_i] = \mathbf{r}[\mathsf{x}_j]$ *and terms of the form* $\mathsf{att}_i = a$ *are satisfied if* $\mathbf{r}[\mathsf{att}_i] = a$*.*

- $\mathbf{T}' \leftarrow \pi_{\mathsf{att}_1,\ldots,\mathsf{att}_h}(\mathbf{T})$*: the* project *operator is parameterized by a set of attributes* $\mathsf{att}_1, \ldots, \mathsf{att}_h \in \mathbb{S}(\mathsf{DB})$*. It takes as input a table* $\mathbf{T} \in \mathsf{DB}$ *and outputs a table* $\mathbf{T}' = \{\langle \mathbf{r}[\mathsf{att}_1], \ldots, \mathbf{r}[\mathsf{att}_h] \rangle : \mathbf{r} \in \mathbf{T}\}$*.*

- $\mathbf{R} \leftarrow \mathbf{T}_1 \times \mathbf{T}_2$*: the* cross product *operator takes as input two tables* $\mathbf{T}_1$ *and* $\mathbf{T}_2$ *and outputs a result table* $\mathsf{R} = \{\langle \mathbf{r}, \mathbf{v} \rangle : \mathbf{r} \in \mathbf{T}_1 \text{ and } \mathbf{v} \in \mathbf{T}_2\}$*, where* $\langle \mathbf{r}, \mathbf{v} \rangle$ *is the concatenation of rows* $\mathbf{r}$ *and* $\mathbf{v}$*.*

We will often also consider the inner join *operator which is defined as follows:*

- $\mathbf{R} \leftarrow \mathbf{T}_1 \bowtie_{\mathsf{att}_1 = \mathsf{att}_2} \mathbf{T}_2$*: the* inner join *operator is parameterized by an expression of the form* $\mathsf{att}_1 = \mathsf{att}_2$*. It takes as input two tables* $\mathbf{T}_1$ *and* $\mathbf{T}_2$ *and outputs a result table* $\mathbf{R} = \{\langle \mathbf{r}, \mathbf{v} \rangle : r \in \mathbf{T}_1 \text{ and } \mathbf{v} \in \mathbf{T}_2 \text{ and } \mathbf{r}[\mathsf{att}_1] = \mathbf{v}[\mathsf{att}_2]\}$*.*

Note that adding the inner join operator to the SPC algebra does not increase its power as inner joins can be computed using select and cross product operations. This is just a notational convenience.

Intuitively, the connection between conjunctive SQL queries and the SPC algebra can be seen as follows: `SELECT` corresponds to the projection operator, `FROM` to the cross product and `WHERE` to the (SPC) select operator.

Query trees. Every query in the SPC algebra can be represented as a query tree which is a a tree-based representation of the query. A query can have several query tree representations each leading to a different query complexity when executed. As an example, the following conjunctive SQL query:

$$\texttt{SELECT } \mathbf{T}_1.\text{ID } \texttt{FROM } \mathbf{T}_1, \mathbf{T}_2 \texttt{ WHERE } \mathbf{T}_2.\text{Department} = \text{Math} \wedge \mathbf{T}_2.\text{Course} = \mathbf{T}_1.\text{Course}\,,$$

has at least three possible query tree representations, which are illustrated in Figure (**??**).

3 Definitions

In this Section, we define the syntax and security of STE schemes. A STE scheme encrypts data structures in such a way that they can be privately queried. There are several natural forms of structured encryption. The original definition of [9] considered schemes that encrypt both a structure and a set of associated data items (e.g., documents, emails, user profiles etc.). In [10], the authors also describe *structure-only* schemes which only encrypt structures. Another distinction can be made between *interactive* and *non-interactive* schemes. Interactive schemes produce encrypted structures that are queried through an interactive two-party protocol, whereas non-interactive schemes produce structures that can be queried by sending a single message, i.e, the token. One can also distinguish between *response-hiding* and *response-revealing* schemes: the latter reveal the query response to the server whereas the former do not.

In this work, we focus on non-interactive structure-only schemes. Our main construction, OPX, is response-hiding but makes use of response-revealing schemes as building blocks. As such, we define both forms below. At a high-level, non-interactive STE works as follows. During a setup phase, the client constructs an encrypted structure EDS under a key K from a plaintext structure DS. The client then sends EDS to the server. During the query phase, the client constructs and sends a token tk generated from its query q and secret key K. The server then uses the token tk to query EDS and recover either a response r or an encryption ct of r depending on whether the scheme is response-revealing or response-hiding.

Definition 3.1 (Response-revealing structured encryption [9])**.** *A response-revealing structured encryption scheme* $\Sigma = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query})$ *consists of three polynomial-time algorithms that work as follows:*

- $(K, \mathsf{EDS}) \leftarrow \mathsf{Setup}(1^k, \mathsf{DS})$*: is a probabilistic algorithm that takes as input a security parameter* 1^k *and a structure* DS *and outputs a secret key* K *and an encrypted structure* EDS*.*
- $\mathsf{tk} \leftarrow \mathsf{Token}(K, q)$*: is a (possibly) probabilistic algorithm that takes as input a secret key* K *and a query* q *and returns a token* tk*.*
- $\{\perp, r\} \leftarrow \mathsf{Query}(\mathsf{EDS}, \mathsf{tk})$*: is a deterministic algorithm that takes as input an encrypted structure* EDS *and a token* tk *and outputs either* $\perp$ *or a response.*

We say that a response-revealing structured encryption scheme Σ *is correct if for all* $k \in \mathbb{N}$*, for all* $\mathsf{poly}(k)$*-size structures* $\mathsf{DS} : \mathbf{Q} \to \mathbf{R}$*, for all* (K, EDS) *output by* $\mathsf{Setup}(1^k, \mathsf{DS})$ *and all sequences of* $m = \mathsf{poly}(k)$ *queries* $q_1, \ldots, q_m$*, for all tokens* tk_i *output by* $\mathsf{Token}(K, q_i)$*,* $\mathsf{Query}(\mathsf{EDS}, \mathsf{tk}_i)$ *returns* $\mathsf{DS}(q_i)$ *with all but negligible probability.*

Definition 3.2 (Response-hiding structured encryption [9])**.** *A response-hiding structured encryption scheme* $\Sigma = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query}, \mathsf{Dec})$ *consists of four polynomial-time algorithms such that* Setup *and* Token *are as in Definition 3.1 and* Query *and* Dec *are defined as follows:*

- $\{\perp, \mathrm{ct}\} \leftarrow \mathsf{Query}(\mathsf{EDS}, \mathsf{tk})$*: is a deterministic algorithm that takes as input an encrypted structured* EDS *and a token* tk *and outputs either* $\perp$ *or a ciphertext* ct*.*
- $r \leftarrow \mathsf{Dec}(K, \mathrm{ct})$*: is a deterministic algorithm that takes as input a secret key* K *and a ciphertext* ct *and outputs a response* r*.*

We say that a response-hiding structured encryption scheme Σ *is correct if for all* $k \in \mathbb{N}$*, for all* $\mathsf{poly}(k)$*-size structures* $\mathsf{DS} : \mathbf{Q} \to \mathbf{R}$*, for all* (K, EDS) *output by* $\mathsf{Setup}(1^k, \mathsf{DS})$ *and all sequences of* $m = \mathsf{poly}(k)$ *queries* $q_1, \ldots, q_m$*, for all tokens* tk_i *output by* $\mathsf{Token}(K, q_i)$*,* $\mathsf{Dec}_K\Big(\mathsf{Query}\Big(\mathsf{EDS}, \mathsf{tk}_i\Big)\Big)$ *returns* $\mathsf{DS}(q_i)$ *with all but negligible probability.*

Security. The standard notion of security for structured encryption guarantees that an encrypted structure reveals no information about its underlying structure beyond the setup leakage $\mathcal{L}_{\mathsf{S}}$ and that the query algorithm reveals no information about the structure and the queries beyond the query leakage $\mathcal{L}_{\mathsf{Q}}$. If this holds for non-adaptively chosen operations then this is referred to as non-adaptive semantic security. If, on the other hand, the operations are chosen adaptively, this leads to the stronger notion of adaptive semantic security. This notion of security was introduced by Curtmola *et al.* in the context of SSE [11] and later generalized to structured encryption in [9].

Definition 3.3 (Adaptive semantic security [11, 9])**.** *Let* $\Sigma = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query})$ *be a response-revealing structured encryption scheme and consider the following probabilistic experiments where* $\mathcal{A}$ *is a stateful adversary,* $\mathcal{S}$ *is a stateful simulator,* $\mathcal{L}_{\mathsf{S}}$ *and* $\mathcal{L}_{\mathsf{Q}}$ *are leakage profiles and* $z \in \{0,1\}^*$*:*

Real$_{\Sigma,\mathcal{A}}(k)$*: given* z *the adversary* $\mathcal{A}$ *outputs a structure* DS*. It receives* EDS *from the challenger, where* $(K, \mathsf{EDS}) \leftarrow \mathsf{Setup}(1^k, \mathsf{DS})$*. The adversary then* adaptively *chooses a polynomial number of queries* $q_1, \ldots, q_m$*. For all* $i \in [m]$*, the adversary receives* $\mathsf{tk} \leftarrow \mathsf{Token}(K, q_i)$*. Finally,* $\mathcal{A}$ *outputs a bit* b *that is output by the experiment.*

Ideal$_{\Sigma,\mathcal{A},\mathcal{S}}(k)$*: given z the adversary $\mathcal{A}$ generates a structure* DS *which it sends to the challenger. Given z and leakage $\mathcal{L}_{\mathsf{S}}(\mathsf{DS})$ from the challenger, the simulator $\mathcal{S}$ returns an encrypted data structure* EDS *to $\mathcal{A}$. The adversary then* adaptively *chooses a polynomial number of operations $q_1, \ldots, q_m$. For all $i \in [m]$, the simulator receives a tuple $(\mathsf{DS}(q_i), \mathcal{L}_{\mathsf{Q}}(\mathsf{DS}, q_i))$ and returns a token tk_i to $\mathcal{A}$. Finally, $\mathcal{A}$ outputs a bit b that is output by the experiment.*

We say that Σ is adaptively $(\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{Q}})$-semantically secure if for all PPT *adversaries $\mathcal{A}$, there exists a* PPT *simulator $\mathcal{S}$ such that for all $z \in \{0,1\}^*$, the following expression is negligible in k:*

$$|\Pr[\,\mathbf{Real}_{\Sigma,\mathcal{A}}(k) = 1\,] - \Pr[\,\mathbf{Ideal}_{\Sigma,\mathcal{A},\mathcal{S}}(k) = 1\,]|$$

The security definition for *response-hiding* schemes can be derived from Definition 3.3 by giving the simulator $(\perp, \mathcal{L}_{\mathsf{Q}}(\mathsf{DS}, q_i))$ instead of $(\mathsf{DS}(q_i), \mathcal{L}_{\mathsf{Q}}(\mathsf{DS}, q_i))$.

Modeling leakage. Every STE scheme is associated with leakage which itself can be composed of multiple *leakage patterns*. The collection of all these leakage patterns forms the scheme's *leakage profile*. Leakage patterns are (families of) functions over the various spaces associated with the underlying data structure. For concreteness, we borrow the nomenclature introduced in [18] and recall some well-known leakage patterns that we make use of in this work. Here $\mathbb{D}$ and $\mathbb{Q}$ refer to the space of all possible data objects and the space of all possible queries for a given data type. In this work, we consider the following leakage patterns:

- the *query equality pattern* is the function family $\mathsf{qeq} = \{\mathsf{qeq}_{k,t}\}_{k,t\in\mathbb{N}}$ with $\mathsf{qeq}_{k,t} : \mathbb{D}_k \times \mathbb{Q}_k^t \to \{0,1\}^{t\times t}$ such that $\mathsf{qeq}_{k,t}(\mathsf{DS}, q_1, \ldots, q_t) = M$, where M is a binary $t \times t$ matrix such that $M[i,j] = 1$ if $q_i = q_j$ and $M[i,j] = 0$ if $q_i \neq q_j$. The query equality pattern is referred to as the search pattern in the SSE literature;
- the *response identity pattern* is the function family $\mathsf{rid} = \{\mathsf{rid}_{k,t}\}_{k,t\in\mathbb{N}}$ with $\mathsf{rid}_{k,t} : \mathbb{D}_k \times \mathbb{Q}_k^t \to [2^{[n]}]^t$ such that $\mathsf{rid}_{k,t}(\mathsf{DS}, q_1, \ldots, q_t) = (\mathsf{DS}[q_1], \ldots, \mathsf{DS}[q_t])$. The response identity pattern is referred to as the access pattern in the SSE literature;
- the *response length pattern* is the function family $\mathsf{rlen} = \{\mathsf{rlen}_{k,t}\}_{k,t\in\mathbb{N}}$ with $\mathsf{rlen}_{k,t} : \mathbb{D}_k \times \mathbb{Q}_k^t \to \mathbb{N}^t$ such that $\mathsf{rlen}_{k,t}(\mathsf{DS}, q_1, \ldots, q_t) = (|\mathsf{DS}[q_1]|, \ldots, |\mathsf{DS}[q_t]|)$;

4 The OPX Construction

We now describe our main construction, OPX, which is a response-hiding STE scheme for relational databases that supports conjunctive SQL queries. It uses a response-revealing multi-map encryption scheme Σ_{MM}, the adaptively-secure encrypted multi-map scheme $\Sigma^{\pi}_{\mathsf{MM}}$ by Cash et al. [6], a symmetric encryption scheme SKE, a pseudo-random function F, and a random oracle H. We describe the scheme in detail in Figures (1), (2), (3), and (4), and provide a high level description below.

Remark on notation. We note that the syntax of OPX matches Definition 3.2 but its queries q are query trees and its tokens tk are token trees; that is, trees where each node is a sub-token. We make this explicit here by referring to query trees as Q and to token trees as TK. Throughout, while processing a query tree, we denote by $\mathbf{R_{in}}$ the set of inputs to an operation/node and by $\mathbf{R_{out}}$ the set of outputs of that operation/node.

Let $\Sigma_{\mathsf{MM}} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Get})$ be a response-revealing multi-map encryption scheme, $\Sigma^{\pi}_{\mathsf{MM}} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Get})$ be the response-revealing multi-map encryption scheme in [6], $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric encryption scheme, $F : \{0,1\}^k \times \{0,1\}^\star \to \{0,1\}^m$ be a pseudo-random function, and $H : \{0,1\}^\star \to \{0,1\}^m$ be a random oracle. Consider the DB encryption scheme $\mathrm{OPX} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query}, \mathsf{Dec})$ defined as follows [a]:

- $\mathsf{Setup}(1^k, \mathsf{DB})$:
  1. initialize a set SET;
  2. initialize multi-maps MM_R, MM_C and MM_V;
  3. initialize multi-maps $(\mathsf{MM}_{\mathsf{att}})_{\mathsf{att} \in \mathsf{DB}^\intercal}$;
  4. initialize multi-maps $(\mathsf{MM}_{\mathsf{att},\mathsf{att}'})_{\mathsf{att},\mathsf{att}' \in \mathsf{DB}^\intercal}$ such that $\mathsf{dom}(\mathsf{att}) = \mathsf{dom}(\mathsf{att}')$;
  5. sample two keys $K_1, K_F \xleftarrow{\$} \{0,1\}^k$;
  6. for all $\mathbf{r} \in \mathsf{DB}$ set
  $$\mathsf{MM}_R\big[\chi(\mathbf{r})\big] := \Big(\mathsf{Enc}_{K_1}(r_1), \ldots \mathsf{Enc}_{K_1}(r_{\#\mathbf{r}}), \chi(\mathbf{r})\Big);$$
  7. compute $(K_R, \mathsf{EMM}_R) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Setup}\big(1^k, \mathsf{MM}_R\big)$;
  8. for all $\mathbf{c} \in \mathsf{DB}^\intercal$, set
  $$\mathsf{MM}_C\big[\chi(\mathbf{c})\big] := \Big(\mathsf{Enc}_{K_1}(c_1), \ldots \mathsf{Enc}_{K_1}(c_{\#\mathbf{c}}), \chi(\mathbf{c})\Big);$$
  9. compute $(K_C, \mathsf{EMM}_C) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Setup}\big(1^k, \mathsf{MM}_C\big)$;
  10. for all $\mathbf{c} \in \mathsf{DB}^\intercal$,
      (a) for all $v \in \mathbf{c}$ and $\mathbf{r} \in \mathsf{DB}_{\mathbf{c}=v}$,
          i. compute $\mathsf{mtk}_{\mathbf{r}} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\Big(K_R, \chi(\mathbf{r})\Big)$,
      (b) set
      $$\mathsf{MM}_V\Big[\big\langle v, \chi(\mathbf{c})\big\rangle\Big] := \Big(\mathsf{mtk}_{\mathbf{r}}\Big)_{\mathbf{r} \in \mathsf{DB}_{\mathbf{c}=v}};$$
  11. compute $(K_V, \mathsf{EMM}_V) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Setup}\big(1^k, \mathsf{MM}_V\big)$;
  12. for all $\mathbf{c} \in \mathsf{DB}^\intercal$,
      (a) for all $\mathbf{c}' \in \mathsf{DB}^\intercal$ such that $\mathsf{dom}(\mathsf{att}(\mathbf{c}')) = \mathsf{dom}(\mathsf{att}(\mathbf{c}))$,
          i. initialize an empty tuple $\mathbf{t}$;
          ii. for all rows $\mathbf{r}_i$ and $\mathbf{r}_j$ in $\mathbf{c}$ and $\mathbf{c}'$, such that $\mathbf{c}[i] = \mathbf{c}'[j]$,
              A. compute $\mathsf{mtk}_i \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\Big(K_R, \chi(\mathbf{r}_i)\Big)$;
              B. compute $\mathsf{mtk}_j \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\Big(K_R, \chi(\mathbf{r}_j)\Big)$;
              C. add $(\mathsf{mtk}_i, \mathsf{mtk}_j)$ to $\mathbf{t}$;
          iii. set
          $$\mathsf{MM}_{\mathbf{c}}\Big[\Big\langle \chi(\mathbf{c}), \chi(\mathbf{c}')\Big\rangle\Big] := \mathbf{t};$$
      (b) compute $(K_{\mathbf{c}}, \mathsf{EMM}_{\mathbf{c}}) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Setup}\big(1^k, \mathsf{MM}_{\mathbf{c}}\big)$;

[a] Note that we omit the description of Dec since it simply decrypts every cell of R.

Figure 1: The OPX scheme (Part 1).

- $\mathsf{Setup}(1^k, \mathrm{Q})$:

13. for all $\mathbf{c} \in \mathsf{DB}^{\mathsf{T}}$,
    (a) for all $v \in \mathbf{c}$,
        i. compute $K_v \leftarrow F_{K_F}(\chi(\mathbf{c}) \| v)$;
        ii. set for all $\mathbf{r} \in \mathsf{DB}_{\mathbf{c}=v}$,
        $$\mathsf{SET} := \mathsf{SET} \bigcup \Big\{ H\big(K_v \| \mathsf{rtk}\big) \Big\},$$
        where $\mathsf{rtk} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}(K_R, \chi(\mathbf{r}))$;
14. for all $\mathbf{c} \in \mathsf{DB}^{\mathsf{T}}$,
    (a) for all $\mathbf{c}' \in \mathsf{DB}^{\mathsf{T}}$ such that $\mathsf{dom}(\mathsf{att}(\mathbf{c}')) = \mathsf{dom}(\mathsf{att}(\mathbf{c}))$,
        i. initialize an empty tuple $\mathbf{t}$;
        ii. for all $\mathbf{r}_i, \mathbf{r}_j \in [m]$ such that $\mathbf{c}[i] = \mathbf{c}'[j]$,
            A. add $\big(\mathsf{rtk}_i, \mathsf{rtk}_j\big)$ to $\mathbf{t}$ where
            $$\mathsf{rtk}_i \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}(K_R, \chi(\mathbf{r}_i)) \quad \text{and} \quad \mathsf{rtk}_j \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}(K_R, \chi(\mathbf{r}_i j).$$
        iii. for all rtk s.t. $(\mathsf{rtk}, \cdot) \in \mathbf{t}$, set
        $$\mathsf{MM}_{\mathbf{c},\mathbf{c}'}\Big[\mathsf{rtk}\Big] := \Big(\mathsf{rtk}'\Big)_{(\mathsf{rtk},\mathsf{rtk}') \in \mathbf{t}}$$
    (b) compute $(K_{\mathbf{c},\mathbf{c}'}, \mathsf{EMM}_{\mathbf{c},\mathbf{c}'}) \leftarrow \Sigma^{\pi}_{\mathsf{MM}}.\mathsf{Setup}\big(1^k, \mathsf{MM}_{\mathbf{c},\mathbf{c}'}\big)$;
15. output $K = (K_1, K_R, K_C, K_V, \{K_{\mathbf{c}}\}_{\mathbf{c} \in \mathsf{DB}^{\mathsf{T}}}, K_F, \{K_{\mathbf{c},\mathbf{c}'}\}_{\mathbf{c},\mathbf{c}' \in \mathsf{DB}^{\mathsf{T}}})$ and $\mathsf{EDB} = (\mathsf{EMM}_R, \mathsf{EMM}_C, \mathsf{EMM}_V, (\mathsf{EMM}_{\mathbf{c},\mathbf{c}'})_{\mathbf{c},\mathbf{c}' \in \mathsf{DB}^{\mathsf{T}}}, \mathsf{SET}, (\mathsf{EMM}_{\mathbf{c}})_{\mathbf{c} \in \mathsf{DB}^{\mathsf{T}}})$.

Figure 2: The OPX scheme (Part 2).

- $\mathsf{Token}(K, \mathrm{Q})$:
  1. initialize a token tree TK with empty nodes and with the same structure as Q;
  2. for every node N accessed in a post-traversal order in Q ,
     (a) if $N \equiv \sigma_{\mathsf{att}=a}(\mathbf{T})$ then set TK_N to
     $$\mathsf{stk} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\Big(K_V, \langle a, \chi(\mathsf{att})\Big);$$
     (b) if $N \equiv \sigma_{\mathsf{att}=a}(\mathbf{R_{in}})$ then set TK_N to $(\mathsf{mtk}, \mathtt{pos})$ where
     $$\mathsf{mtk} \leftarrow F_{K_F}\Big(\chi(\mathsf{att}) \| a\Big)$$
     and $\mathtt{pos}$ denotes the position of att in $\mathbf{R_{in}}$.
     (c) if $N \equiv \mathbf{T}_1 \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{T}_2$ then set TK_N to $(\mathsf{jtk}, \mathtt{pos})$ where
     $$\mathsf{jtk} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\Big(K_{\mathsf{att}_1}, \Big\langle \chi(\mathsf{att}_1), \chi(\mathsf{att}_2)\Big\rangle\Big),$$
     and $\mathtt{pos}$ is the position of attribute att_1 in $\mathbf{R_{in}}$.
     (d) if $N \equiv \mathbf{T} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{R_{in}}$ then set the corresponding node in TK to $(\mathsf{etk}, \mathtt{pos}_1, \mathtt{pos}_2)$ where
     $$\mathsf{etk} := K_{\mathsf{att}_1,\mathsf{att}_2};$$
     and $\mathtt{pos}_1$, $\mathtt{pos}_2$ are the positions of the attributes att_1, att_2 in $\mathbf{R_{in}}$, respectively.
     (e) if $N \equiv \mathbf{R}_{\mathbf{in}}^{(l)} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{R}_{\mathbf{in}}^{(r)}$ then set TK_N to $(\mathtt{pos}_1, \mathtt{pos}_2)$ where $\mathtt{pos}_1$ and $\mathtt{pos}_2$ are the column positions of att_1 and att_2 in $\mathbf{R}_{\mathbf{in}}^{(l)}$ and $\mathbf{R}_{\mathbf{in}}^{(r)}$, respectively.
     (f) if $N \equiv \pi_{\mathsf{att}}(\mathbf{T})$ then set TK_N to ptk where
     $$\mathsf{ptk} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\Big(K_C, \chi(\mathsf{att}_i)\Big).$$
     (g) if $N \equiv \pi_{\mathsf{att}_1,\cdots,\mathsf{att}_z}(\mathbf{R_{in}})$ then set TK_N to
     $$\Big(\mathtt{pos}_1, \cdots, \mathtt{pos}_z\Big),$$
     where $\mathtt{pos}_i$ is the column position of att_i in $\mathbf{R_{in}}$.
     (h) if $N \equiv [a]$ then set TK_N to $[\mathsf{Enc}_{K_1}(a)]$.
     (i) if $N \equiv \times$ then set TK_N to $\times$.
  3. output TK.

Figure 3: The OPX scheme (Part 3).

- $\mathsf{Query}(\mathsf{EDB}, \mathsf{tk})$:
  1. parse EDB as $(\mathsf{EMM}_R, \mathsf{EMM}_C, \mathsf{EMM}_V, (\mathsf{EMM}_{\mathbf{c},\mathbf{c}'})_{\mathbf{c},\mathbf{c}' \in \mathsf{DB}^\intercal}, \mathsf{SET}, (\mathsf{EMM}_{\mathbf{c}})_{\mathbf{c} \in \mathsf{DB}^\intercal})$.
  2. for every node N accessed in a post-traversal order in TK,
     - if $N \equiv \mathsf{stk}$, it computes
     $$(\mathsf{rtk}_1, \cdots, \mathsf{rtk}_s) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Query}\Big(\mathsf{stk}, \mathsf{EMM}_V\Big),$$
     and sets $\mathbf{R_{out}} := (\mathsf{rtk}_1, \cdots, \mathsf{rtk}_s)$;
     - if $N \equiv (\mathsf{mtk}, \mathtt{pos})$, then for all rtk in $\mathbf{R_{in}}$ in the column at position $\mathtt{pos}$, if
     $$H(\mathsf{mtk} \| \mathsf{rtk}) \notin \mathsf{SET}$$
     then it removes the row from $\mathbf{R_{in}}$. Finally, it sets $\mathbf{R_{out}} := \mathbf{R_{in}}$;
     - if $N \equiv (\mathsf{jtk}, \mathtt{pos})$, then it computes
     $$\Big((\mathsf{rtk}_1, \mathsf{rtk}'_1), \ldots, (\mathsf{rtk}_s, \mathsf{rtk}'_s)\Big) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Query}(\mathsf{jtk}, \mathsf{EMM}_{\mathtt{pos}}),$$
     and sets
     $$\mathbf{R_{out}} := \Big((\mathsf{rtk}_i, \mathsf{rtk}'_i)\Big)_{i \in [s]};$$
     - if $N \equiv (\mathsf{etk}, \mathtt{pos}_1, \mathtt{pos}_2)$, then for each row $\mathbf{r}$ in $\mathbf{R_{in}}$, it computes $\mathsf{ltk} \leftarrow \Sigma^{\pi}_{\mathsf{MM}}.\mathsf{Token}(\mathsf{etk}, \mathsf{rtk})$, and
     $$(\mathsf{rtk}_1, \cdots, \mathsf{rtk}_s) \leftarrow \Sigma^{\pi}_{\mathsf{MM}}.\mathsf{Query}(\mathsf{ltk}, \mathsf{EMM}_{\mathtt{pos}_1, \mathtt{pos}_2}),$$
     where $\mathsf{rtk} = \mathbf{r}[\mathsf{att}_{\mathtt{pos}_2}]$, and appends the new rows $\Big(\mathsf{rtk}_i\Big)_{i \in [s]} \times \mathbf{r}$ to $\mathbf{R_{out}}$;
     - if $N \equiv (\mathtt{pos}_1, \mathtt{pos}_2)$, then it sets
     $$\mathbf{R_{out}} := \mathbf{R}^{(l)}_{\mathbf{in}} \bowtie_{\mathtt{pos}_1 = \mathtt{pos}_2} \mathbf{R}^{(r)}_{\mathbf{in}},$$
     where $\mathbf{R}^{(l)}_{\mathbf{in}}$ and $\mathbf{R}^{(r)}_{\mathbf{in}}$ are the left and right input respectively;
     - if $N \equiv \mathsf{ptk}$ then it computes
     $$(\mathrm{ct}_1, \cdots, \mathrm{ct}_s) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Query}\Big(\mathsf{ptk}, \mathsf{EMM}_C\Big)$$
     and sets $\mathbf{R_{out}} := (\mathrm{ct}_1, \cdots, \mathrm{ct}_s)$;
     - if $N \equiv (\mathtt{pos}_1, \cdots, \mathtt{pos}_z)$, then it computes $\mathbf{R_{out}} := \pi_{\mathtt{pos}_1, \cdots, \mathtt{pos}_z}(\mathbf{R_{in}})$;
     - if $N \equiv \times$ then it computes
     $$\mathbf{R_{out}} := \mathbf{R}^{(l)}_{\mathbf{in}} \times \mathbf{R}^{(r)}_{\mathbf{in}};$$
  3. it replaces each cell rtk in $\mathbf{R}^{\mathsf{root}}_{\mathbf{out}}$ by $\mathrm{ct} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Query}(\mathsf{rtk}, \mathsf{EMM}_R)$;
  4. output $\mathbf{R}^{\mathsf{root}}_{\mathbf{out}}$.

Figure 4: The OPX scheme (Part 4).

Setup. The Setup algorithm takes as input a database $\mathsf{DB} = (\mathbf{T}_1, \cdots, \mathbf{T}_n)$ and a security parameter k. It first samples a key $K_1 \xleftarrow{\$} \{0,1\}^k$, and then initializes a multi-map MM_R such that for all rows $\mathbf{r} \in \mathsf{DB}$, it sets

$$\mathsf{MM}_R\Big[\chi(\mathbf{r})\Big] := \Big(\mathsf{Enc}_{K_1}(r_1), \cdots, \mathsf{Enc}_{K_1}(r_{\#\mathbf{r}}), \chi(\mathbf{r})\Big),$$

It then computes

$$(K_R, \mathsf{EMM}_R) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Setup}\Big(1^k, \mathsf{MM}_R\Big).$$

It initializes a multi-map MM_C such that for all columns $\mathbf{c} \in \mathsf{DB}^\intercal$, it sets

$$\mathsf{MM}_C\Big[\chi(\mathbf{c})\Big] := \Big(\mathsf{Enc}_{K_1}(c_1), \cdots, \mathsf{Enc}_{K_1}(c_{\#\mathbf{c}}), \chi(\mathbf{c})\Big),$$

It then computes

$$(K_C, \mathsf{EMM}_C) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Setup}\Big(1^k, \mathsf{MM}_C\Big).$$

It initializes a multi-map MM_V, and for each $\mathbf{c} \in \mathsf{DB}^\intercal$, all $v \in \mathbf{c}$ and $\mathbf{r} \in \mathsf{DB}_{\mathbf{c}=v}$, it computes

$$\mathsf{mtk}_{\mathbf{r}} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\Big(K_R, \chi(\mathbf{r})\Big),$$

and sets

$$\mathsf{MM}_V\Big[\langle v, \chi(\mathbf{c})\rangle\Big] := \Big(\mathsf{mtk}_{\mathbf{r}}\Big)_{\mathbf{r} \in \mathsf{DB}_{\mathbf{c}=v}}.$$

It then computes

$$(K_V, \mathsf{EMM}_V) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Setup}(1^k, \mathsf{MM}_V).$$

It initializes a set of multi-maps $\{\mathsf{MM}_{\mathbf{c}}\}_{\mathbf{c} \in \mathsf{DB}^\intercal}$. For all columns $\mathbf{c}, \mathbf{c}' \in \mathsf{DB}^\intercal$ that have the same domain such that $\mathsf{dom}(\mathsf{att}(\mathbf{c})) = \mathsf{dom}(\mathsf{att}(\mathbf{c}'))$, it initiates an empty tuple $\mathbf{t}$ that it populates as follows. For all rows $\mathbf{r}_i$ and $\mathbf{r}_j$ in column $\mathbf{c}$ and $\mathbf{c}'$, respectively, that verify

$$\mathbf{c}[i] = \mathbf{c}'[j],$$

it inserts $(\mathsf{rtk}_i, \mathsf{rtk}_j)$ in $\mathbf{t}$ where

$$\mathsf{rtk}_i \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}(K_R, \chi(\mathbf{r}_i))$$

and

$$\mathsf{rtk}_j \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}(K_R, \chi(\mathbf{r}_i j)),$$

and sets

$$\mathsf{MM}_{\mathbf{c}}\Big[\langle \chi(\mathbf{c}), \chi(\mathbf{c}')\rangle\Big] := \mathbf{t}.$$

It then computes, for all $\mathbf{c} \in \mathsf{DB}^\intercal$,

$$(K_{\mathbf{c}}, \mathsf{EMM}_{\mathbf{c}}) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Setup}\Big(1^k, \mathsf{MM}_{\mathbf{c}}\Big).$$

It initializes a set SET and computes for each column $\mathbf{c} \in \mathsf{DB}^{\mathsf{T}}$, and for all $v \in \mathbf{c}$, a key K_v such that

$$K_v \leftarrow F_{K_F}(\chi(\mathbf{c}) \| v),$$

where $K_F \xleftarrow{\$} \{0,1\}^k$. Then for all rows $\mathbf{r}$ in $\mathsf{DB}_{\mathbf{c}=v}$, it sets

$$\mathsf{SET} := \mathsf{SET} \bigcup \Big\{ H(K_v \| \mathsf{rtk}) \Big\},$$

where $\mathsf{rtk} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}(K_R, \chi(\mathbf{r}))$. It then initializes a set of multi-maps $\{\mathsf{MM}_{\mathsf{att},\mathsf{att}'}\}$ for $\mathsf{att}, \mathsf{att}' \in \mathbb{S}(\mathsf{DB})$ and $\mathsf{dom}(\mathsf{att}) = \mathsf{dom}(\mathsf{att}')$. For all columns $\mathbf{c}, \mathbf{c}' \in \mathsf{DB}^{\mathsf{T}}$ that have the same domain, it initiates an empty tuple $\mathbf{t}$ that it populates as follows. For all rows $\mathbf{r}_i$ and $\mathbf{r}_j$ in column $\mathbf{c}$ and $\mathbf{c}'$, respectively, that verify

$$\mathbf{c}[i] = \mathbf{c}'[j],$$

it inserts $(\mathsf{rtk}_i, \mathsf{rtk}_j)$ in $\mathbf{t}$ where

$$\mathsf{rtk}_i \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}(K_R, \chi(\mathbf{r}_i))$$

and

$$\mathsf{rtk}_j \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}(K_R, \chi(\mathbf{r}_i j)).$$

Then for all rtk such that $(\mathsf{rtk}, \cdot) \in \mathbf{t}$, it sets

$$\mathsf{MM}_{\mathbf{c},\mathbf{c}'}\Big[\mathsf{rtk}\Big] := \Big(\mathsf{rtk}'\Big)_{(\mathsf{rtk},\mathsf{rtk}') \in \mathbf{t}}$$

then computes

$$(K_{\mathbf{c},\mathbf{c}'}, \mathsf{EMM}_{\mathbf{c},\mathbf{c}'}) \leftarrow \Sigma^{\pi}_{\mathsf{MM}}.\mathsf{Setup}\Big(1^k, \mathsf{MM}_{\mathbf{c},\mathbf{c}'}\Big).$$

Finally, it outputs a key $K = (K_1, K_R, K_C, K_V, \{K_{\mathbf{c}}\}_{\mathbf{c} \in \mathsf{DB}^{\mathsf{T}}}, K_F, \{K_{\mathbf{c},\mathbf{c}'}\}_{\mathbf{c},\mathbf{c}' \in \mathsf{DB}^{\mathsf{T}}})$ and $\mathsf{EDB} = (\mathsf{EMM}_R, \mathsf{EMM}_C, \mathsf{EMM}_V, (\mathsf{EMM}_{\mathbf{c},\mathbf{c}'})_{\mathbf{c},\mathbf{c}' \in \mathsf{DB}^{\mathsf{T}}}, \mathsf{SET}, (\mathsf{EMM}_{\mathbf{c}})_{\mathbf{c} \in \mathsf{DB}^{\mathsf{T}}})$.

Token. The Token algorithm takes as input a key K and a query tree Q and outputs a token tree TK. The token tree is a copy of Q and first initialized with empty nodes. The algorithm performs a post-order traversal of the query tree and, for every visited node N, does the following:

- **(leaf select)** if N is a leaf node of form $\sigma_{\mathsf{att}=a}(\mathbf{T})$ then set the corresponding node in TK to

$$\mathsf{stk} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\Big(K_V, \langle a, \chi(\mathsf{att}) \Big);$$

- **(internal constant select):** if N is an internal node of form $\sigma_{\mathsf{att}=a}(\mathbf{R_{in}})$ then set the corresponding node in TK to $(\mathsf{mtk}, \mathtt{pos})$ where

$$\mathsf{mtk} \leftarrow F_{K_F}\Big(\chi(\mathsf{att}) \| a\Big)$$

  and $\mathtt{pos}$ denotes the position of att in $\mathbf{R_{in}}$.

- **(leaf join):** if N is a leaf node of form $\mathbf{T}_1 \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{T}_2$ then set the corresponding node in TK to $(\mathsf{jtk}, \mathtt{pos})$ where

$$\mathsf{jtk} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\Big(K_{\mathsf{att}_1}, \Big\langle \chi(\mathsf{att}_1), \chi(\mathsf{att}_2) \Big\rangle\Big),$$

  and $\mathtt{pos}$ denotes the positions of att_1 in $\mathbf{R_{in}}$.

- **(internal join):** if N is an internal node of form $\mathbf{T} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{R_{in}}$, then set the corresponding node in TK to $(\mathsf{etk}, \mathtt{pos}_1, \mathtt{pos}_2)$ where

$$\mathsf{etk} := K_{\mathsf{att}_1,\mathsf{att}_2};$$

  and $\mathtt{pos}_1$, $\mathtt{pos}_2$ denote the positions of att_1 and att_2 in $\mathbf{R_{in}}$, respectively.

- **(intermediate internal join):** if N is an internal node of form $\mathbf{R}_{\mathbf{in}}^{(l)} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{R}_{\mathbf{in}}^{(r)}$ then set the corresponding node in TK to $(\mathtt{pos}_1, \mathtt{pos}_2)$ where $\mathtt{pos}_1$ and $\mathtt{pos}_2$ are the column positions of att_1 and att_2 in $\mathbf{R}_{\mathbf{in}}^{(l)}$ and $\mathbf{R}_{\mathbf{in}}^{(r)}$, respectively.

- **(leaf projection):** if N is a leaf node of form $\pi_{\mathsf{att}}(\mathbf{T})$ then set the corresponding node to ptk where

$$\mathsf{ptk} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\Big(K_C, \chi(\mathsf{att}_i)\Big)$$

- **(internal projection):** if N is an internal node of form $\pi_{\mathsf{att}_1,\cdots,\mathsf{att}_z}(\mathbf{R_{in}})$ then set the corresponding node to

$$\Big(\mathtt{pos}_1, \cdots, \mathtt{pos}_z\Big),$$

  where $\mathtt{pos}_i$ is the column position of att_i in $\mathbf{R_{in}}$.

- **(leaf scalars):** if N is a node of form $[a]$ then set the corresponding node to $[\mathsf{Enc}_{K_1}(a)]$.

- **(cross product):** if N is a node of form $\times$ then keep it with no changes.

Query. The algorithm takes as input the encrypted database EDB and the token tree TK. It performs a post-order traversal of tk and, for each visited node N, does the following:

- **(leaf select):** if N has form stk, it computes

$$(\mathsf{rtk}_1, \cdots, \mathsf{rtk}_s) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Query}\Big(\mathsf{stk}, \mathsf{EMM}_V\Big)$$

  and sets $\mathbf{R_{out}} := (\mathsf{rtk}_1, \cdots, \mathsf{rtk}_s)$.

- **(internal constant select):** if N has form $(\mathsf{mtk}, \mathtt{pos})$, then for all rtk in $\mathbf{R_{in}}$ in the column at position $\mathtt{pos}$, if

$$H(\mathsf{mtk} \| \mathsf{rtk}) \notin \mathsf{SET}$$

  then it removes the row from $\mathbf{R_{in}}$. Finally, it sets $\mathbf{R_{out}} := \mathbf{R_{in}}$.

- **(leaf join):** if N has form $(\mathsf{jtk}, \mathtt{pos})$, then it computes

$$\Big((\mathsf{rtk}_1, \mathsf{rtk}'_1), \ldots, (\mathsf{rtk}_s, \mathsf{rtk}'_s)\Big) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Query}(\mathsf{jtk}, \mathsf{EMM}_{\mathtt{pos}}),$$

  and sets

$$\mathbf{R_{out}} := \Big((\mathsf{rtk}_i, \mathsf{rtk}'_i)\Big)_{i\in[s]}$$

- **(internal join):** if N has form $(\mathsf{etk}, \mathtt{pos}_1, \mathtt{pos}_2)$, then for each row $\mathbf{r}$ in $\mathbf{R_{in}}$, it computes $\mathsf{ltk} \leftarrow \Sigma^{\pi}_{\mathsf{MM}}.\mathsf{Token}(\mathsf{etk}, \mathsf{rtk})$, and

$$(\mathsf{rtk}_1, \cdots, \mathsf{rtk}_s) \leftarrow \Sigma^{\pi}_{\mathsf{MM}}.\mathsf{Query}(\mathsf{ltk}, \mathsf{EMM}_{\mathtt{pos}_1,\mathtt{pos}_2}),$$

  where $\mathsf{rtk} = \mathbf{r}[\mathsf{att}_{\mathtt{pos}_2}]$, and appends the new rows

$$\Big(\mathsf{rtk}_i\Big)_{i\in[s]} \times \mathbf{r}$$

  to $\mathbf{R_{out}}$

- **(intermediate internal join):** if N has form $(\mathtt{pos}_1, \mathtt{pos}_2)$, then it sets

$$\mathbf{R_{out}} := \mathbf{R}^{(l)}_{\mathbf{in}} \bowtie_{\mathtt{pos}_1=\mathtt{pos}_2} \mathbf{R}^{(r)}_{\mathbf{in}}$$

- **(leaf projection):** if N is a leaf node of form ptk then it computes

$$(\mathrm{ct}_1, \cdots, \mathrm{ct}_s) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Query}\Big(\mathsf{ptk}, \mathsf{EMM}_C\Big)$$

  and sets $\mathbf{R_{out}} := (\mathrm{ct}_1, \cdots, \mathrm{ct}_s)$.

- **(internal projection):** if N is an internal node of form $(\mathtt{pos}_1, \cdots, \mathtt{pos}_z)$, then it computes

$$\mathbf{R_{out}} := \pi_{\mathtt{pos}_1,\cdots,\mathtt{pos}_z}(\mathbf{R_{in}})$$

- **(cross product):** if N is a node of form $\times$ then it computes

$$\mathbf{R_{out}} := \mathbf{R}^{(l)}_{\mathbf{in}} \times \mathbf{R}^{(r)}_{\mathbf{in}},$$

  where $\mathbf{R}^{(l)}_{\mathbf{in}}$ and $\mathbf{R}^{(r)}_{\mathbf{in}}$ are the left and right input respectively.

Now, it replaces each cell rtk in $\mathbf{R}^{\mathsf{root}}_{\mathbf{out}}$ by

$$\mathrm{ct} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Query}(\mathsf{rtk}, \mathsf{EMM}_R).$$

4.1 Efficiency

We now turn to analyzing the search and storage efficiency of our construction.

Query complexity. Given a potentially optimized query tree Q of an SPC query, we show that the search complexity of OPX is asymptotically optimal.

Theorem 4.1. *If Σ_{mm} is optimal, then the time and space complexity of the* Query *algorithm presented in Section (4) is optimal.*

The proof of the theorem is in Appendix A.

Storage complexity. The storage complexity of OPX is similar to that of SPX asymptotically, but is larger concretely. This is because OPX needs two additional encrypted structures: a collection of encrypted multi-maps $(\mathsf{EMM}_{\mathbf{c},\mathbf{c}'})_{\mathbf{c},\mathbf{c}'\in\mathsf{DB}^\intercal}$ and an encrypted set SET.

For a database $\mathsf{DB} = (\mathbf{T}_1, \ldots, \mathbf{T}_n)$, OPX produces three encrypted multi-maps EMM_R, EMM_C, EMM_V, two collections of encrypted multi-maps $(\mathsf{EMM}_{\mathbf{c},\mathbf{c}'})_{\mathbf{c},\mathbf{c}'\in\mathsf{DB}^\intercal}$ and $(\mathsf{EMM}_{\mathbf{c}})_{\mathbf{c}\in\mathsf{DB}^\intercal}$, and a set structure SET. For ease of exposition, we assume that each table is composed of m rows. Also, note that standard multi-map encryption schemes [11, 9, 19, 7, 6] produce encrypted structures with storage overhead that is linear in the sum of the tuple sizes. Using such a scheme as the underlying multi-map encryption scheme, we have that EMM_R and EMM_C are $O(\sum_{\mathbf{r}\in\mathsf{DB}} \#\mathbf{r})$ and $O(\sum_{\mathbf{c}\in\mathsf{DB}^\intercal} \#\mathbf{c})$, respectively, since the former maps the coordinates of each row in DB to their (encrypted) row and the latter maps the coordinates of very column to their (encrypted) columns. Since EMM_V maps each cell in DB to tokens for the rows that contain the same value, it requires $O(\sum_{\mathbf{c}\in\mathsf{DB}^\intercal} \sum_{v\in\mathbf{c}} \#\mathsf{DB}_{\mathsf{att}(\mathbf{c})=v})$ storage. Similarly, SET contains the pseudo- random evaluation of the coordinate of all rows in the database and therefore requires $O(\sum_{\mathbf{c}\in\mathsf{DB}^\intercal} \sum_{v\in\mathbf{c}} \#\mathsf{DB}_{\mathsf{att}(\mathbf{c})=v})$. For each $\mathbf{c} \in \mathsf{DB}^\intercal$, an encrypted multi-map $\mathsf{EMM}_{\mathbf{c}}$ maps each pair of form $(\mathbf{c}, \mathbf{c}')$ such that $\mathsf{dom}(\mathsf{att}(\mathbf{c})) = \mathsf{dom}(\mathsf{att}(\mathbf{c}'))$ to a tuple of tokens for rows in $\mathsf{DB}_{\mathsf{att}(\mathbf{c})=\mathsf{att}(\mathbf{c}')}$. As such, the collection $(\mathsf{EMM}_{\mathbf{c}})_{\mathbf{c}\in\mathsf{DB}^\intercal}$ has size

$$O\Big(\sum_{\mathbf{c}\in\mathsf{DB}^\intercal} \sum_{\mathbf{c}':\mathsf{dom}(\mathsf{att}(\mathbf{c}'))=\mathsf{dom}(\mathsf{att}(\mathbf{c}))} \#\mathsf{DB}_{\mathsf{att}(\mathbf{c})=\mathsf{att}(\mathbf{c}')}\Big).$$

Similarly, for all $\mathbf{c}, \mathbf{c}' \in \mathsf{DB}^\intercal$, an encrypted multi-map $\mathsf{EMM}_{\mathsf{att},\mathsf{att}'}$ maps the coordinate of each row $\mathbf{r}$ in the column att to all the coordinates of rows $\mathbf{r}'$ in att' that have the same value such that $\mathbf{r}[\mathsf{att}] = \mathbf{r}'[\mathsf{att}']$. The size of $(\mathsf{EMM}_{\mathbf{c},\mathbf{c}'})_{\mathbf{c},\mathbf{c}'\in\mathsf{DB}^\intercal}$ is exactly the same as the earlier collection.

Note that the expression above will vary greatly depending on the number of columns in DB that have the same domain. In the worst case, all columns will have a common domain and the expression will be a sum of $O((\sum_i \|\mathbf{T}_i\|_c)^2)$ terms of the form $\#\mathsf{DB}_{\mathsf{att}(\mathbf{c})=\mathsf{att}(\mathbf{c}')}$. In the best case, none of the columns will share a domain and both collections will be empty. In practice, however, we expect there to be some relatively small number of columns with common domains. In Appendix (C), we provide a concrete example of the storage overhead of an encrypted database.

5 Security and Leakage of OPX

We show that OPX is adaptively-semantically secure with respect to a well-specified leakage profile. Similar to the leakage profile SPX [16], the profile of OPX is composed of a "black-box component" in the sense that it comes from the underlying STE schemes, and a "non-black-box component" that comes from OPX directly. In this section, we will first describe and prove this leakage profile in a black-box manner, i.e., without assuming any specific instantiation of the underlying STE schemes

except for $\Sigma^{\pi}_{\mathsf{MM}}$ which is a concrete response-revealing multi-map encryption scheme by Cash et al. [6]. Then, as a second step, we consider two instantiations with different concrete leakage profiles that illustrate the impact on the overall leakage profile of OPX. In particular, depending on the chosen concrete instantiation, we will show that the resulting leakage profile can be significantly different.

5.1 Black-Box Leakage Profile

In the following, we describe the setup and query leakage of OPX without any assumption on how the underlying data structure encryption schemes work.

Setup leakage. The setup leakage captures what a persistent adversary learns by only observing the encrypted structure and before observing any query execution. The setup leakage of OPX is equal to the setup leakage of SPX along with the setup leakage of Σ_{DX} and the number of cells of all tables in the database such that[2]

$$\mathcal{L}^{\mathsf{opx}}_{\mathsf{S}}(\mathsf{DB}) = \Big(\left(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}(\mathsf{MM}_{\mathbf{c}})\right)_{\mathbf{c}\in\mathsf{DB}^{\intercal}}, \mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}\left(\mathsf{MM}_R\right), \mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}\left(\mathsf{MM}_C\right), \\ \mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}\left(\mathsf{MM}_V\right), \left(\mathcal{L}^{\pi}_{\mathsf{S}}(\mathsf{MM}_{\mathbf{c},\mathbf{c}'})\right)_{\mathbf{c},\mathbf{c}'\in\mathsf{DB}^{\intercal}}, n \cdot \sum_{i=1}^{n} \|T_i\|_c \Big),$$

where $\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}$, $\mathcal{L}^{\pi}_{\mathsf{S}}$, n and $\|\mathbf{T}_i\|$ are the setup leakage of Σ_{MM}, the setup leakage of $\Sigma^{\pi}_{\mathsf{MM}}$ which is equal to the sum of all tuple sizes in a given multi-map, the number of tables, and the number of columns in the ith table, respectively.

Query leakage. The query leakage captures what a persistent adversary learns when it observes the token and query execution. The query leakage of OPX is represented as a *leakage tree* that has the same form as of the query tree Q. In particular, the query leakage, denoted here Λ, starts empty and is then populated in a recursive manner as the query execution goes through in a post-order traversal of the nodes of Q. In particular, for every node N visited in Q, the query leakage is constructed as follows.

Cross product. If the node $N \equiv \mathsf{xnode}$, then this is is a *cross product* pattern which is defined as

$$\mathcal{X}(\mathsf{xnode}) = \begin{cases} (\mathtt{scalar}, |a|) & \text{if } \mathsf{xnode} \equiv [a]; \\ (\mathtt{cross}, \perp) & \text{if } \mathsf{xnode} \equiv \times; \end{cases}$$

This pattern captures what the server learns when it executes a scalar node or a cross product node. The query leakage is now equal to

$$\Lambda := \Lambda \bigcup \Big\{ \mathcal{X}(\mathsf{xnode}) \Big\}.$$

[2]Note that this information will be revealed to the adversary through the size of the set structure SET

Projection. If $N \equiv \mathsf{pnode}$, then this is a *projection pattern* which is defined as

$$\mathcal{P}(\mathsf{pnode}) = \begin{cases} \left(\mathtt{leaf}, \mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}\left(\mathsf{MM}_C, \chi(\mathsf{att})\right)\right) & \text{if } \mathsf{pnode} \equiv \pi_{\mathsf{att}}(\mathbf{T}); \\ \left(\mathtt{in}, f(\mathsf{att}_1), \cdots, f(\mathsf{att}_z)\right) & \text{if } \mathsf{pnode} \equiv \pi_{\mathsf{att}_1,\cdots,\mathsf{att}_z}(\mathbf{R_{in}}); \end{cases}$$

where $f \xleftarrow{\$} \left\{\{0,1\}^* \to \{0,1\}^{\log(\rho)}\right\}$ is a uniformly sampled function and ρ is the total number of attributes in DB. The projection pattern captures the leakage produced when the server executes a projection node, whether it is a leaf or an internal node. If the node pnode in Q is a *leaf projection*, then $\mathcal{P}(\mathsf{pnode})$ captures the leakage produced when the server queries EMM_C to retrieve the encrypted content of the column att. More precisely, $\mathcal{P}(\mathsf{pnode})$ reveals the Σ_{MM} query leakage on the coordinates of the projected attribute. Otherwise, if the node pnode is an *internal projection* in Q, then $\mathcal{P}(\mathsf{pnode})$ reveals the position of the attributes, $\mathsf{att}_1, \cdots, \mathsf{att}_z$, in $\mathbf{R_{in}}$ – the intermediary result table given as input to pnode. The query leakage is now equal to

$$\Lambda := \Lambda \bigcup \left\{\mathcal{P}(\mathsf{pnode})\right\}.$$

Selection. If $N \equiv \mathsf{snode}$, then this is a *selection pattern* which is defined as

$$\mathcal{Z}(\mathsf{snode}) = \begin{cases} \left(\mathtt{leaf}, \mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}\left(\mathsf{MM}_V, \left\langle a, \chi(\mathsf{att})\right\rangle\right), \left(\mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}(\mathsf{MM}_R, \chi(\mathbf{r})\right)_{\mathbf{r} \in \mathsf{DB}_{\mathsf{att}=a}}\right) & \text{if } \mathsf{snode} \equiv \sigma_{\mathsf{att}=a}(\mathbf{T}); \\ \left(\mathtt{in}, f(\mathsf{att}), g(a \| \mathsf{att}), \left(\mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}(\mathsf{MM}_R, \chi(\mathbf{r})\right)_{\chi(\mathbf{r}) \in \mathbf{R_{in}} \wedge \mathbf{r}[\mathsf{att}]=a}\right) & \text{if } \mathsf{snode} \equiv \sigma_{\mathsf{att}=a}(\mathbf{R_{in}}); \end{cases}$$

where $g \xleftarrow{\$} \left\{\{0,1\}^* \to \{0,1\}^{\log(\gamma)}\right\}$ is a uniformly sampled function, and γ is the sum of distinct values in every column in the entire database. The selection pattern captures the leakage produced when the server executes a selection node, whether it is a leaf or an internal node. If the node snode is a *leaf selection* node, then $\mathcal{Z}(\mathsf{snode})$ captures the leakage produced when the server queries EMM_V to retrieve some row tokens. More precisely, $\mathcal{Z}(\mathsf{snode})$ reveals the Σ_{MM} query leakage on the coordinates of the attribute att and the constant a. It also reveals the Σ_{MM} query leakage on all coordinates of rows whose cell values at attribute att match the constant a. Otherwise, if the node snode is an *internal selection* node, then $\mathcal{Z}(\mathsf{snode})$ captures the leakage produced when the server removes all row tokens in the intermediate result set $\mathbf{R_{in}}$ that do not belong to the set structure SET. In particular, $\mathcal{Z}(\mathsf{snode})$ reveals the Σ_{MM} query leakage on all coordinates of rows $\mathbf{r}$ in $\mathbf{R_{in}}$ that match the constant a at the attribute att. The query leakage is now equal to

$$\Lambda := \Lambda \bigcup \left\{\mathcal{Z}(\mathsf{snode})\right\}.$$

Join. If $N \equiv \mathsf{jnode}$, then this is a *join pattern* which is defined as follows. If jnode has form $\mathbf{T}_1 \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{T}_2$ then,

$$\begin{aligned} \mathcal{J}(\mathsf{jnode}) = & \Big(\mathtt{leaf}, f(\mathsf{att}_1), \mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}\Big(\mathsf{MM}_{\mathsf{att}_1}, \Big\langle \chi(\mathsf{att}_1), \chi(\mathsf{att}_2)\Big\rangle\Big), \\ & \Big\{\mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}(\mathsf{MM}_R, \chi(\mathbf{r}_1), \mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}(\mathsf{MM}_R, \chi(\mathbf{r}_2)\Big\}_{(\mathbf{r}_1,\mathbf{r}_2) \in \mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2}}\Big), \end{aligned}$$

In this case, $\mathcal{J}(\mathsf{jnode})$ captures the leakage produced when the server retrieves some $\mathsf{EMM}_{\mathsf{att}_1}$ which it in turn queries to retrieve row tokens. More precisely, it reveals if and when $\mathsf{EMM}_{\mathsf{att}_1}$ has been accessed in the past. In addition, it reveals the query leakage of Σ_{MM} on the coordinates of att_1 and att_2 and, for every pair of rows $(\mathbf{r}_1, \mathbf{r}_2)$ in $\mathsf{DB}_{\mathsf{att}_{i,1}=\mathsf{att}_{i,2}}$, it reveals the Σ_{MM} query leakage on their coordinates. If jnode has form $\mathbf{T} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{R_{in}}$ then,

$$\mathcal{J}(\mathsf{jnode}) = \left(\mathtt{in}, \langle f(\mathsf{att}_1), f(\mathsf{att}_2)\rangle, \left(\mathcal{L}^{\pi}_{\mathsf{Q}}\left(\mathsf{MM}_{\mathsf{att}_1,\mathsf{att}_2}, \chi(\mathbf{r})\right)\right)_{\chi(\mathbf{r})\in\mathbf{R_{in}}[\mathsf{att}_2]}, \left\{\mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}(\mathsf{MM}_R, \chi(\mathbf{r}_1)\right\}_{\substack{(\mathbf{r}_1,\mathbf{r}_2)\in\mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2} \\ \wedge\chi(\mathbf{r}_2)\in\mathbf{R_{in}}[\mathsf{att}_2]}}\right),$$

where $\mathbf{R_{in}}[\mathsf{att}]$ denotes the cell values in $\mathbf{R_{in}}$ at attribute att. In this case, $\mathcal{J}(\mathsf{jnode})$ captures the leakage produced when the server retrieves $\mathsf{EMM}_{\mathsf{att}_1,\mathsf{att}_2}$ which it in turn queries to retrieve row tokens. More precisely, it reveals if and when $\mathsf{EMM}_{\mathsf{att}_1,\mathsf{att}_2}$ has been accessed in the past. In addition, it reveals the query leakage of $\Sigma^{\pi}_{\mathsf{MM}}$ on the coordinates of rows $\mathbf{r}$ that belong to $\mathbf{R_{in}}[\mathsf{att}]$ and, for every pair of rows $(\mathbf{r}_1, \mathbf{r}_2)$ in $\mathsf{DB}_{\mathsf{att}_{i,1}=\mathsf{att}_{i,2}}$ such that $\chi(\mathbf{r}_2) \in \mathbf{R_{in}}[\mathsf{att}_2]$, it reveals the Σ_{MM} query leakage on their row coordinates. In particular, the concrete query leakage of $\Sigma^{\pi}_{\mathsf{MM}}$ reveals if and when the same query is evaluated (search pattern) as well as the response identifiers (access pattern). If jnode has form $\mathbf{R}^{(l)}_{\mathbf{in}} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{R}^{(r)}_{\mathbf{in}}$ then,

$$\mathcal{J}(\mathsf{jnode}) = \left(\mathtt{inter}, f(\mathsf{att}_1), f(\mathsf{att}_2)\right),$$

In this case, $\mathcal{J}(\mathsf{jnode})$ captures the leakage produced when the server removes all the rows in $\mathbf{R}^{(l)}_{\mathbf{in}} \times \mathbf{R}^{(r)}_{\mathbf{in}}$ to only keep those which have the same cell value at both attributes att_1 and att_2. The query leakage is now equal to

$$\Lambda := \Lambda \bigcup \left\{\mathcal{J}(\mathsf{jnode})\right\}.$$

Finally, it sets

$$\mathcal{L}^{\mathsf{opx}}_{\mathsf{Q}}(\mathsf{DB}, \mathrm{Q}) := \Lambda.$$

5.2 Security of OPX

We now prove that OPX is adaptively semantically-secure with respect to the leakage profile described in the previous sub-section.

Theorem 5.1. *If F is a pseudo-random function, SKE is RCPA secure, $\Sigma^{\pi}_{\mathsf{MM}}$ is adaptively $(\mathcal{L}^{\pi}_{\mathsf{S}}, \mathcal{L}^{\pi}_{\mathsf{Q}})$-secure, and Σ_{MM} is adaptively $(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}, \mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}})$-secure, then* OPX *is adaptively $(\mathcal{L}^{\mathsf{opx}}_{\mathsf{S}}, \mathcal{L}^{\mathsf{opx}}_{\mathsf{Q}})$-secure in the random oracle model.*

The proof of Theorem 5.1 is in Appendix (B).

5.3 Concrete Leakage Profile

In this section, we are interested in the leakage profile of OPX when the underlying data structure encryption schemes are instantiated with specific constructions and a well-specified concrete leakage profile. Note that in this section, we make the additional assumption that $\Sigma^{\pi}_{\mathsf{MM}}$ from [6] is replaced with an almost leakage free multi-map encryption scheme. However, this scheme needs to verify some key-equivocation property which is the case for the volume hiding schemes like PBS [18], VLH or AVLH [17] if built using the adaptively-secure $\Sigma^{\pi}_{\mathsf{MM}}$ scheme as the underlying multi-map encryption scheme.

(Almost) Leakage-free data structure encryption schemes. We make the assumption that the underlying response-revealing multi-map encryption scheme Σ_{mm} is almost-leakage free in that it leaks the response length pattern, known as the volume pattern, and the response identity pattern such that

$$\mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}(\mathsf{MM}, q) = \Big(\mathsf{rlen}, \mathsf{rid}\Big).$$

To instantiate such a scheme, one can use oblivious RAM (ORAM) simulation techniques [14] in a black-box fashion, or more customized/advanced schemes such as the oblivious tree structures (OTS) [**?**] or the TWORAM construction [13] with a careful parametrization of the block-sizes, or the AZL construction based on the piggy-backing scheme PBS [18]. These constructions however incur an additional overhead, and some of them, work under new trade-offs. Note that if a construction is response-hiding, then it may require one round of interaction to reveal the response. Note that the leakage profile of OPX can be further improved by using *completely* leakage-free data structures that can also hide the volume pattern, but we defer the details to the full version of this work.

In the following, we describe the concrete leakage profile of OPX when instantiated with a (almost) leakage-free data structure encryption. Specifically, when the node is an xnode, the revealed cross-product pattern remains the same. If the node is a pnode, then the projection pattern added to Λ is now equal to

$$\mathcal{P}(\mathsf{pnode}) = \begin{cases} \Big(\mathtt{leaf}, \big(|c_j|\big)_{j\in[\#\mathbf{c}[\mathsf{att}]]}, \mathrm{AccP}(\mathsf{att})\Big) & \text{if } \mathsf{pnode}_i \equiv \pi_{\mathsf{att}}(\mathbf{T}); \\ \Big(\mathtt{in}, f(\mathsf{att}_1), \cdots, f(\mathsf{att}_z)\Big) & \text{if } \mathsf{pnode}_i \equiv \pi_{\mathsf{att}_1,\cdots,\mathsf{att}_z}(\mathbf{R_{in}}). \end{cases}$$

where $\mathrm{AccP}(\mathsf{att})$ denotes if and when the attribute att has been accessed before.

If the node is an snode, then the revealed selection pattern added to Λ is now equal to

$$\mathcal{Z}(\mathsf{snode}) = \begin{cases} \Big(\mathtt{leaf}, \Big\{|\mathbf{r}|, \mathrm{AccP}(\mathbf{r})\Big\}_{\mathbf{r}\in\mathsf{DB}_{\mathsf{att}=a}}\Big) & \text{if } \mathsf{snode} \equiv \sigma_{\mathsf{att}=a}(\mathbf{T}); \\ \Big(\mathtt{in}, g(a\|\mathsf{att}), \Big\{|\mathbf{r}|, \mathrm{AccP}(\mathbf{r})\Big\}_{\chi(\mathbf{r})\in\mathbf{R_{in}}\wedge\mathbf{r}[\mathsf{att}]=a}\Big) & \text{if } \mathsf{snode} \equiv \sigma_{\mathsf{att}=a}(\mathbf{R_{in}}); \end{cases}$$

If the node is a jnode, then the revealed join pattern added to Λ is now equal to

$$\mathcal{J}(\mathsf{jnode}) = \Big(\mathtt{leaf}, f(\mathsf{att}_1), \Big\{|\mathbf{r}_1|, \mathrm{AccP}(\mathbf{r}_1), |\mathbf{r}_2|, \mathrm{AccP}(\mathbf{r}_2)\Big\}_{(\mathbf{r}_1,\mathbf{r}_2)\in\mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2}}\Big),$$

if jnode has form $\mathbf{T}_1 \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{T}_2$ and,

$$\mathcal{J}(\mathsf{jnode}) = \Big(\mathtt{in}, \langle f(\mathsf{att}_1), f(\mathsf{att}_2)\rangle, \Big\{|\mathbf{r}_1|, \mathrm{AccP}(\mathbf{r}_1)\Big\}_{\substack{(\mathbf{r}_1,\mathbf{r}_2)\in\mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2} \\ \wedge\chi(r_2)\in\mathbf{R_{in}}[\mathsf{att}_2]}}\Big),$$

if jnode has form $\mathbf{T} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{R_{in}}$ and,

$$\mathcal{J}(\mathsf{jnode}) = \Big(\mathtt{inter}, f(\mathsf{att}_1), f(\mathsf{att}_2)\Big),$$

if jnode has form $\mathbf{R}_{\mathbf{in}}^{(l)} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{R}_{\mathbf{in}}^{(r)}$.

Variant. Note that the leakage profile of OPX can be further improved with some slight modifications to the main OPX construction. In particular, if the underlying response-revealing multi-map is replaced with a response-hiding scheme, then the access pattern, $\mathrm{AccP}(\mathbf{r})$, of an accessed row, $\mathbf{r}$, can be completely hidden. Note that even the response length of the intermediary results will not be disclosed as the underlying scheme is leakage-free as per our assumption. For example, in the case of a *leaf select node*, the output will now be a set of row coordinates, instead of row tokens. And in order to proceed to the next node, the client and server need to interact to first decrypt the row coordinate and execute the next operation. Note that this approach will not incur any additional query overhead to what is added by using leakage-free schemes; however it will add additional interaction between the client and the server. The concrete leakage profile of this modified scheme will be the type of nodes composing the query plan, i.e., whether the node is a join, select, or a cross-product node. We defer the details of this variant to the full version of this work.

Efficiency. We have shown in Theorem 4.1 that both the OPX query algorithm and the equivalent plaintext execution on the same query tree Q have exactly the same query complexity if the underlying multi-map and dictionary encryption schemes are instantiated using standard techniques [11, 9, 19, 7, 6]. However, in the (almost) leakage-free setting, the query complexity of OPX is higher for the simple reason that the cost of querying a leakage-free data structure encryption scheme is higher than the one of querying a standard (optimal) scheme. More precisely, at any step where the client and server execute a Σ_{mm} query protocol, then the query complexity will be higher depending on the executed node. We describe below this impact in more details.

- **(case 1):** If the node is a *leaf selection node* of the form $\sigma_{\mathsf{att}=a}(\mathbf{T})$, then the overhead is equal to

$$O\Big(\#\mathsf{DB}_{\mathsf{att}=a} \cdot \log\Big(m \cdot \sum_{i=1}^{n} \|\mathbf{T}_i\|_c\Big)\Big).$$

  where m is the maximum number of cells in a table; instead of $O(m)$ – the query complexity of a plaintext execution on the same node.

- **(case 2):** If the node is a *leaf join node* of the form $\mathbf{T}_1 \bowtie_{\mathsf{att}_1=\mathsf{att}_2} (\mathbf{T}_2)$, then the overhead is equal to

$$O\Big(\#\mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2} \cdot \log\Big(\sum_{\mathsf{att}\in\mathbb{S}(\mathsf{DB})} \sum_{\substack{\mathsf{att}'\in\mathbb{S}(\mathsf{DB}) \\ \mathsf{dom}(\mathsf{att})=\mathsf{dom}(\mathsf{att}')}} \#\mathsf{DB}_{\mathsf{att}=\mathsf{att}'}\Big)\Big),$$

  where $\mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2}$ is the tuple composed of all joined pairs between columns att_1 and att_2; instead of $O(\#\mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2})$ – the query complexity of a plaintext execution on the same node.

- **(case 3):** If the node is an *internal join node* of the form $\mathbf{T} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} (\mathbf{R_{in}})$, then the overhead is equal to

$$O\Big(\sum_{\chi(\mathbf{r})\in\mathbf{R_{in}}[\mathsf{att}_2]} \Big(\#\mathsf{DB}_{\mathsf{att}_1=\mathrm{Val}_{\mathsf{att}_2}(\mathbf{r})} \cdot \log\Big(\sum_{\mathsf{att}\in\mathbb{S}(\mathsf{DB})} \sum_{\substack{\mathsf{att}'\in\mathbb{S}(\mathsf{DB}) \\ \mathsf{dom}(\mathsf{att})=\mathsf{dom}(\mathsf{att}')}} \#\mathsf{DB}_{\mathsf{att}=\mathsf{att}'}\Big)\Big)\Big),$$

  where $\mathrm{Val}_{\mathsf{att}_2}(\mathbf{r})$ is the cell value of row $\mathbf{r}$ at attribute att_2; instead of $O(\sum_{\chi(\mathbf{r})\in\mathbf{R_{in}}[\mathsf{att}_2]}(\#\mathsf{DB}_{\mathsf{att}_1=\mathrm{Val}_{\mathsf{att}_2}(\mathbf{r})}))$– the query complexity of a plaintext execution on the same node.

- **(case 5):** If the node is a *leaf projection node* of the form $\pi_{\mathsf{att}}(\mathbf{T})$, then the overhead is equal to

$$O\Big(m \cdot \log\Big(m \cdot \sum_{i=1}^{n} \|\mathbf{T}_i\|_c\Big)\Big),$$

  where m is the maximum number of cells in the table.

- **(case 5):** if the node is a *scalar node*, a *cross-product node*, an *intermediate internal join*, an *internal projection node*, or an *internal selection node*, then the query complexity is similar to a plaintext execution as no multi-map or dictionary query executions are required in the process.

Note that using (almost) leakage-free data structures to instantiate OPX does not incur any asymptotical storage overhead.

Standard data structure encryption schemes. In this section, we describe the leakage profile of OPX if instantiated with standard data structure encryption schemes [11, 9, 19, 7, 6]. By *standard*, we refer to a class of well-studied data structure encryption schemes that reveal the *response identity pattern* (rid), and the *query equality pattern* (qeq), known as the access pattern and the search pattern in the SSE literature, respectively. The search pattern reveals if and when a query is repeated while the access pattern reveals the identities of the responses. The concrete leakage profile of OPX when instantiated with these standard data structures is the same as the one detailed in the abstract section except that we replace the black box notation $\mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}$ with rid and qeq on the same inputs. Below, we give a high level intuition on what each pattern will disclose.

Select pattern. Independently of the type of the selection node, then an adversary can learn the number of rows containing the same value as well as the frequency with which a particular row has been accessed, and also the size of that row. If many queries have been performed on the same table and the same column, then the adversary can build a frequency histogram of that specific column's contents. Now depending on the composition of the query tree, an adversary can build a more detailed histogram if more *internal* selection are performed on the same attribute.

Join pattern. Among all patterns, the join pattern leaks the most. The adversary learns the number of rows that have equal values in a given pair of attributes. In addition, it learns the frequency with which these rows have been accessed in the past, eventually following the execution of a different type of nodes such as a projection or a selection. Similar to the selection pattern, the adversary can build therefore a histogram summarizing the frequency of apparition of rows that it gets richer with more operations down the query tree. If the join node is internal, then the adversary learns a bit more information as for every row, it knows exactly the rows in a different attribute that have the same value. The adversary can help the adversary for example to trace back to the leaf join leakage information it collected to identify the exact rows that have the same values. This is also true in general for all the information the adversary collects from different nodes as long as the operations are correlated. Finally, if the node is an *intermediate internal node*, then the execution of such a node leads to the propagation of the frequency information cross different attributes.

Projection pattern. This pattern simply discloses the number of rows in a specific attributes (size of the column) along with the frequency with which these rows have been accessed.

Note that we dismissed a discussion on the cross-product pattern as it is self-explanatory and does not involve querying any data structure encryption scheme.

Efficiency. With respect to efficiency, we have shown in Theorem 4.1 that the execution of the OPX query algorithm and its plaintext counterpart have exactly the same asymptotics.

References

[1] R. Ada Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 85–100, 2011.

[2] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *ACM SIGMOD International Conference on Management of Data*, pages 563–574, 2004.

[3] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with cipherbase. In *CIDR*, 2013.

[4] M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and efficiently searchable encryption. In A. Menezes, editor, *Advances in Cryptology – CRYPTO '07*, Lecture Notes in Computer Science, pages 535–552. Springer, 2007.

[5] A. Boldyreva, N. Chenette, Y. Lee, and A. O'neill. Order-preserving symmetric encryption. In *Advances in Cryptology - EUROCRYPT 2009*, pages 224–241, 2009.

[6] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *Network and Distributed System Security Symposium (NDSS '14)*, 2014.

[7] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology - CRYPTO '13*. Springer, 2013.

[8] A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *ACM Symposium on Theory of Computing (STOC '77)*, pages 77–90. ACM, 1977.

[9] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *Advances in Cryptology - ASIACRYPT '10*, volume 6477 of *Lecture Notes in Computer Science*, pages 577–594. Springer, 2010.

[10] M. Chase and S. Kamara. Structured encryption and controlled disclosure. Technical Report 2011/010.pdf, IACR Cryptology ePrint Archive, 2010.

[11] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *ACM Conference on Computer and Communications Security (CCS '06)*, pages 79–88. ACM, 2006.

[12] B. A. Fisch, B. Vo, F. Krell, A. Kumarasubramanian, V. Kolesnikov, T. Malkin, and S. M. Bellovin. Malicious-client security in blind seer: a scalable private dbms. In *IEEE Symposium on Security and Privacy*, pages 395–410. IEEE, 2015.

[13] S. Garg, P. Mohassel, and C. Papamanthou. TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption. In *Advances in Cryptology - CRYPTO 2016*, pages 563–592, 2016.

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

[15] H. Hacigümücs, B. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 216–227, 2002.

[16] S. Kamara and T. Moataz. SQL on Structurally-Encrypted Data. In *Asiacrypt*, 2018.

[17] S. Kamara and T. Moataz. Computationally volume-hiding structured encryption. In *Advances in Cryptology - Eurocrypt' 19*, 2019.

[18] S. Kamara, T. Moataz, and O. Ohrimenko. Structured encryption and leakae suppression. In *Advances in Cryptology - CRYPTO '18*, 2018.

[19] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *ACM Conference on Computer and Communications Security (CCS '12)*. ACM Press, 2012.

[20] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 2008.

[21] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '15, pages 644–655. ACM, 2015.

[22] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S.-G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind seer: A scalable private dbms. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 359–374. IEEE, 2014.

[23] SAP Software Solutions. SEEED. https://www.sics.se/sites/default/files/pub/andreasschaad.pdf.

[24] D. Song, D. Wagner, and A. Perrig. Practical techniques for searching on encrypted data. In *IEEE Symposium on Research in Security and Privacy*, pages 44–55. IEEE Computer Society, 2000.

A Proof of Theorem 4.1

Theorem 4.1. *If the query algorithm of Σ_{mm} is optimal, then the time and space complexity of the* Query *algorithm presented in Section (4) is optimal.*

Proof. A query tree Q can be composed of four different types of nodes: (1) a cross-product node xnode, (2) a projection node pnode, (3) a selection node snode, and a (4) a join node jnode. We will show that for each type of nodes, the search and space complexity on plaintext text relational database is asymptotically equal to the search and space complexity required by the Query algorithm of OPX. We assume in this proof that the plaintext database has indices to speed-up lookup operations on every attribute.

- **(case 1):** if the node is a cross-product node, then the output of the node, xnode, in a plaintext database given a left and a right input $\mathbf{R}_{\mathbf{in}}^{(l)}$ and $\mathbf{R}_{\mathbf{in}}^{(r)}$, respectively, is equal to

$$\mathbf{R}_{\mathbf{out}} = \mathbf{R}_{\mathbf{in}}^{(l)} \times \mathbf{R}_{\mathbf{in}}^{(r)},$$

  which is the exact same operation performed by the Query algorithm of OPX when the node is a cross-product node.

- **(case 2):** if the node is a projection node, then there are two possible cases. If the node pnode has form $\pi_{\mathsf{att}}(\mathbf{T})$, a leaf projection node, then a plaintext database will require a work linear in $O(m)$ to fetch all the cells of the attribute att and where m is the number of cell in the column. On the other hand, OPX performs a Query operation on EMM_C to fetch the corresponding encrypted cells. Assuming that Σ_{MM} has an optimal search complexity, the amount of work is also linear in $O(m)$.[3]

  The second case is when the projection node has form $\pi_{\mathsf{att}}(\mathbf{R}_{\mathbf{in}})$, an interior projection node. In this case, a plaintext database will simply select the corresponding columns from the input $\mathbf{R}_{\mathbf{in}}$ which has search complexity equal to $O(\#\mathbf{R}_{\mathbf{in}}[\mathsf{att}])$ which is the number of cells of the attribute att in $\mathbf{R}_{\mathbf{in}}$. In the Query algorithm of OPX, the exact same operation is performed and therefore, the same complexity is required.

- **(case 3):** if the node is a selection node, there there are three possible cases. If the node snode has form $\sigma_{\mathsf{att}=a}(\mathbf{T})$, a leaf selection node, then a plaintext database will require a work linear in $O(\#\mathsf{DB}_{\mathsf{att}=a})$ which is the number of cells in the attribute att equal to a. On the other hand, OPX performs a Query operation on EMM_C to fetch the corresponding cells in $\mathsf{DB}_{\mathsf{att}=a}$. Assuming that Σ_{MM} has an optimal search complexity, then the amount of work is equal to $O(\#\mathsf{DB}_{\mathsf{att}=a})$.

  The second case is when the selection node has form $\sigma_{\mathsf{att}=a}(\mathbf{R}_{\mathbf{in}})$, an interior selection node. In this case, a plaintext database has to go linearly over the entire column att in $\mathbf{R}_{\mathbf{in}}$ to only output the rows in $\mathbf{R}_{\mathbf{in}}$ with the cell at the attribute att equal to the constant a. That is, the search complexity is equal to $O(\mathbf{R}_{\mathbf{in}}[\mathsf{att}])$. On the other hand, OPX tests for each row in $\mathbf{R}_{\mathbf{in}}[\mathsf{att}]$ whether it exists in SET. Assuming that test membership in SET is optimal, then the search complexity is equal to $O(\mathbf{R}_{\mathbf{in}}[\mathsf{att}])$

[3] Note that we are not accounting for the security parameter in our computation and only focusing on the number of cells.

The third case is when the selection node has form $\sigma_{\mathsf{att}_1=\mathsf{att}_2}(\mathbf{R_{in}})$, an interior variable select node. In this case, a plaintext will simply remove any row in $\mathbf{R_{in}}$ such that the cell values are not equal. This has search complexity equal to $O(\#\mathbf{R_{in}}[\mathsf{att}_1])$. On the other hand, OPX similarly removes all rows that have equal equal cell value at both columns att_1 and att_2. Clearly, the plaintext and encrypted operations have the same search and space complexity.

- **(case 4):** if the node is a join node, then there are two possible cases. If the node jnode has form $\mathbf{T}_1 \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{T}_2$, then a plaintext database would at least require $O(\#\mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2})$ which is the result of the join operation on the columns att_1 and att_2. On the other hand, OPX queries $\mathsf{EMM}_{\mathsf{att}_1}$ to fetch the join result. Assuming that Σ_{MM} has an optimal search complexity, then the search complexity is equal to $O\#(\mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2})$.

  The second case occurs when the join node has form $\mathbf{T} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{R_{in}}$, an interior join node. In this case, a plaintext database has to go over every cell at the attribute att_2 and checks if there are any rows in table $\mathbf{T}$ at attribute att_1 that are equal to the value in the selected cell. The search complexity is equal to

$$O\Big(\max(\#\mathbf{R_{in}}[\mathsf{att}_2], \#\mathbf{R_{out}}[\mathsf{att}_2])\Big),$$

  which is itself equal to the maximum value of either (1) the number of cells in $\mathbf{R_{in}}[\mathsf{att}]$ or (2) the size of joinable rows which is equal to $\#\mathbf{R_{out}}[\mathsf{att}_2]$ (or equivalently to $\#\mathbf{R_{out}}[\mathsf{att}_1]$). On the other hand, OPX queries $\mathsf{EMM}_{\mathsf{att}_1,\mathsf{att}_2}$ to fetch the joinable result. Similar to the plaintext scenario, OPX will for each row token in $\mathbf{R_{in}}[\mathsf{att}_2]$ fetch the joinable rows, if any, from $\mathsf{EMM}_{\mathsf{att}_1,\mathsf{att}_2}$. Since $\Sigma^{\pi}_{\mathsf{MM}}$ has an optimal search complexity, then the search complexity is equal to $O(\max(\#\mathbf{R_{in}}[\mathsf{att}_2], \#\mathbf{R_{out}}[\mathsf{att}_2]))$ as the same operation is performed.

Finally, OPX will query EMM_R to retrieve all the encrypted rows corresponding to the rows tokens in $\mathbf{R}^{\mathsf{root}}_{\mathbf{in}}$. Assuming that Σ_{mm} has an optimal search complexity, then this step will require $O(\#\mathbf{R}^{\mathsf{root}}_{\mathbf{in}})$. Note that this operation would add exactly the same complexity as the sum of the output size of the child nodes, and therefore would not have an impact on the final asymptotic result.

To sum up, we have shown that whatever the type of the node, both the plaintext and OPX query algorithm executions require the same space and search complexities.

■

B Proof of Theorem 5.1

Theorem 5.1. *If F is a pseudo-random function, SKE is RCPA secure, $\Sigma^{\pi}_{\mathsf{MM}}$ is adaptively $(\mathcal{L}^{\pi}_{\mathsf{S}}, \mathcal{L}^{\pi}_{\mathsf{Q}})$-secure, and Σ_{MM} is adaptively $(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}, \mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}})$-secure, then* OPX *is adaptively $(\mathcal{L}^{\mathsf{opx}}_{\mathsf{S}}, \mathcal{L}^{\mathsf{opx}}_{\mathsf{Q}})$-secure in the random oracle model.*

Proof. Let $\mathcal{S}_{\mathsf{MM}}$ and $\mathcal{S}^{\pi}_{\mathsf{MM}}$ be the simulators guaranteed to exist by the adaptive security of Σ_{MM} and $\Sigma^{\pi}_{\mathsf{MM}}$ and consider the OPX simulator $\mathcal{S}$ that works as follows. Given $\mathcal{L}^{\mathsf{opx}}_{\mathsf{S}}(\mathsf{DB})$, $\mathcal{S}$ simulates EDB by computing $\mathsf{EMM}_R \leftarrow \mathcal{S}_{\mathsf{MM}}(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}(\mathsf{MM}_R))$, $\mathsf{EMM}_C \leftarrow \mathcal{S}_{\mathsf{MM}}(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}(\mathsf{MM}_C))$, $\mathsf{EMM}_V \leftarrow \mathcal{S}_{\mathsf{MM}}(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}(\mathsf{MM}_V))$, for all $\mathbf{c} \in \mathsf{DB}^{\mathsf{T}}$, $\mathsf{EMM}_{\mathbf{c}} \leftarrow \mathcal{S}_{\mathsf{MM}}(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}(\mathsf{MM}_{\mathbf{c}}))$, and for all $\mathbf{c}, \mathbf{c}' \in \mathsf{DB}^{\mathsf{T}}$, $\mathsf{EMM}_{\mathbf{c},\mathbf{c}'} \leftarrow$

$\mathcal{S}^{\pi}_{\mathsf{MM}}(\mathcal{L}^{\pi}_{\mathsf{S}}(\mathsf{MM}_{\mathbf{c},\mathbf{c}'}))$. Given (n,ρ), it instantiates an empty set SET, and inserts $r_{i,j} \xleftarrow{\$} \{0,1\}^k$ in SET for $i \in [n]$ and $j \in [\rho]$. $\mathcal{S}$ outputs

$$\mathsf{EDB} = (\mathsf{EMM}_R, \mathsf{EMM}_C, \mathsf{EMM}_V, (\mathsf{EMM}_{\mathbf{c}})_{\mathbf{c}\in\mathsf{DB}^{\intercal}}, \mathsf{SET}, (\mathsf{EMM}_{\mathbf{c},\mathbf{c}'})_{\mathbf{c},\mathbf{c}'\in\mathsf{DB}^{\intercal}}).$$

Recall that OPX is response-hiding so $\mathcal{S}$ receives $(\perp, \mathcal{L}^{\mathsf{opx}}_{\mathsf{Q}}(\mathsf{DB},\mathrm{Q}))$ as input in the $\mathbf{Ideal}_{\mathrm{SPX},\mathcal{A},\mathcal{S}}(k)$ experiment. Given this input, $\mathcal{S}$ parses $\mathcal{L}^{\mathsf{opx}}_{\mathsf{Q}}(\mathsf{DB},\mathrm{Q})$ as a leakage tree. It then instantiates a token tree TK with the same structure. It samples uniformly at random a key $K_1 \xleftarrow{\$} \{0,1\}^k$, and creates s set $\mathsf{SET}^{\star}$ such that $\mathsf{SET}^{\star} := \mathsf{SET}$. For each node N, retrieved in a post-order traversal from the leakage tree, it simulates the corresponding node in the token tree TK as follows.

- **(Cross product).** If N has form $(\mathtt{scalar}, |a|)$ then it sets TK_N to $[\mathsf{Enc}_{K_1}(\mathbf{0}^{|a|})]$. Otherwise if N has form $(\mathtt{cross}, \perp)$, then it sets TK_N to $\times$.
- **(Projection).** If N has form $\Big(\mathtt{leaf}, \mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}\Big(\mathsf{MM}_C, \chi(\mathsf{att})\Big)\Big)$ then it sets

  $$\mathsf{TK}_N \leftarrow \mathcal{S}_{\mathsf{MM}}\Big(\mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}\Big(\mathsf{MM}_C, \chi(\mathsf{att})\Big)\Big),$$

  If N has form $(\mathtt{in}, f(\mathsf{att}_1), \cdots, f(\mathsf{att}_z))$, then it sets TK_N to $(f(\mathsf{att}_1), \cdots, f(\mathsf{att}_z))$.
- **(Selection case-1).** If N has form

  $$\Big(\mathtt{leaf}, \mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}\Big(\mathsf{MM}_V, \Big\langle a, \chi(\mathsf{att})\Big\rangle\Big), \Big(\mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}(\mathsf{MM}_R, \chi(\mathbf{r})\Big)_{\mathbf{r}\in\mathsf{DB}_{\mathsf{att}=a}}\Big)$$

  then it first sets for all $\mathbf{r} \in \mathsf{DB}_{\mathsf{att}=a}$,

  $$\mathsf{mtk}_{\mathbf{r}} \leftarrow \mathcal{S}_{\mathsf{MM}}\Big(\mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}(\mathsf{MM}_R, \chi(\mathbf{r}))\Big),$$

  then it sets,

  $$\mathsf{TK}_N \leftarrow \mathcal{S}_{\mathsf{MM}}\Big((\mathsf{mtk}_{\mathbf{r}})_{\mathbf{r}\in\mathsf{DB}_{\mathsf{att}=a}}, \mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}\Big(\mathsf{MM}_V, \Big\langle a, \chi(\mathsf{att})\Big\rangle\Big)\Big).$$
- **(Selection case-2).** If N has form

  $$\Big(\mathtt{in}, f(\mathsf{att}), g(a\|\mathsf{att}), \Big(\mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}(\mathsf{MM}_R, \chi(\mathbf{r})\Big)_{\chi(\mathbf{r})\in\mathbf{R_{in}}\wedge\mathbf{r}[\mathsf{att}]=a}\Big)$$

  then if $g(a\|\mathsf{att})$ has never been revealed before,
  - for all $\mathbf{r} \in \mathsf{DB}$ such that $\chi(\mathbf{r}) \in \mathbf{R_{in}}$ and $\mathbf{r}[\mathsf{att}] = a$, it sets

    $$\mathsf{mtk}_{\mathbf{r}} \leftarrow \mathcal{S}_{\mathsf{MM}}\Big(\mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}(\mathsf{MM}_R, \chi(\mathbf{r}))\Big)$$

  - it samples a key $K_{g(a\|\mathsf{att})} \xleftarrow{\$} \{0,1\}^k$;

– for each $\mathbf{r} \in \mathsf{DB}$ such that $\chi(\mathbf{r}) \in \mathbf{R_{in}}$ and $\mathbf{r}[\mathsf{att}] = a$, it picks and removes uniformly at random a value r in $\mathsf{SET}^\star$ and sets

$$H(K_{g(a\|\mathsf{att})}\|\mathsf{mtk}_{\mathbf{r}}) := r;$$

– it sets

$$\mathsf{TK}_N \leftarrow (K_{g(a\|\mathsf{att})}, f(\mathsf{att})).$$

Otherwise, if $g(a\|\mathsf{att})$ has been revealed before then,

– for all $\mathbf{r} \in \mathsf{DB}$ such that $\chi(\mathbf{r}) \in \mathbf{R_{in}}$ and $\mathbf{r}[\mathsf{att}] = a$, it sets

$$\mathsf{mtk}_{\mathbf{r}} \leftarrow \mathcal{S}_{\mathsf{MM}}\Big(\mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}(\mathsf{MM}_R, \chi(\mathbf{r}))\Big)$$

– for all $\mathbf{r} \in \mathsf{DB}$ such that $\chi(\mathbf{r}) \in \mathbf{R_{in}}$ and $\mathbf{r}[\mathsf{att}] = a$, if $H(K_{g(a\|\mathsf{att})}\|\mathsf{mtk}_{\mathbf{r}})$ has not been set yet, then it picks and removes uniformly at random a value $r \in \mathsf{SET}^\star$ and sets

$$H(K_{g(a\|\mathsf{att})}\|\mathsf{mtk}_{\mathbf{r}}) := r;$$

– it sets

$$\mathsf{TK}_N \leftarrow (K_{g(a\|\mathsf{att})}, f(\mathsf{att})).$$

- **(Join case-1).** If N has form

$$\Big(\mathtt{leaf}, f(\mathsf{att}_1), \mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}\Big(\mathsf{MM}_{\mathsf{att}_1}, \Big\langle \chi(\mathsf{att}_1), \chi(\mathsf{att}_2)\Big\rangle\Big),$$
$$\Big\{\mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}(\mathsf{MM}_R, \chi(\mathbf{r}_1), \mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}(\mathsf{MM}_R, \chi(\mathbf{r}_2)\Big\}_{(\mathbf{r}_1,\mathbf{r}_2)\in\mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2}}\Big),$$

then it sets for all $(\mathbf{r}_1, \mathbf{r}_2) \in \mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2}$,

$$\mathsf{mtk}_1 \leftarrow \mathcal{S}_{\mathsf{MM}}\Big(\mathcal{L}^{\mathsf{MM}}_{\mathsf{Q}}\Big(\mathsf{MM}_R, \chi(\mathbf{r}_1)\Big)\Big)$$

and

$$\mathsf{mtk}_2 \leftarrow \mathcal{S}_{\mathsf{MM}}\Big(\mathcal{L}^{\mathsf{MM}}_{\mathsf{Q}}\Big(\mathsf{MM}_R, \chi(\mathbf{r}_2)\Big)\Big),$$

it then sets

$$\mathsf{TK}_N \leftarrow \Big(\mathcal{S}_{\mathsf{MM}}\Big(\Big\{\mathsf{mtk}_{\mathbf{r}_1}, \mathsf{mtk}_{\mathbf{r}_2}\Big\}_{(\mathbf{r}_1,\mathbf{r}_2)\in\mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2}}, \mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}\Big(\mathsf{MM}_{\mathsf{att}_1}, \Big\langle \chi(\mathsf{att}_1), \chi(\mathsf{att}_2)\Big\rangle\Big)\Big), f(\mathsf{att}_1)\Big)$$

- **(Join case-2).** If N has form

$$\Big(\mathtt{in}, \langle f(\mathsf{att}_1), f(\mathsf{att}_2)\rangle, \Big(\mathcal{L}^{\pi}_{\mathsf{Q}}\Big(\mathsf{MM}_{\mathsf{att}_1,\mathsf{att}_2}, \chi(\mathbf{r})\Big)\Big)_{\chi(\mathbf{r})\in\mathbf{R_{in}}[\mathsf{att}_2]}, \Big\{\mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}(\mathsf{MM}_R, \chi(\mathbf{r}_1)\Big\}_{\substack{(\mathbf{r}_1,\mathbf{r}_2)\in\mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2}\\ \wedge\chi(\mathbf{r}_2)\in\mathbf{R_{in}}[\mathsf{att}_2]}}\Big),$$

then it first computes for all $(\mathbf{r}_1, \mathbf{r}_2) \in \mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2}$ and $\chi(\mathbf{r}_2) \in \mathbf{R_{in}}[\mathsf{att}_2]$,

$$\mathsf{mtk}_1 \leftarrow \mathcal{S}_{\mathsf{MM}}\Big(\mathcal{L}^{\mathsf{MM}}_{\mathsf{Q}}\Big(\mathsf{MM}_R, \chi(\mathbf{r}_1)\Big)\Big)$$

then if $\langle f(\mathsf{att}_1), f(\mathsf{att}_2)\rangle$ has never been queried before, and by leveraging the key-equivocation of $\Sigma^{\pi}_{\mathsf{MM}}$, it generates a key such that[4]

$$K_{f(\mathsf{att}_1),f(\mathsf{att}_2)} \leftarrow \mathcal{S}^{\pi}_{\mathsf{MM}}\Big(\{\mathsf{mtk}_{\mathbf{r}}\}_{\mathbf{r}}, \Big(\mathcal{L}^{\pi}_{\mathsf{Q}}\Big(\mathsf{MM}_{\mathsf{att}_1,\mathsf{att}_2}, \chi(\mathbf{r})\Big)\Big)_{\chi(\mathbf{r})}\Big)$$

otherwise if $\langle f(\mathsf{att}_1), f(\mathsf{att}_2)\rangle$ has been queried before, it uses the previously generated key and sets

$$\mathsf{TK}_N \leftarrow \Big(K_{f(\mathsf{att}_1),f(\mathsf{att}_2)}, f(\mathsf{att}_1), f(\mathsf{att}_2)\Big)$$

- **(Join case-3).** If N has form $\Big(\mathtt{inter}, f(\mathsf{att}_1), f(\mathsf{att}_2)\Big)$, then it sets

$$\mathsf{TK}_N \leftarrow (f(\mathsf{att}_1), f(\mathsf{att}_2))$$

It remains to show that for all probabilistic polynomial-time adversaries $\mathcal{A}$, the probability that $\mathbf{Real}_{\mathrm{OPX},\mathcal{A}}(k)$ outputs 1 is negligibly-close to the probability that $\mathbf{Ideal}_{\mathrm{OPX},\mathcal{A},\mathcal{S}}(k)$ outputs 1. We do this using the following sequence of games:

Game_0 : is the same as a $\mathbf{Real}_{\mathrm{OPX},\mathcal{A}}(k)$ experiment.

Game_1 : is the same as Game_0, except that EMM_C is replaced with the output of $\mathcal{S}_{\mathsf{MM}}(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}(\mathsf{MM}_C))$ and every *leaf projection* node of form $\pi_{\mathsf{att}}(\mathbf{T})$ is replaced with the output of

$$\mathcal{S}_{\mathsf{MM}}\Big(\mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}\Big(\mathsf{MM}_C, \chi(\mathsf{att})\Big)\Big),$$

Game_2 : is the same as Game_1, except that EMM_V is replaced with the output of $\mathcal{S}_{\mathsf{MM}}(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}(\mathsf{MM}_V))$ and, every *leaf select* node of form $\sigma_{\mathsf{att}=a}(\mathbf{T})$ is replaced with the output of

$$\mathcal{S}_{\mathsf{MM}}\Big((\mathsf{mtk}_{\mathbf{r}})_{\mathbf{r}\in\mathsf{DB}_{\mathsf{att}=a}}, \mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}\Big(\mathsf{MM}_V, \Big\langle a, \chi(\mathsf{att})\Big\rangle\Big)\Big).$$

Game_{2+i} for $i \in [\#\mathsf{DB}^{\mathsf{T}}]$: is the same as Game_{1+i}, except that $\mathsf{EMM}_{\mathbf{c}_i}$ is replaced with the output of $\mathcal{S}_{\mathsf{MM}}(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}(\mathsf{MM}_{\mathbf{c}_i}))$ and, every *leaf join* node of form $\mathbf{T}_1 \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{T}_2$ is replaced with the output of

$$\Big(\mathcal{S}_{\mathsf{MM}}\Big(\Big\{\mathsf{mtk}_{\mathbf{r}_1}, \mathsf{mtk}_{\mathbf{r}_2}\Big\}_{(\mathbf{r}_1,\mathbf{r}_2)\in\mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2}}, \mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}\Big(\mathsf{MM}_{\mathsf{att}_1}, \Big\langle \chi(\mathsf{att}_1), \chi(\mathsf{att}_2)\Big\rangle\Big)\Big), f(\mathsf{att}_1)\Big)$$

[4]Note that the key will be generated based on all previously simulated row tokens on that particular column; and this is why we omit the indices from the notation in order to capture this aspect.

ID	Name	Course
A05	Alice	16
A12	Bob	18
A03	Eve	18

Course	Department
16	CS
18	Math

Figure 5: Plaintext database DB.

$\mathsf{Game}_{3+\#\mathsf{DB}^\intercal}$: is the same as $\mathsf{Game}_{2+\#\mathsf{DB}^\intercal}$, except that SET is replaced by a set composed of values generated uniformly at random, and every *internal select* node of the form $\sigma_{\mathsf{att}=a}(\mathbf{R_{in}})$ is replaced with $(K_{g(a\|\mathsf{att})}, f(\mathsf{att}))$, where $K_{g(a\|\mathsf{att})}$ is generated as detailed above.

$\mathsf{Game}_{3+\#\mathsf{DB}^\intercal+i}$ for $i \in [(\#\mathsf{DB}^\intercal)^2]$: is the same as $\mathsf{Game}_{2+\#\mathsf{DB}^\intercal+i}$, except that $\mathsf{EMM}_{\mathbf{c}_i,\mathbf{c}'_i}$ is replaced with the output of $\mathcal{S}_{\mathsf{MM}}(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}(\mathsf{MM}_{\mathbf{c}_i,\mathbf{c}'_i}))$, and every *internal join* node of form $\mathbf{T} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{R_{in}}$ is replaced with the output of

$$\left(\mathcal{S}^{\pi}_{\mathsf{MM}}\left(\{\mathsf{mtk}_{\mathbf{r}}\}_{\mathbf{r}}, \left(\mathcal{L}^{\pi}_{\mathsf{Q}}\left(\mathsf{MM}_{\mathsf{att}_1,\mathsf{att}_2}, \chi(\mathbf{r})\right)\right)_{\chi(\mathbf{r})}\right), f(\mathsf{att}_1), f(\mathsf{att}_2)\right)$$

$\mathsf{Game}_{4+\#\mathsf{DB}^\intercal+(\#\mathsf{DB}^\intercal)^2}$: is the same as $\mathsf{Game}_{3+\#\mathsf{DB}^\intercal+(\#\mathsf{DB}^\intercal)^2}$ except that EMM_R is replaced with the output of $\mathcal{S}_{\mathsf{MM}}(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}(\mathsf{MM}_R))$ and every row token $\mathsf{mtk}_{\mathbf{r}}$ for a row $\mathbf{r}$ is replaced with the output of[5] of

$$\mathcal{S}_{\mathsf{MM}}\left(\mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}\left(\mathsf{MM}_R, \left\langle \mathsf{tbl}(\mathbf{r}), \mathsf{rrk}(\mathbf{r})\right\rangle\right)\right)$$

where $\mathrm{ct}_j \leftarrow \mathsf{Enc}_{K_1}(r_j)$.

$\mathsf{Game}_{5+\#\mathsf{DB}^\intercal+(\#\mathsf{DB}^\intercal)^2}$: is the same as $\mathsf{Game}_{4+\#\mathsf{DB}^\intercal+(\#\mathsf{DB}^\intercal)^2}$, except that every SKE encryption ct of a message m is replaced with $\mathrm{ct} \leftarrow \mathsf{Enc}_{K_1}(\mathbf{0}^{|m|})$.

Note that $\mathsf{Game}_{5+\#\mathsf{DB}^\intercal+(\#\mathsf{DB}^\intercal)^2}$ is identical to $\mathbf{Ideal}_{\mathrm{OPX},\mathcal{A},\mathcal{S}}(k)$.

■

C A Concrete Example of Indexed HNF

Similar to [16], our examples also rely on a small database DB composed of two tables $\mathbf{T}_1$ and $\mathbf{T}_2$ that have three and two rows, respectively. The schema of $\mathbf{T}_1$ is $\mathbb{S}(\mathbf{T}_1) = (\mathsf{ID}, \mathsf{Name}, \mathsf{Course})$ and that of $\mathbf{T}_2$ is $\mathbb{S}(\mathbf{T}_2) = (\mathsf{Course}, \mathsf{Department})$. The tables are described in Figure (5).

Figure (6) shows the result of applying our method to index the database $\mathsf{DB} = (\mathbf{T}_1, \mathbf{T}_2)$, as detailed in Section (**??**). There are five multi-maps MM_R, MM_C, MM_V, $\mathsf{MM}_{\mathsf{Course}}$, $\mathsf{MM}_{\mathsf{T}_2.\mathsf{Course},\mathsf{T}_1.\mathsf{Course}}$, and a set SET. e detail below how the indexing works for this example.

The first multi-map, MM_R, maps every row in each table to its encrypted content. As an instance, the first row of $\mathbf{T}_1$ is composed of three values $(A05, \mathrm{Alice}, 16)$ that will get encrypted

[5]Note that we are making the assumption that all attributes have the same domain, otherwise, there would be a number of games smaller than $(\#\mathsf{DB}^\intercal)^2$.

MM_R	
$\mathsf{T}_1 \| r_1$	$\mathsf{Enc}_K(\text{A05})$, $\mathsf{Enc}_K(\text{Alice})$, $\mathsf{Enc}_K(16)$
$\mathsf{T}_1 \| r_2$	$\mathsf{Enc}_K(\text{A12})$, $\mathsf{Enc}_K(\text{Bob})$, $\mathsf{Enc}_K(18)$
$\mathsf{T}_1 \| r_3$	$\mathsf{Enc}_K(\text{A03})$, $\mathsf{Enc}_K(\text{Eve})$, $\mathsf{Enc}_K(18)$
$\mathsf{T}_2 \| r_1$	$\mathsf{Enc}_K(16)$, $\mathsf{Enc}_K(\text{CS})$
$\mathsf{T}_2 \| r_2$	$\mathsf{Enc}_K(18)$, $\mathsf{Enc}_K(\text{Math})$

MM_C	
$\mathsf{T}_1 \| c_1$	$\mathsf{Enc}_K(\text{A05})$, $\mathsf{Enc}_K(\text{A12})$, $\mathsf{Enc}_K(\text{A03})$
$\mathsf{T}_1 \| c_2$	$\mathsf{Enc}_K(\text{Alice})$, $\mathsf{Enc}_K(\text{Bob})$, $\mathsf{Enc}_K(\text{Eve})$
$\mathsf{T}_1 \| c_3$	$\mathsf{Enc}_K(16)$, $\mathsf{Enc}_K(18)$, $\mathsf{Enc}_K(18)$
$\mathsf{T}_2 \| c_1$	$\mathsf{Enc}_K(16)$, $\mathsf{Enc}_K(18)$
$\mathsf{T}_2 \| c_2$	$\mathsf{Enc}_K(\text{CS})$, $\mathsf{Enc}_K(\text{Math})$

$\mathsf{MM}_{\mathsf{Course}}$	
$\mathsf{T}_1 \| c_3 \| \mathsf{T}_2 \| c_1$	$(\mathsf{T}_1 \| r_1, \mathsf{T}_2 \| r_1)$, $(\mathsf{T}_1 \| r_2, \mathsf{T}_2 \| r_2)$, $(\mathsf{T}_1 \| r_3, \mathsf{T}_2 \| r_2)$

$\mathsf{MM}_{\mathsf{T}_2.\mathsf{Course},\mathsf{T}_1.\mathsf{Course}}$	
$\mathsf{T}_2 \| r_1$	(T_1, r_1)
$\mathsf{T}_2 \| r_2$	(T_1, r_2), (T_1, r_3)

MM_V	
$\mathsf{T}_1 \| c_1 \| \text{A05}$	T_1, r_1
$\mathsf{T}_1 \| c_1 \| \text{A12}$	T_1, r_2
$\mathsf{T}_1 \| c_1 \| \text{A03}$	T_1, r_3
$\mathsf{T}_1 \| c_2 \| \text{Alice}$	T_1, r_1
$\mathsf{T}_1 \| c_2 \| \text{Bob}$	T_1, r_2
$\mathsf{T}_1 \| c_2 \| \text{Eve}$	T_1, r_3
$\mathsf{T}_1 \| c_3 \| 16$	T_1, r_1
$\mathsf{T}_1 \| c_3 \| 18$	$(\mathsf{T}_1, r_2), (\mathsf{T}_1, r_3)$
$\mathsf{T}_2 \| c_1 \| 16$	T_2, r_1
$\mathsf{T}_2 \| c_1 \| 18$	T_2, r_2
$\mathsf{T}_2 \| c_2 \| \text{CS}$	T_2, r_1
$\mathsf{T}_2 \| c_2 \| \text{Math}$	T_2, r_2

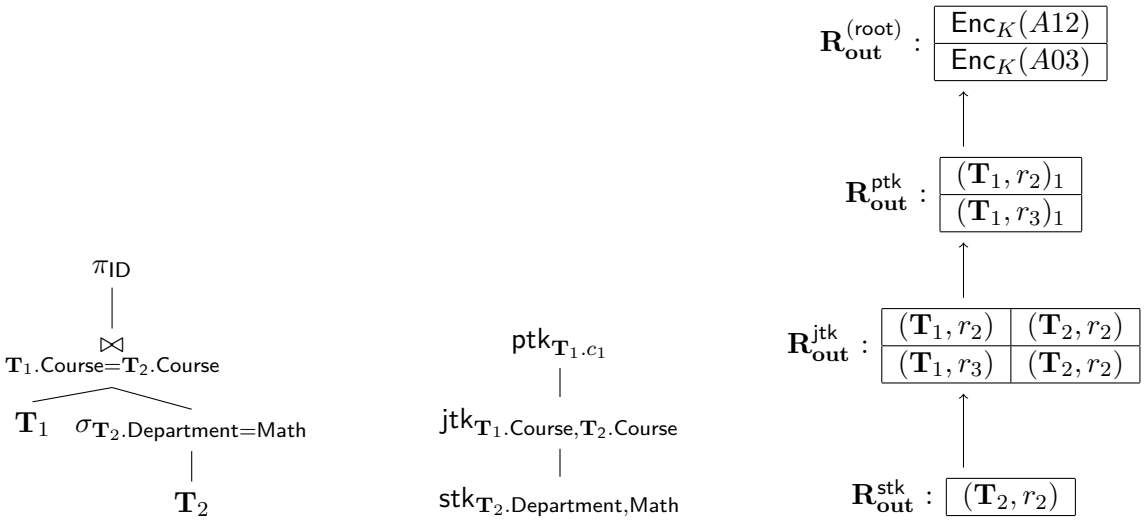
SET
$\mathsf{T}_1 \| r_1 \| c_1 \| \text{A05}$
$\mathsf{T}_1 \| r_2 \| c_1 \| \text{A12}$
$\mathsf{T}_1 \| r_3 \| c_1 \| \text{A03}$
$\mathsf{T}_1 \| r_1 \| c_2 \| \text{Alice}$
$\mathsf{T}_1 \| r_2 \| c_2 \| \text{Bob}$
$\mathsf{T}_1 \| r_3 \| c_2 \| \text{Eve}$
$\mathsf{T}_1 \| r_1 \| c_3 \| 16$
$\mathsf{T}_1 \| r_2 \| c_3 \| 18$
$\mathsf{T}_1 \| r_3 \| c_3 \| 18$
$\mathsf{T}_2 \| r_1 \| c_1 \| 16$
$\mathsf{T}_2 \| r_2 \| c_1 \| 18$
$\mathsf{T}_2 \| r_1 \| c_2 \| \text{CS}$
$\mathsf{T}_2 \| r_2 \| c_2 \| \text{Math}$

Figure 6: Indexed database.

and stored in MM_R. Since DB has five rows, MM_R has five pairs. The second multi-map, MM_C, maps each column of every table to its encrypted content. Similarly, as DB is composed of five columns in total, MM_C has five pairs. The third multi-map, MM_V, maps every unique value in every table to its coordinates in the plaintext table. For example, the value 18 in $\mathbf{T}_1$ exists in two positions, in particular, in the second and third row. The join multi-map, $\mathsf{MM}_{\mathsf{Course}}$, maps the columns' coordinates to the pair of rows that have the same value. In our example, as the first row of both tables contains 16, and the second and third rows of $\mathbf{T}_1$ and the second row of $\mathbf{T}_2$ contain 18, the label/tuple pair

$$\Big(\mathbf{T}_1 \| \mathbf{c}_3 \| \mathbf{T}_2 \| \mathbf{c}_1, ((\mathbf{T}_1 \| r_1, \mathbf{T}_2 \| r_1), (\mathbf{T}_1 \| r_2, \mathbf{T}_2 \| r_2)), (\mathbf{T}_1 \| r_3, \mathbf{T}_2 \| r_2))\Big)$$

is added to $\mathsf{MM}_{\mathsf{Course}}$. The correlated join multi-map, $\mathsf{MM}_{\mathsf{T}_2.\mathsf{Course},\mathsf{T}_1.\mathsf{Course}}$, maps every row in each table to all rows that contain the same value. In our example, for the attribute Course, the first row in $\mathbf{T}_2$ maps to the first row in $\mathbf{T}_1$ while the second row in $\mathbf{T}_2$ maps to second and third rows in $\mathbf{T}_1$. Finally, the set structure SET stores all values in every row and every attribute.



(a) Query tree (b) Token tree (c) Intermediate results of token tree

Figure 7: A query tree translated to a token tree which is then executed using the indexed database.

A concrete query. Let us consider the following simple SQL query

$$\mathsf{Select}\ \mathbf{T}_1.\mathrm{ID}\ \mathsf{From}\ \mathbf{T}_1, \mathbf{T}_2\ \mathsf{Where}\ \mathbf{T}_2.\mathrm{Department} = \mathrm{Math}\ \mathsf{AND}\ \mathbf{T}_2.\mathrm{Course} = \mathbf{T}_1.\mathrm{Course}.$$

This SQL query can be rewritten as a query tree, see Figure (**??**), and then translated, based on OPX protocol into a token tree as depicted in Figure (7b).[6]

We detail in Figure (7c) the intermediary results of the token tree execution using the indexed database and provide below a high level description of how it works.

The server starts by fetching from MM_V the tuple corresponding to $\mathbf{T}_2 \| c_2 \| 18$, which is equal to $\{(\mathbf{T}_2, r_2)\}$. This represents the first intermediary output $\mathbf{R}^{\mathsf{stk}}_{\mathbf{out}}$ which is also the input for the next node. For each element in $\mathbf{R}^{\mathsf{stk}}_{\mathbf{out}}$, the server fetches the corresponding tuple in $\mathsf{MM}_{\mathbf{T}_2.\mathsf{Course},\mathbf{T}_1.\mathsf{Course}}$, which is equal to $\{(\mathbf{T}_1, r_2), (\mathbf{T}_1, r_3)\}$. Now, the second intermediary output $\mathbf{R}^{\mathsf{jtk}}_{\mathbf{out}}$ is composed of all row coordinates from $\mathbf{T}_1$ that match $\mathbf{T}_2$. For the internal projection node, given $(1, \mathsf{in})$, the server will simply output the row tokens in the first attribute as $\mathbf{R}^{\mathsf{ptk}}_{\mathbf{out}}$.

Finally, the server fetches tuples from the MM_R that correspond to the remaining row tokens, as the final result of $\mathbf{R}^{\mathsf{root}}_{\mathbf{out}}$, which is equal to

$$\mathbf{R}^{\mathsf{root}}_{\mathbf{out}} = (\mathsf{Enc}_K(A12), \mathsf{Enc}_K(A03)).$$

Concrete storage overhead. The plaintext database DB is composed of thirteen cells excluding the tables attributes.[7] The indexed structure consists of fifty eight pairs. Assuming that a pair and a cell have the same bit length, our indexed representation of the database has a multiplicative

[6] For sake of clarity, this example of token tree generation does not accurately reflect the token protocol of OPX, but only gives a high level idea of its algorithmic generation.

[7] Note that our calculation does not take into account the security parameter and consider every (encrypted) cell as a one unit of storage.

storage overhead of 4.46. In particular, each of the multi-maps MM_R, MM_C, MM_V and the set SET have the same size as the plaintext database (i.e., 13 pairs). This explains the $4\times$ factor. It is worth emphasizing that even if one considers a larger database, the $4\times$ factor remains unchanged. The additive component of the multiplicative factor, i.e., the 0.46, will vary, however, from one database to another depending on the number of columns with the same domain and the number of equal rows in these columns.