SQL on Structurally-Encrypted Databases

Seny Kamara and Tarik Moataz

Brown University, Providence, USA
`seny@brown.edu, tarik_moataz@brown.edu`

Abstract. We show how to encrypt a relational database in such a way that it can efficiently support a large class of SQL queries. Our construction is based solely on structured encryption (STE) and does not make use of any property-preserving encryption (PPE) schemes such as deterministic and order-preserving encryption. As such, our approach leaks considerably less than PPE-based solutions which have recently been shown to reveal a lot of information in certain settings (Naveed et al., *CCS '15*). Our construction is efficient and—under some conditions on the database and queries—can have asymptotically-optimal query complexity. We also show how to extend our solution to be dynamic while maintaining the scheme's optimal query complexity.

1 Introduction

The problem of encrypted search has received attention from industry, academia and government due to its potential applications to cloud computing and database security. Most of the progress in this area, however, has been in the setting of keyword search on encrypted documents. While this has many applications in practice (e.g., email, NoSQL databases, desktop search engines, cloud document storage), much of the data produced and consumed in practice is stored and processed in *relational databases*. A relational database is, roughly speaking, a set of tables with rows representing entities/items and columns representing their attributes. The relational database model was proposed by Codd [18] and most relational DBs are queried using the structured query language (SQL) which is a special-purpose declarative language introduced by Chamberlain and Boyce [14].

The problem of encrypted relational DBs is one of the "holy-grails" of database security. As far as we know, it was first explicitly considered by Hacigümüs, Iyer, Li and Mehrotra [25] who described a quantization-based approach which leaks the range within which an item falls. In [37], Popa, Redfield, Zeldovich and Balakrishnan describe a system called CryptDB that can support a non-trivial subset of SQL without quantization. CryptDB achieves this in part by making use of property-preserving encryption (PPE) schemes like deterministic and order-preserving (OPE) encryption, which reveal equality and order, respectively. The high-level approach is to replace the plaintext operations needed to execute a SQL query (e.g., equality tests and comparisons) by the same operations on PPE-encrypted ciphertexts. This approach was later adopted by other systems including Cipherbase [3] and SEEED [23]. While this leads to systems that are efficient and legacy-friendly, it was shown by Naveed, Kamara and Wright [34] that PPE-based EDB systems can leak a lot of information when used in certain settings like electronic medical records (EMRs). In light of this result, the major open problem in encrypted search and, more generally, in database security is whether it is possible to efficiently execute SQL queries on encrypted DBs with less leakage than the PPE-based solutions.

Our contributions. In this work, we address this problem and propose the first solution for SQL on encrypted DBs that does not make use of either PPE or general-purpose primitives like fully-homomorphic encryption (FHE) or oblivious RAM (ORAM).[1] As such, our scheme leaks less than any of the previously-known practical approaches and is more practical than any solution based on FHE or ORAM. Our approach is efficient and handles a sub-class of SQL queries and an even larger class if we allow for a small amount of post-processing at the client.

More precisely, our construction handles the class of *conjunctive queries* [2] [15] which corresponds to SQL queries of the form

$$\mathsf{Select}\ \textit{attributes}\ \mathsf{From}\ \textit{tables}\ \mathsf{Where}\ \big(\mathsf{att}_1 = X_1 \wedge \cdots \wedge \mathsf{att}_\ell = X_\ell\big),$$

where att_1 through att_ℓ are attributes in the DB schema and X_1 through X_n are either attributes or constants. For ease of exposition, we mainly focus on conjunctive queries with Where predicates that are *uncorrelated* which, very roughly speaking, means that the attributes are not the same across terms (we refer the reader to Section 5 for a precise definition). The case of correlated predicates is quite involved so it is deferred to the full version of this work. While the class of conjunctive queries is smaller than the class supported by the PPE-based solutions, it is one of the most well-studied and useful classes of queries. Furthermore, as mentioned above, if one allows for a small amount of post-processing at the client, we show how to extend the expressiveness of our solution to a wider sub-class.

With respect to efficiency, we show that the query complexity of our scheme is asymptotically optimal in time and space when $(s_1 + \cdots + s_t)/h = O(1)$, where t denotes the number of tables in the query, s_i denotes the number of columns in the ith table and h denotes the number of attributes in the Select term of the query. Towards analyzing the asymptotic complexity of our solution, we precisely characterize the result size of an SPC query as a function of the

[1] In the full version of this work, we present a dynamic variant of our construction that makes use of ORAM to achieve forward-security, but it is only used to store and manage one of several data structures needed by the scheme. In other words, ORAM is not used to store and manage the entire database.

[2] We stress that conjunctive queries in the context of relational databases (and as used throughout this work) is conceptually unrelated to conjunctive *keyword* queries as studied in the searchable encryption literature (e.g., in [12,28]). In particular, our scheme does not make use of any searchable encryption schemes for conjunctive keyword queries and our problem cannot be solved by applying these schemes directly on tables. However it is worth mentioning that some of the techniques in the expressive SSE literature could possibly be leveraged to achieve a better leakage profile.

query and of the underlying relational database. This analysis, deferred to the full version of this work, could be of independent interest.

We also show how to extend our construction to be dynamic and to support two traditional SQL update operations: row addition and row deletions. Surprisingly, our dynamic construction has the same asymptotic efficiency as our static construction. Finally, we show how to extend our dynamic construction to be forward-secure at the cost of a poly-logarithmic overhead for updates but maintaining the same query complexity.

1.1 Possible Approaches

PPE-based. The PPE-based approach to EDBs essentially replaces the plaintext execution of a SQL query with an encrypted execution of the query by executing the server's low-level operations (i.e., comparisons and equality tests) directly on the encrypted cells. This can be done thanks to the properties of PPE which guarantee that operations on plaintexts can be done on ciphertexts as well. This "plug-and-play" approach makes the design of EDBs relatively straightforward since the only requirement is to replace plaintext cells with PPE-encrypted cells. This approach however has been shown to leak a lot of information in certain scenarios [34].

SSE-based. Searchable symmetric encryption (SSE) allows one to perform search queries on an encrypted document collection. While SSE constructions do not yield an encrypted relational database, they could be used to handle a very small subset of SQL. By applying SSE to a column one could handle queries of the form

$$\mathsf{Select}\ \mathit{attribute}\ \mathsf{From}\ \mathit{table}\ \mathsf{Where}\ \mathsf{att} = X,$$

where att is the attribute that has been indexed with SSE and X is a constant. If the SSE scheme supports ranges this would extend to queries of the form

$$\mathsf{Select}\ \mathit{attribute}\ \mathsf{From}\ \mathit{table}\ \mathsf{Where}\ \mathsf{att} \ominus X,$$

where $\ominus \in \{=, <, >\}$ and if it supports conjunctions it would extend to

$$\mathsf{Select}\ \mathit{attribute}\ \mathsf{From}\ \mathit{table}\ \mathsf{Where}\ \big(\mathsf{att} = X_1 \wedge \cdots \wedge \mathsf{att} = X_\ell\big).$$

Note that the supported queries in both cases are limited to a *single* column and a *single* table, and *don't support joins or projections.* This is, unfortunately, far from what is expected from a relational database. In addition, extending existing expressive SSE schemes (e.g., OXT [12], BlindSeer [36] or IEX [28]) to handle SQL operations would be highly non-trivial—unless one used the naive approach of executing many simple queries and having the server build the response (e.g., like the naive approach to conjunctions or disjunctions) which would leak a lot more. In general, expressiveness in SSE does not imply the same level of expressiveness in the relational setting, i.e., we cannot use an expressive SSE scheme in a "plug-and-play" fashion (similar to PPE) to handle the same level of expressiveness in relational databases.

Generic approaches. Fully-homomorphic encryption (FHE) or oblivious RAM (ORAM) could be used in a black-box fashion to handle *full* SQL. However, these approaches would be inefficient due the the inherent cost of the primitives.

1.2 Our Techniques

Conceptual approach. Our first step towards a solution is in isolating some of the conceptual difficulties of the problem. Relational DBs are relatively simple from a data structure perspective since they just consist of a set of two-dimensional arrays. The high-level challenge stems from SQL and, in particular, from its complexity (it can express first-order logic) and the fact that it is *declarative*. To overcome this we restrict ourselves to a simpler but widely applicable and well-studied subset of SQL queries (see above) and we take a more procedural view. More precisely, we work with the *relational algebra* formulation of SQL which is more amenable to cryptographic techniques. The relational algebra was introduced by Codd [18] as a way to formalize queries on relational databases. Roughly speaking, it consists of all the queries that can be expressed from a set of basic operations. It was later shown by Chandra and Merlin [15] that three of these operations (selection, projection and cross product) capture a large class of useful queries called *conjunctive queries* that have particularly nice theoretical properties. Since their introduction, conjunctive queries have been studied extensively in the database literature.

The subset of the relational algebra expressed by the selection, projection and cross product operators is also called the *SPC algebra*. By working in the SPC algebra, we not only get a procedural representation of SQL queries, but we also reduce the problem to handling just three basic operations. Conceptually, this is reminiscent of the benefits one gets by working with circuits in secure multi-party computation and FHE. Another important advantage of working in the SPC algebra is that it admits a *normal form*; that is, every SPC query can be written in a standard form. By working with this normal form, we get another benefit of general-purpose solutions which are that we can design and analyze a *single* construction that handles *all* SPC queries. Note, however, that like circuit representations the SPC normal form is not always guaranteed to be the most efficient.

The SPC algebra. As mentioned, the SPC algebra consists of all queries that can be expressed by a combination of the select, project and cross product operators which, at a high-level, work as follows. The select operator σ_Ψ takes as input a table T and outputs the rows of T that satisfy the predicate Ψ. The project operator $\pi_{\mathsf{att}_1,\ldots,\mathsf{att}_h}$ takes as input a table T and outputs the columns of T indexed by $\mathsf{att}_1,\ldots,\mathsf{att}_h$. Finally, the cross product operator $\mathsf{T}_1 \times \mathsf{T}_2$ takes two tables as input and outputs a third table consisting of rows in the cross product of T_1 and T_2 when viewed as sets of rows. An SPC query in normal form over a database $\mathsf{DB} = (\mathsf{T}_1, \ldots, \mathsf{T}_n)$ has the form,

$$\pi_{\mathsf{att}_1,\cdots,\mathsf{att}_h}\bigg([a_1] \times \cdots \times [a_f] \times \sigma_\Psi(\mathsf{T}_{i_1} \times \cdots \times \mathsf{T}_{i_t})\bigg),$$

where $[a_j]$ is a 1×1 table that holds a constant a_j for all $j \in [f]$, Ψ is of the form $\mathsf{att}_1 = X_1 \wedge \cdots \wedge \mathsf{att}_\ell = X_\ell$ where $\mathsf{att}_1, \ldots, \mathsf{att}_\ell$ are attributes in the schema of DB and $X_1, \ldots, X_\ell$ are either attributes or constants. So, concretely, our problem reduces to the problem of encrypting a relational database $\mathsf{DB} = (\mathsf{T}_1, \ldots, \mathsf{T}_n)$ in such a way that it can support SPC queries in normal form.

Structured encryption & constructive queries. The main difficulty in the case of relational DBs and, in particular, in handling SPC queries is that queries are *constructive* in the sense that they produce new data structures from the original base structure. Intuitively, handling constructive queries (without interaction) is particularly challenging because the intermediate and final structures that have to be created by the server to answer the query are *dependent* on the query and, therefore, cannot be constructed by the client in the setup/preprocessing phase. An important observation about relational DBs that underlies our approach, however, is that while SPC queries are constructive, they are not arbitrarily so. In other words, the tables needed to answer an SPC query are not completely arbitrary but are structured in a way that can be predicted at setup. What is query-dependent is the *content* of these tables but, crucially, all of that content is already stored in the original database. So the challenge then is to provide the server with the means to construct the appropriate intermediate and final tables and to design encrypted structures that will allow it to efficiently find the (encrypted) content it needs to create those tables.

Handling SPC normal form queries. By taking a closer look at the SPC normal form, one can see that the first intermediate table needed to answer a query is the cross product $\mathsf{T}' = \mathsf{T}_{i_1} \times \cdots \times \mathsf{T}_{i_t}$. Ignoring the cross products with $[a_1], \ldots, [a_f]$ for ease of exposition, the remaining intermediate tables as well as the final table are "sub-tables" of T' that result from selecting a subset of rows (according to Ψ) and keeping a subset of columns (according to $\mathsf{att}_1, \ldots, \mathsf{att}_h$). Handling such a query naively requires one to first compute the cross product of the tables which can be prohibitively large. As we show in Section 5, however, SPC normal form queries can be rewritten in a different and optimized form we introduce called the heuristic normal form (HNF). We then show how to encrypt the database in such a way that we can handle queries in their HNF form. At a high level, we achieve this by creating a set of encrypted structures that store different representations of the database. For example, one of the encrypted structures stores a row-wise representation of the database whereas another stores a column-wise representation. By using these various representations and by combining them in an appropriate manner, we can generate tokens for the server to recover the encrypted database rows needed for it to process the query in its HNF form.

The SPX framework. We describe and analyze our scheme using algorithms that make black-box use of several lower-level STE schemes (e.g., multi-map and dictionary encryption schemes). As such, our construction is more of a framework that can be used to design encrypted relational databases with various efficiency/leakage trade-offs. In fact, in Section 7.1, we describe an instantiations of our framework with a zero-leakage variant of the TWORAM-based construction of Garg, Papamanthou and Mohassel [20] which results in a very low-leakage construction at the cost of an additional poly-logarithmic overhead.

Dynamism. We show how to extend our static construction to be dynamic. This is challenging as we want to maintain the scheme's query complexity while not introducing additional leakage. From a functionality perspective, we restrict our attention to row additions and deletions and leave as important open problem the handling of more complex update operations. While real-world databases also handle edits, we note that these two update operations are already interesting in practice and non-trivial to achieve. As discussed above, we store different encrypted representations of the database. One of these representations, however, stores parts of the database that are highly inter-correlated. The difficulty this poses is that we cannot simply add or remove items from this structure as any change affects all the other items stored in the structure. We introduce a two-party protocol to solve this challenge without the client having to trivially download the entire structure and without leaking too much information to the server. We then show how to extend this solution to be forward-secure at the cost of a poly-logarithmic blowup (for updates). This is achieved by storing and managing one of the structures in an oblivious RAM.

A note on our techniques. We stress that our approach to handle the SPC algebra is very different from how these queries are handled on plaintext databases. In other words, our approach does not simply replicate standard data structures and algorithms from the database literature. In fact, our approach to handling SPC queries could be of independent interest for plaintext relational databases.

2 Related Work

Searchable & structured encryption. Encrypted search was first considered explicitly by Song et al. in [38] which introduced the notion of searchable symmetric encryption (SSE). Goh provided the first security definition for SSE and a solution based on Bloom filters with linear search complexity. Curtmola et al. introduced and formulated the notion of adaptive semantic security for SSE [19] together with optimal-time and optimal-space constructions. Chase and Kamara introduced the notion of structured encryption which generalizes SSE to arbitrary data structures [16]. Cash et al. [11] show how to construct optimal-time SSE schemes with low I/O complexity and Cash and Tessaro [13] gave lower bounds on the locality of adaptively-secure SSE schemes. Asharov et al. build SSE schemes with optimal locality, optimal space overhead and nearly-optimal read efficiency [4]. Garg et al. [20] presented a new SSE construction with reduced leakage leveraging oblivious RAM and garbled RAM techniques. Bost [9] proposed an efficient forward-secure SSE construction based on trapdoor permutations. SSE has also been considered in the multi-user setting [19,27]. Pappas et al. [36] proposed a multi-user SSE construction based on garbled circuits and Bloom filters that can support Boolean formulas, ranges and stemming. Other approaches for encrypted search include oblivious RAMs (ORAM) [22], secure multi-party computation [6], functional encryption [8] and fully-homomorphic encryption [21] as well as solutions based on deterministic encryption [5] and order-preserving encryption (OPE) [7].

Encrypted relational databases. As far as we know the first encrypted relational DB solution was proposed by Hacigümüs et al. [25] and was based on quantization. Roughly speaking, the attribute space of each column is partitioned into bins and each element in the column is replaced with its bin number. Popa et al. proposed CryptDB [37]. CryptDB was the first non-quantization-based solution and can handle a large subset of SQL. Instead of quantization, CryptDB relies on PPE like deterministic encryption [5] and OPE [2,7]. The CryptDB design influenced the Cipherbase system from Arasu *et al.* [3] and the SEEED system from Grofig *et al.* [23]. In [34], Naveed et al. study the security of these PPE-based solutions in the context of medical data. Recently, Grubbs, Ristenpart and Shmatikov [24] point out pitfalls in integrating encrypted database solutions in real-world database management systems (DBMS).

Attacks on SSE. While we do not consider the problem of designing an SSE scheme in this work, we can use SSE schemes as building blocks to instantiate SPX. Several works have proposed attacks that try to exploit the leakage of SSE. This includes the query-recovery attacks of Islam et al. [26], of Cash et al. [10] and of Zhang et al. [40]. Recently, Abdelraheem et al. [33], presented attacks on encrypted relational databases. We briefly mention here that although the attacks in [33] are ostensibly on relational EDBs, they are not related to or applicable to our construction. For more details on these attacks and their relation to our work we refer the reader to Section 7.3.

3 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 $x \overset{\$}{\leftarrow} X$ 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 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 $\{(\ell_i, \mathbf{t}_i)\}_{i\leq n}$ 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. Note that tuples may have different lengths. 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). We refer to the set of all possible queries a data structure supports as its *query space* and to the set of its possible responses as its *response space*. For some data structure DS we sometimes write $\mathsf{DS} : \mathbf{Q} \to \mathbf{R}$ to mean that DS has query and response spaces $\mathbf{Q}$ and $\mathbf{R}$, respectively.

Relational databases. A relational database $\mathsf{DB} = (\mathsf{T}_1, \ldots, \mathsf{T}_n)$ is a set of *tables* where each table 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 *domain* (e.g., integers, booleans, strings). We define the *schema* of a table T to be its set of attributes and denote it $\mathbb{S}(\mathsf{T})$. The schema of a database $\mathsf{DB} = (\mathsf{T}_1, \ldots, \mathsf{T}_n)$ is then the set $\mathbb{S}(\mathsf{DB}) = \bigcup_i \mathbb{S}(\mathsf{T}_i)$. We assume the attributes in $\mathbb{S}(\mathsf{DB})$ are unique and represented as positive integers. We denote a table T's number of rows as $\|\mathsf{T}\|_r$ and its number of columns as $\|\mathsf{T}\|_c$.

We sometimes view tables as a tuple of rows and write $\mathbf{r} \in \mathsf{T}$ and sometimes as a tuple of columns and write $\mathbf{c} \in \mathsf{T}^\intercal$. Similarly, we write $\mathbf{r} \in \mathsf{DB}$ and $\mathbf{c} \in \mathsf{DB}^\intercal$ for $\mathbf{r} \in \bigcup_i \mathsf{T}_i$ and $\mathbf{c} \in \bigcup_i \mathsf{T}_i^\intercal$, respectively. For a row $\mathbf{r} \in \mathsf{T}_i$, its table identifier $\mathsf{tbl}(\mathbf{r})$ is i and its row rank $\mathsf{rrk}(\mathbf{r})$ is its position in T_i when viewed as a tuple of rows. Similarly, for a column $\mathbf{c} \in \mathsf{T}_i^\intercal$, its table identifier $\mathsf{tbl}(\mathbf{c})$ is i and its column rank $\mathsf{crk}(\mathbf{c})$ is its position in 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 $\chi(\mathbf{r}) \overset{def}{=} (\mathsf{tbl}(\mathbf{r}), \mathsf{rrk}(\mathbf{r}))$ and $\chi(\mathbf{c}) \overset{def}{=} (\mathsf{tbl}(\mathbf{c}), \mathsf{crk}(\mathbf{c}))$, respectively, as their *coordinates* in DB. Similarly, we denote by $\chi(\mathsf{att})$ the coordinate of column $\mathbf{c}$ with attribute $\mathsf{att} \in \mathbb{S}(\mathsf{DB})$ such that $\chi(\mathsf{att}) = \chi(\mathbf{c})$. 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 $\mathbf{r} \in \mathsf{T}_i$ 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 any pair of attributes $\mathsf{att}_1, \mathsf{att}_2 \in \mathbb{S}(\mathsf{DB})$ with the same domain 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 $\big\{(\mathbf{r}_1, \mathbf{r}_2) \in \mathsf{DB}^2 : \mathbf{r}_1[\mathsf{att}_1] = \mathbf{r}_2[\mathsf{att}_2]\big\}$. 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 $\big\{\mathbf{r} \in \mathsf{DB} : \mathbf{r}[\mathsf{att}] = a\big\}$.

SQL. In practice, relational databases are queried using the special-purpose language SQL, introduced by Chamberlain and Boyce [14]. 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 typically have the form

$$\mathsf{Select}\ \textit{attributes}\ \mathsf{From}\ \textit{tables}\ \mathsf{Where}\ \textit{condition},$$

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* [15] 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

$$\textsf{Select}\ \textit{attributes}\ \textsf{From}\ \textit{tables}\ \textsf{Where}\ \big(\textsf{att}_1 = X_1 \wedge \cdots \wedge \textsf{att}_\ell = X_\ell\big),$$

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

The SPC algebra. It was shown by Chandra and Merlin [15] 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 $\textsf{T}$ and outputs a new table $\textsf{T}'$ that includes the rows of $\textsf{T}$ that satisfy the predicate Ψ. The *projection* operator $\pi_{\textsf{att}_1,\ldots,\textsf{att}_h}$ is parameterized by a set of attributes $\textsf{att}_1, \ldots, \textsf{att}_h$ and takes as input a table $\textsf{T}$ and outputs a table $\textsf{T}'$ that consists of the columns of $\textsf{T}$ indexed by $\textsf{att}_1$ through $\textsf{att}_n$. The *cross product* operator $\times$ takes as input two tables $\textsf{T}_1$ and $\textsf{T}_2$ and outputs a new table $\textsf{T}' = \textsf{T}_1 \times \textsf{T}_2$ such that each row of $\textsf{T}'$ is an element of the cross product between the set of rows of $\textsf{T}_1$ and the set of rows of $\textsf{T}_2$. 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 1 below.

Definition 1 (SPC algebra). *Let* $\textsf{DB} = (\textsf{T}_1, \ldots, \textsf{T}_n)$ *be a relational database. The SPC algebra consists of any query that results from the combination of the following operators:*

- $\textsf{T}' \leftarrow \sigma_\Psi(\textsf{T})$*: the* select *operator is parameterized with a predicate* Ψ *of form* $\textsf{att}_1 = X_1 \wedge \cdots \wedge \textsf{att}_\ell = X_\ell$*, where* $\textsf{att}_i \in \mathbb{S}(\textsf{DB})$ *and* X_i *is either a constant equal to a in the domain of* $\textsf{att}_i$ *(type-1) or an attribute* $\textsf{x}_j \in \mathbb{S}(\textsf{DB})$ *(type-2). It takes as input a table* $\textsf{T} \in \textsf{DB}$ *and outputs a table* $\textsf{T}' = \{\mathbf{r} \in \textsf{T} : \Psi(\mathbf{r}) = 1\}$*, where terms of the form* $\textsf{att}_i = \textsf{x}_j$ *are satisfied if* $\mathbf{r}[\textsf{att}_i] = \mathbf{r}[\textsf{x}_j]$ *and terms of the form* $\textsf{att}_i = a$ *are satisfied if* $\mathbf{r}[\textsf{att}_i] = a$*.*
- $\textsf{T}' \leftarrow \pi_{\textsf{att}_1,\ldots,\textsf{att}_h}(\textsf{T})$*: the* project *operator is parameterized by a set of attributes* $\textsf{att}_1, \ldots, \textsf{att}_h \in \mathbb{S}(\textsf{DB})$*. It takes as input a table* $\textsf{T} \in \textsf{DB}$ *and outputs a table* $\textsf{T}' = \{\langle \mathbf{r}[\textsf{att}_1], \ldots, \mathbf{r}[\textsf{att}_h]\rangle : \mathbf{r} \in \textsf{T}\}$*.*
- $\textsf{R} \leftarrow \textsf{T}_1 \times \textsf{T}_2$*: the* cross product *operator takes as input two tables* $\textsf{T}_1$ *and* $\textsf{T}_2$ *and outputs a result table* $\textsf{R} = \big\{\big\langle \mathbf{r}, \mathbf{v}\big\rangle : \mathbf{r} \in \textsf{T}_1 \text{ and } \mathbf{v} \in \textsf{T}_2\big\}$*, where* $\langle \mathbf{r}, \mathbf{v}\rangle$ *is the concatenation of rows* $\mathbf{r}$ *and* $\mathbf{v}$*.*

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.

SPC normal form. Any query in the SPC algebra can be reduced to a *normal form* using a certain set of well-known identities. The normal form of an SPC query over a relational database $\textsf{DB} = (\textsf{T}_1, \ldots, \textsf{T}_n)$ has the form:

$$\pi_{\textsf{att}_1,\cdots,\textsf{att}_h}\Bigg([a_1] \times \cdots \times [a_f] \times \sigma_\Psi(\textsf{T}_{i_1} \times \cdots \times \textsf{T}_{i_t})\Bigg),$$

where $a_1, \ldots, a_f \in \bigcup_{\mathsf{att} \in \mathbb{S}(\mathsf{DB})} \mathsf{dom}(\mathsf{att})$ and $[a_j]$ is the 1×1 table that holds a_j. The 1×1 tables are needed for the normal form to have enough expressive power to capture the SPC algebra (for more details see [1]). Here, the attributes $\mathsf{att}_1, \ldots, \mathsf{att}_h$ in the projection are either in $\mathbb{S}(\mathsf{DB})$ or refer to the columns generated by $[a_1]$ through $[a_f]$. In the latter case, we say that they are *virtual attributes* and are in $\mathbb{S}(\mathsf{VDB})$, where VDB is the *virtual database* defined as $\mathsf{VDB} = \big([a_1], \ldots, [a_f]\big)$.

One of the advantages of working in the relational algebra is that it allows for powerful optimization techniques. Given a query, we can use several identities to rewrite the query so that it can be executed more efficiently. The topic of query optimization is a large and important area of research in both database theory and engineering and real-world database management systems crucially rely on sophisticated query optimization algorithms. The main disadvantage of working with SPC queries in normal form is that their execution is extremely expensive, i.e., exponential in t. Furthermore, it is a-priori unclear how one could use standard query optimization techniques over encrypted data. We will see in Section 5, however, that these challenges can be overcome.

We note that while executing normal form SPC queries is prohibitively expensive, converting conjunctive SQL queries to normal form SPC queries is a well-studied problem with highly-optimized solutions. In particular, the queries that result from such a translation are "compact" in the sense that the number of projects, selects and cross products in the resulting SPC query is the same as the number of attributes, tables and conditions, respectively, in the original SQL query (for an overview of SQL-to-SPC translation we refer the reader to [39]).

Basic cryptographic primitives. We make use of encryption schemes that are random-ciphertext-secure against chosen-plaintext attacks (RCPA). RCPA-secure encryption can be instantiated practically using either the standard PRF-based private-key encryption scheme or, e.g., AES in counter mode.

4 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 [16] considered schemes that encrypt both a structure and a set of associated data items (e.g., documents, emails, user profiles etc.). In [17], 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.

Our main construction, SPX, is response-hiding but makes use of response-revealing schemes as building blocks. Furthermore, SPX's building blocks can be instantiated using either non-interactive or interactive schemes. We define response-hiding and response-revealing schemes below, but only for the non-interactive setting. The definitions, however, can be naturally extended to the interactive case. 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 2 (Response-revealing structured encryption [16]). *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*.*
- $\big\{\perp, r\big\} \leftarrow \mathsf{Query}(\mathsf{EDS}, \mathsf{tk})$*: 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* $\mathsf{DS} : \mathbf{Q} \rightarrow \mathbf{R}$*, 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 (Response-hiding structured encryption [16]). *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 2 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* $\mathsf{DS} : \mathbf{Q} \rightarrow \mathbf{R}$*, 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 [19] and later generalized to structured encryption in [16].

Definition 4 (Adaptive semantic security [19,16]). *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 $z \in \{0,1\}^*$:*

$\mathbf{Real}_{\Sigma,\mathcal{A}}(k)$*: 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.*

$\mathbf{Ideal}_{\Sigma,\mathcal{A},\mathcal{S}}(k)$*: 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 $\big(\mathsf{DS}(q_i), \mathcal{L}_\mathsf{Q}(\mathsf{DS}, q_i)\big)$ 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 there exists a PPT *simulator $\mathcal{S}$ such that for all* PPT *adversaries $\mathcal{A}$, for all $z \in \{0,1\}^*$, the following expression is negligible in k:*

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

The security definition for *response-hiding* schemes can be derived from Definition 4 by giving the simulator $\big(\perp, \mathcal{L}_\mathsf{Q}(\mathsf{DS}, q_i)\big)$ instead of $\big(\mathsf{DS}(q_i), \mathcal{L}_\mathsf{Q}(\mathsf{DS}, q_i)\big)$.

5 SPX: A Relational Database Encryption Scheme

In this Section we describe our main construction SPX. We start by giving a high-level overview of two of the main techniques we rely on. The first is how we index the DB to in order to handle HNF queries efficiently. The second is how we use the "chaining" technique from [16] to build complex encrypted structures from simpler ones.

Database indexing. The first step of our construction is to build different representations of the database, each designed to handle a particular operation of the SPC algebra. These representations are designed—when combined in an appropriate manner— to support the *efficient* processing of SPC queries. We use four representations. The first is a *row-wise* representation of the database instantiated as a multi-map MM_R that maps the coordinate of every row in the DB (recall that a coordinate is a row rank / table identifier pair) to the contents of the row. The second representation is a *column-wise* representation of the DB. Similarly, we create a multi-map MM_C that maps the coordinate of every column to the contents of that. The third representation, contrary to MM_R and MM_C, does not store any content of the table but the equality relation among values in the database. For this, we create a multi-map MM_V that maps each value in every column to all the rows that contain the same value. Finally, the fourth representation is a set of multi-maps, one for every column $\mathbf{c}$ in the DB. Each multi-map, $\mathsf{MM}_{\mathbf{c}}$, maps a pair of column coordinates to all the rows that have the same value in both those columns. Now, using multi-map and dictionary encryption schemes, we encrypt all these representations. This results in the encrypted multi-maps $\mathsf{EMM}_R, \mathsf{EMM}_C, \mathsf{EMM}_V$ and an encrypted dictionary EDX (which stores all the all EMM_c's).

Chaining and constructive queries. The different representations we just described are designed so that, given an SPC query, the server can generate the intermediate (encrypted) tables needed to produce the final (encrypted) result/table. To do this, the server will need to make further intermediate queries on these (intermediate) encrypted tables. This type of query evaluation is *constructive* in the sense that the intermediate and final encrypted tables are not the result of pre-processing at setup time but are constructed at query time by the server as a function of the query and the underlying DB. To handle this, we use the chaining technique of [16]. At a high level, the idea is to store query tokens for one encrypted structure as the responses of another encrypted structure. By carefully chaining the various encrypted multi-maps (EMMs) described above, we can handle constructive queries by first querying some subset of the EMMs to recover either tokens for EMMs further down the chain or encrypted content which we will use to populate intermediate tables. This process proceeds further down the chain until the final result/table is constructed.

Security and efficiency. The database representations we choose along with the careful chaining of their encryptions provide us a way to control both the efficiency and the security of scheme. While intermediate results/tables will vary depending on the query, the chaining sequence remains the same for any SPC query written in our heuristic normal form. The chaining sequence is important because it determines the leakage profile of the construction. We analyze the security of our scheme in black-box manner; that is, we provide a black-box leakage profile that is a function of the leakage profile of the underlying encrypted multi-map and encrypted dictionaries used. This allows us to isolate the leakage that is coming from the underlying building blocks and the leakage that is coming directly from our construction. This further enables us to reason about and decide which concrete instantiations to use as building blocks so that we can choose the kind of leakage/performance tradeoff that is most appropriate.

From an efficiency standpoint, we show that when SPX is instantiated with optimal-time encrypted multi-map and dictionary schemes, it can achieve optimal query complexity and linear storage complexity (in the size of the DB) under natural assumptions about the database.

5.1 (Plaintext) Database Indexing

As detailed above, SPX relies on several ideas and techniques. Some of these are cryptographic and some are not. To better explain these techniques we will progressively build our solution; starting with a naive plaintext algorithm for evaluating SPC queries and ending with a detailed description of SPX.

The naive SPC algorithm. The naive way to evaluate an SPC normal form query

$$\pi_{\mathsf{att}_1,\cdots,\mathsf{att}_h}\left([a_1] \times \cdots [a_f] \times \sigma_\Psi(\mathsf{T}_{i_1} \times \cdots \times \mathsf{T}_{i_t})\right)$$

on a database $\mathsf{DB} = (\mathsf{T}_1, \ldots, \mathsf{T}_n)$ is to first compute $\mathsf{R}_1 := \mathsf{T}_{i_1} \times \cdots \times \mathsf{T}_{i_t}$, then $\mathsf{R}_2 := \sigma_\Psi(\mathsf{R}_1)$, then $\mathsf{R}_3 := [a_1] \times \cdots \times [a_f] \times \mathsf{R}_2$ and finally $\mathsf{R} := \pi_{\mathsf{att}_1,\ldots,\mathsf{att}_h}(\mathsf{R}_3)$. This algorithm is dominated by the cross product computation which is $O(m^t \cdot \sum_{i=1}^t s_i)$, where $m = \max_{i=1}^t \|\mathsf{T}_i\|_r$ and $s_i = \|\mathsf{T}_i\|_c$. The exponential blowup in t is the main reason normal form SPC queries are never used in practice. In addition, since m is usually very large the naive algorithm is prohibitive even for small t.

The benefit of working with the SPC normal form is generality; that is, we can handle an entire class of queries by finding a solution for a single well-specified query form. The disadvantage, however, is that normal form queries take exponential time to evaluate even on a plaintext database.

Heuristic normal form (HNF). We show that certain optimizations can be applied to the SPC normal form so that its evaluation time only induces a multiplicative factor of $\sum_{i=1}^t s_i/h$ over the optimal evaluation time on a plaintext database. We refer to this new normal form as the *heuristic normal form.* In some cases, this multiplicative factor is a constant as it does not depend on the size of the result and, in such cases, the HNF evaluation is optimal. The idea is inspired by a query optimization heuristic from database theory which takes advantage of a distributive property between the select and cross product operators. For example, if the predicate $\Psi = \big(\mathsf{att}_1 = a_1 \wedge \cdots \wedge \mathsf{att}_\ell = a_\ell\big)$ is only composed of type-1 terms and if, for all $i \in [\ell]$, $\mathsf{att}_i \in \mathsf{T}_i$, and the number of terms in Ψ equals the number of tables in the cross product, $\ell = t$, then we have the identity

$$\sigma_\Psi\big(\mathsf{T}_1 \times \cdots \times \mathsf{T}_t\big) = \sigma_{\mathsf{att}_1=a_1}(\mathsf{T}_1) \times \cdots \times \sigma_{\mathsf{att}_t=a_t}(\mathsf{T}_t).$$

In the database literature this is known as "pushing selects through products" and, depending on the selectivity of the terms, it can greatly reduce the cost of the evaluation. We extend this approach to arbitrary conjunctive predicates which can have both type-1 and type-2 terms. Optimizing these queries is quite involved because the terms can have complex dependencies. In the following, we say that a query is *correlated* if its predicate Ψ satisfies any of the following properties: (1) two or more type-2 terms share a common attribute; (2) a type-1 and type-2 term share a common attribute; (3) the attributes of two or more type-2 terms are from the same table; and (4) the attributes from a type-1 and type-2 term are from the same table. We say that a query is *uncorrelated* if it is not correlated. For ease of exposition, we only describe here how to handle uncorrelated queries and treat the case of correlated queries in the full version of this work.

HNF for uncorrelated queries. If Ψ is uncorrelated, we process each term of Ψ and apply the following rules. Let φ be an empty query. If there are $p \geq 1$ type-1 terms $\mathsf{att}_1 = a_1, \ldots, \mathsf{att}_p = a_p$ from some table T, then we set

$$\varphi := \varphi \times \left(\sigma_{\mathsf{att}_1 = a_1}(\mathsf{T}) \cap \cdots \cap \sigma_{\mathsf{att}_p = a_p}(\mathsf{T})\right),$$

and remove these terms from Ψ. If the term has form $\mathsf{att}_1 = \mathsf{att}_2$ (i.e., is type-2), where att_1 and att_2 are from tables T_1 and T_2, respectively, then we set

$$\varphi := \varphi \times \sigma_{\mathsf{att}_1 = \mathsf{att}_2}(\mathsf{T}_1 \times \mathsf{T}_2).$$

Note that if att_1 and att_2 are from the same table T, then $\mathsf{T}_1 = \mathsf{T}_2 = \mathsf{T}$ above.

At the end of this rewriting process, we say that the query

$$\pi_{\mathsf{att}_1, \cdots, \mathsf{att}_h}\left([a_1] \times \cdots [a_f] \times \varphi\right)$$

is in the heuristic SPC normal form or simply the heuristic normal form.

Indexing. In database systems, select and project operations can be executed in one of two ways: with or without an index. In an unindexed execution, the database management system evaluates the operation using sequential scan. For example, to evaluate the operation $\sigma_{\mathsf{att}=a}(\mathsf{T})$, it scans the rows of T and returns the ones that satisfy $\mathsf{att} = a$. In an indexed execution, on the other hand, the database management system uses a pre-computed data structure (e.g., an index) to find the relevant rows in sub-linear time. Here, we give an overview of how one can index the database to support efficient heuristic normal form queries. Note that our indexing strategy is really designed so that we can support heuristic normal form queries on encrypted data (which we discuss below) so it is not necessarily the most natural way to index a plaintext database.

Given a database $\mathsf{DB} = (\mathsf{T}_1, \ldots, \mathsf{T}_n)$, we first create a multi-map MM_R that stores, for all $\mathbf{r} \in \mathsf{DB}$, the pair

$$\left(\chi(\mathbf{r}), \mathbf{r}\right).$$

In other words, the multi-map MM_R maps row coordinates to rows. We then create a second multi-map MM_C that maps column coordinates to columns. Following this, we build a third multi-map, MM_V, that maps every value/column pair $(v, \chi(\mathbf{c}))$ in the database to the coordinates of the rows that hold v in column $\mathbf{c}$. That is, for all columns $\mathbf{c} \in \mathsf{DB}^{\mathsf{T}}$ and all values $v \in \mathbf{c}$, MM_V stores the pair

$$\left(\left\langle v, \chi(\mathbf{c})\right\rangle, \left(\chi(\mathbf{r})\right)_{\mathbf{r} \in \mathsf{DB}_{\mathsf{att}(\mathbf{c}) = v}}\right).$$

Finally, we build a set of multi-maps for every column $\mathbf{c} \in \mathsf{DB}^{\mathsf{T}}$. More precisely, for all columns $\mathbf{c} \in \mathsf{DB}^{\mathsf{T}}$ we create the multi-map $\mathsf{MM}_{\mathbf{c}}$ which maps the coordinates of $\mathbf{c}$ and any other column $\mathbf{c}'$ that has the same domain as $\mathbf{c}$, to the coordinates of rows $\mathbf{r}$ and $\mathbf{r}'$ such that $\mathbf{r}[\mathbf{c}] = \mathbf{r}'[\mathbf{c}']$. More precisely, for all $\mathbf{c}' \in \mathsf{DB}^{\mathsf{T}}$ such that $\mathsf{dom}(\mathbf{c}') = \mathsf{dom}(\mathbf{c})$, $\mathsf{MM}_{\mathbf{c}}$ stores pairs

$$\left(\left\langle \chi(\mathbf{c}), \chi(\mathbf{c}') \right\rangle, \left(\chi(\mathbf{r}), \chi(\mathbf{r}')\right)_{(\mathbf{r},\mathbf{r}') \in \mathsf{DB}_{\mathsf{att}(\mathbf{c})=\mathsf{att}(\mathbf{c}')}}\right).$$

To speed up access to the multi-map $\mathsf{MM}_{\mathbf{c}}$, we store it in a dictionary DX. That is, for all $\mathbf{c} \in \mathsf{DB}^{\mathsf{T}}$, we set

$$\mathsf{DX}[\chi(\mathbf{c})] := \mathsf{MM}_{\mathbf{c}}.$$

Note that, in practice, we could store a pointer to $\mathsf{MM}_{\mathbf{c}}$ in the dictionary instead.

Indexed execution of HNF queries. We now show how to perform an *indexed* execution of heuristic normal form queries using these structures.[3] For clarity, we use a small database composed of two tables and a simple SQL query. We hope that this example clarifies some of the ideas behind our construction.

Recall that HNF queries have form

$$\pi_{\mathsf{att}_1,\cdots,\mathsf{att}_h}\Big([a_1] \times \cdots \times [a_f] \times \varphi\Big),$$

where $\varphi = \varphi_1 \times \cdots \times \varphi_d$ with each φ_i having form either $\sigma_{\mathsf{att}_1=a_1}(\mathsf{T}) \cap \cdots \cap \sigma_{\mathsf{att}_p=a_p}(\mathsf{T})$ or $\sigma_{\mathsf{att}_1=\mathsf{att}_2}(\mathsf{T}_1 \times \mathsf{T}_2)$. We process each φ_i and create a set R_i of rows as follows:

– **(Case 1)** If φ_i has form $\sigma_{\mathsf{att}_1=a_1}(\mathsf{T}) \cap \cdots \cap \sigma_{\mathsf{att}_p=a_p}(\mathsf{T})$ we recover for each term $\sigma_{\mathsf{att}_j=a_j}(\mathsf{T})$ a set R'_j by computing

$$\Big(\chi(\mathbf{r})\Big)_{\mathbf{r} \in \mathsf{DB}_{\mathsf{att}_j=a_j}} := \mathsf{MM}_V\left[\left\langle a_j, \chi(\mathsf{att}_j)\right\rangle\right]$$

and querying MM_R on each of the returned row coordinates. We then set

$$R_i = R'_1 \cap \cdots \cap R'_p.$$

– **(Case 2)** If φ_i has form $\sigma_{\mathsf{att}_1=\mathsf{att}_2}(\mathsf{T}_1 \times \mathsf{T}_2)$, we first compute $\mathsf{MM}_{\mathsf{att}_1} := \mathsf{DX}[\chi(\mathsf{att}_1)]$ and

$$\Big(\chi(\mathbf{r}_1), \chi(\mathbf{r}_2)\Big)_{(\mathbf{r}_1,\mathbf{r}_2) \in \mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2}} := \mathsf{MM}_{\mathsf{att}_1}\left[\left\langle \chi(\mathsf{att}_1), \chi(\mathsf{att}_2)\right\rangle\right].$$

Then we query MM_R on all of the returned row coordinates to produce a set

$$R_i := \Big\{\mathbf{r}_1 \times \mathbf{r}_2\Big\}_{(\mathbf{r}_1,\mathbf{r}_2) \in \mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2}}.$$

[3] In the full version of this work, we provide a concrete example that walks through our indexed HNF algorithm.

After processing $\varphi_1, \ldots, \varphi_d$, we compute a temporary table

$$\mathsf{S} := [a_1] \times \cdots \times [a_f] \times R_1 \times \cdots \times R_d.$$

We then consider the set of attributes in the project operation that are in tables that appear in the select operation. Specifically, this is the set:

$$I = \left\{ \mathsf{att} \in S : \mathsf{att} \in \bigcup_{j=1}^{t} \mathbb{S}(\mathsf{T}_{i_j}) \right\},$$

where $S \overset{def}{=} \{\mathsf{att}_1, \ldots, \mathsf{att}_h\}$. Suppose I has $z \geq 1$ elements which we denote $(\mathsf{att}^i_1, \ldots, \mathsf{att}^i_z)$.We compute

$$\mathsf{W} := \pi_{\mathsf{att}^i_1, \ldots, \mathsf{att}^i_z}(\mathsf{S}).$$

We then consider the attributes in the project operation that are not in the tables that appear in the select operation; that is, the set $O = S \setminus I$. Suppose O has $h - z$ elements which we denote $(\mathsf{att}^o_1, \ldots, \mathsf{att}^o_{h-z})$. For all $1 \leq j \leq h - z$, we compute

$$\mathbf{c}_j := \mathsf{MM}_c\big[\chi(\mathsf{att}^o_j)\big].$$

Finally, we generate the result table

$$\mathsf{R} := \mathbf{c}_1 \times \cdots \times \mathbf{c}_{h-z} \times \mathsf{W},$$

where the $\mathbf{c}_j$'s are viewed as single-column tables.

5.2 Detailed Construction

We now describe our SPX construction at a high-level. Due to space limitations, we defer the pseudo-code to the full version of this work. The scheme makes black-box use of a response-revealing multi-map encryption scheme $\Sigma_{\mathsf{MM}} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Get})$, of a response-revealing dictionary encryption scheme $\Sigma_{\mathsf{DX}} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Get})$, of a symmetric-key encryption scheme $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$. Note that encrypted multi-maps and dictionaries can be instantiated using a variety of schemes [19,16,30,12,11,35].

Overview. At a high-level, the Setup algorithm takes as input a database $\mathsf{DB} = (\mathsf{T}_1, \ldots, \mathsf{T}_n)$, creates the multi-maps MM_R, MM_C, MM_V, $\{\mathsf{MM}_{\mathbf{c}}\}_{\mathbf{c} \in \mathsf{DB}^\intercal}$ and the dictionary DX, as described above, and then encrypts each structure with the appropriate structured encryption scheme. The Token algorithm works by parsing the heuristic normal form query and generating appropriate tokens for each structure so as to enable the server to perform an indexed execution of the query (over encrypted data) as described in the previous paragraph.

Setup. The Setup algorithm takes as input a relational database $\mathsf{DB} = (\mathsf{T}_1, \ldots, \mathsf{T}_n)$ and indexes it as above. This results in three multi-maps MM_R, MM_V and MM_C and a dictionary DX that stores pointers to an additional set of multi-maps

$\{\mathsf{MM}_{\mathbf{c}}\}_{\mathbf{c}\in\mathsf{DB}^{\intercal}}$. The algorithm then encrypts every row $\mathbf{r}$ in MM_R using SKE. In other words, MM_R now holds value/tuple pairs of the form

$$\Big(\chi(\mathbf{r}), \Big(\mathsf{Enc}_{K_1}(r_1), \ldots, \mathsf{Enc}_{K_1}(r_{\#\mathbf{r}})\Big)\Big),$$

where $K_1 \xleftarrow{\$} \{0,1\}^k$. It then encrypts MM_R with Σ_{MM} which results in a key K_R and an encrypted multi-map EMM_R. It then encrypts every column $\mathbf{c}$ in MM_C using SKE in the same manner as above and encrypts MM_C with Σ_{MM}. This results in K_C and an encrypted multi-map EMM_C.

Now for all $\mathbf{r} \in \mathsf{DB}$, it replaces all occurrences of $\chi(\mathbf{r})$ in MM_V and $\{\mathsf{MM}_{\mathbf{c}}\}_{\mathbf{c}\in\mathsf{DB}^{\intercal}}$ with

$$\mathsf{rtk}_{\mathbf{r}} := \Sigma_{\mathsf{MM}}.\mathsf{Token}(K_R, \chi(\mathbf{r})).$$

It then encrypts MM_V and $\{\mathsf{MM}_{\mathbf{c}}\}_{\mathbf{c}\in\mathsf{DB}}$ with Σ_{MM} which results in keys K_V and $\{K_{\mathbf{c}}\}_{\mathbf{c}\in\mathsf{DB}^{\intercal}}$ and encrypted multi-maps EMM_V and $\{\mathsf{EMM}_{\mathbf{c}}\}_{\mathbf{c}\in\mathsf{DB}^{\intercal}}$. It then stores pairs $\big(\chi(\mathbf{c}), \mathsf{EMM}_{\mathbf{c}}\big)_{\mathbf{c}\in\mathsf{DB}^{\intercal}}$ in a dictionary DX and encrypts DX with Σ_{DX} which results in a key K_D and an encrypted dictionary EDX.

Finally, the Setup algorithms then outputs the key

$$K = (K_1, K_R, K_V, K_C, K_D, \{K_{\mathbf{c}}\}_{\mathbf{c}\in\mathsf{DB}^{\intercal}}),$$

and the encrypted database

$$\mathsf{EDB} = (\mathsf{EMM}_R, \mathsf{EMM}_C, \mathsf{EMM}_V, \mathsf{EDX}).$$

Token. The Token algorithm takes as input a secret key K and a query q in SPC normal form. It first transforms it in heuristic normal form:

$$\pi_{\mathsf{att}_1,\cdots,\mathsf{att}_h}\bigg([a_1] \times \cdots [a_f] \times \varphi_1 \times \cdots \times \varphi_d\bigg).$$

For all $i \in [h]$, if the project attribute att_i does not appear in $\varphi_1 \times \cdots \times \varphi_d$, the algorithm computes

$$\mathsf{ptk}_i := \Sigma_{\mathsf{MM}}.\mathsf{Token}\big(K_C, \chi(\mathsf{att}_i)\big),$$

and sets $\mathsf{ytk}_i = (\mathsf{ptk}_i, \mathtt{out})$; otherwise it sets

$$\mathsf{ptk}_i := \mathtt{pos}_i,$$

where $\mathtt{pos}_i \in \Big[\sum_{j=1}^{t} \|\mathsf{T}_{i_j}\|_c\Big]$ denotes the position of the attribute in the tables referenced in $\varphi_1 \times \cdots \times \varphi_d$. It then sets $\mathsf{ytk}_i = (\mathsf{ptk}_i, \mathtt{in})$.

For every constant a_1 through a_f it computes $\mathrm{e}_1 \leftarrow \mathsf{Enc}_{K_1}(a_1)$ through $\mathrm{e}_f \leftarrow \mathsf{Enc}_{K_1}(a_f)$. It then processes φ_1 through φ_d and for each φ_i it does the following:

– **(Case 1)** if φ_i has form $\sigma_{\mathsf{att}_1=a_1}(\mathsf{T}) \cap \cdots \cap \sigma_{\mathsf{att}_p=a_p}(\mathsf{T})$, it sets

$$\mathsf{stk}_i := (\mathsf{itk}_1, \ldots, \mathsf{itk}_p),$$

where, for all $j \in [p]$,

$$\mathsf{itk}_j := \Sigma_{\mathsf{MM}}.\mathsf{Token}\big(K_V, \langle a_j, \chi(\mathsf{att}_j)\rangle\big).$$

– **(Case 2)** if φ_i has form $\sigma_{\mathsf{att}_1=\mathsf{att}_2}(\mathsf{T}_1 \times \mathsf{T}_2)$ it sets $\mathsf{stk}_i := (\mathsf{dtk}_i, \mathsf{jtk}_i)$, where

$$\mathsf{dtk}_i := \Sigma_{\mathsf{DX}}.\mathsf{Token}\big(K_D, \chi(\mathsf{att}_1)\big)$$

and

$$\mathsf{jtk}_i := \Sigma_{\mathsf{MM}}.\mathsf{Token}\bigg(K_{\mathbf{c}}, \bigg\langle \chi(\mathsf{att}_1), \chi(\mathsf{att}_2) \bigg\rangle\bigg).$$

Finally, it outputs the token

$$\mathsf{tk} = \bigg(\big(\mathsf{ytk}_i\big)_{i\in[h]}, \big(e_i\big)_{i\in[f]}, \big(\mathsf{stk}_i\big)_{i\in[d]} \bigg).$$

Query. The Query algorithm works like the plaintext indexed HNF query evaluation algorithm we described above. Given a token

$$\mathsf{tk} = \bigg(\big(\mathsf{ytk}_i\big)_{i\in[z]}, \big(e_i\big)_{i\in[f]}, \big(\mathsf{stk}_i\big)_{i\in[d]} \bigg)$$

as input, it process the sub-tokens $(\mathsf{stk}_1, \ldots, \mathsf{stk}_d)$. For each stk_i it recovers a set of encrypted rows R_i as follows:

– **(Case 1)** if stk_i has form $(\mathsf{itk}_1, \ldots, \mathsf{itk}_p)$, then it recovers, for all $j \in [p]$, a set R'_j by first computing

$$(\mathsf{rtk}_1, \ldots, \mathsf{rtk}_s) := \Sigma_{\mathsf{MM}}.\mathsf{Get}(\mathsf{EMM}_V, \mathsf{itk}_j).$$

It then computes

$$\mathbf{ct}_1 := \Sigma_{\mathsf{MM}}.\mathsf{Get}(\mathsf{EMM}_R, \mathsf{rtk}_1), \ldots, \mathbf{ct}_s := \Sigma_{\mathsf{MM}}.\mathsf{Get}(\mathsf{EMM}_R, \mathsf{rtk}_s),$$

and sets $R'_j := \{\mathbf{ct}_1, \ldots, \mathbf{ct}_s\}$. Finally, it sets $R_i = R'_1 \cap \cdots \cap R'_p$.

– **(Case 2)** if stk_i has form $(\mathsf{dtk}_i, \mathsf{jtk}_i)$ it first computes

$$\mathsf{EMM}_{\mathbf{c}} := \Sigma_{\mathsf{DX}}.\mathsf{Get}(\mathsf{EDX}, \mathsf{dtk}_i)$$

and

$$\bigg((\mathsf{rtk}_1, \mathsf{rtk}'_1), \ldots, (\mathsf{rtk}_s, \mathsf{rtk}'_s) \bigg) := \Sigma_{\mathsf{MM}}.\mathsf{Get}(\mathsf{EMM}_{\mathbf{c}}, \mathsf{jtk}_i).$$

It then computes

$$\mathbf{ct}_1 := \Sigma_{\mathsf{MM}}.\mathsf{Get}(\mathsf{EMM}_R, \mathsf{rtk}_1), \ldots, \mathbf{ct}_s := \Sigma_{\mathsf{MM}}.\mathsf{Get}(\mathsf{EMM}_R, \mathsf{rtk}_s),$$

and

$$\mathbf{ct}'_1 := \Sigma_{\mathsf{MM}}.\mathsf{Get}(\mathsf{EMM}_R, \mathsf{rtk}'_1), \ldots, \mathbf{ct}'_s := \Sigma_{\mathsf{MM}}.\mathsf{Get}(\mathsf{EMM}_R, \mathsf{rtk}'_s).$$

Finally, it sets $R_i = \bigg\{ \mathbf{ct}_j \times \mathbf{ct}'_j \bigg\}_{j\in[s]}$.

After processing stk_1 through stk_d, it creates the temporary encrypted table

$$\mathsf{S} = \mathrm{e}_{a_1} \times \cdots \times \mathrm{e}_{a_f} \times R_1 \times \cdots \times R_d.$$

Let $(\mathsf{ytk}^i_1, \ldots, \mathsf{ytk}^i_z)$ be the ytk sub-tokens with form $(\mathsf{ptk}_i, \mathsf{in})$. It then computes

$$\mathsf{W} := \pi_{\mathsf{ptk}_1, \cdots, \mathsf{ptk}_z}\big(\mathsf{S}\big).$$

Let $(\mathsf{ytk}^o_1, \ldots, \mathsf{ytk}^o_{h-z})$ be the ytk sub-tokens with form $(\mathsf{ptk}_i, \mathsf{out})$. For all $i \in [h-z]$, it computes

$$\mathbf{ct}_i := \Sigma_{\mathsf{MM}}.\mathsf{Get}(\mathsf{EMM}_C, \mathsf{ptk}_i).$$

Finally, it generates the response table

$$\mathsf{R} := \mathbf{ct}_1 \times \cdots \times \mathbf{ct}_{h-z} \times \mathsf{W},$$

where the encrypted column $\mathbf{ct}_i$ is viewed as a single-column table.

Decryption. The Dec algorithm takes as input a secret key K and the response table R returned by the server and simply decrypts each cell of R.

5.3 Efficiency

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

Search complexity. Consider an SPC query written in its heuristic normal form

$$\pi_{\mathsf{att}_1, \cdots, \mathsf{att}_h}\Bigg([a_1] \times \cdots [a_f] \times \varphi_1 \times \cdots \times \varphi_d\Bigg).$$

We show in the full version of this work that the size of the result table over a plaintext database (in cells) is linear in

$$\#\mathsf{R} = h \cdot \big(m^{h-z} \cdot \prod_{i=1}^{d} \#R_i\big), \tag{1}$$

where $z = \#\Big\{\mathsf{att} \in S : \mathsf{att} \in \bigcup_{j=1}^{t} \mathbb{S}(\mathsf{T}_{i_j})\Big\}$ and $S \overset{def}{=} \{\mathsf{att}_1, \ldots, \mathsf{att}_h\}$, and R_i is the set of rows returned by the evaluation of the term φ_i.

Theorem 1. *If Σ_{DX} and Σ_{MM} are optimal dictionary and multi-map encryption schemes, then the time and space complexity of the* Query *algorithm presented in Section 5.2 is*

$$O\left(\frac{\#\mathsf{R}}{h} \cdot \sum_{i=1}^{t} s_i\right),$$

where h is the number of selected attributes, s_i the number of attributes of the ith table for all $i \in [t]$, and $\#\mathsf{R}$ is the size of the result table over plaintext database as shown in Eq. 1.

We defer the proof of Theorem 1 to the full version of this work.

Corollary 1. *If $h^{-1} \cdot \sum_{i=1}^{t} s_i$ is a constant in $\#\mathsf{R}$, then both time and space complexity are in $O(\#\mathsf{R})$, which is optimal.*

The corollary follows simply from Theorem 1. Optimality here refers to query complexity that is linear in the size of the response table, which is the minimum time needed to return it. This is similar to the SSE setting where optimal solutions are linear in the number of documents that hold the keyword.

Storage complexity. For a database $\mathsf{DB} = (\mathsf{T}_1, \ldots, \mathsf{T}_n)$, SPX produces four encrypted multi-maps EMM_R, EMM_C, EMM_V and EDX. For ease of exposition, we again assume each table has m rows. Finally, note that standard multi-map encryption schemes [19,30,12,11] produce encrypted structures with storage overhead that is linear in sum of the tuple sizes. Using such a scheme as the underlying multi-map encryption scheme, we have that EMM_R and EMM_C will be $O(\sum_{\mathbf{r}\in\mathsf{DB}} \#\mathbf{r})$ and $O\big(\sum_{\mathbf{c}\in\mathsf{DB}^\intercal} \#\mathbf{c}\big)$, 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 $O\big(\sum_{\mathbf{c}\in\mathsf{DB}^\intercal} \sum_{v\in\mathbf{c}} \#\mathsf{DB}_{\mathsf{att}(\mathbf{c})=v}\big)$ storage. EDX maps the coordinates of each column $\mathbf{c} \in \mathsf{DB}^\intercal$ to an encrypted multi-map $\mathsf{EMM}_{\mathbf{c}}$ which in turn 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, EDX will have size

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

Note that the expression will vary greatly depending on the number of columns in DB with the same domain. In the worst case, all columns will have a common domain and the expression will be a sum of $O\big(\big(\sum_i \|\mathsf{T}_i\|_c\big)^2\big)$ 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 EDX will be empty. In practice, however, we expect there to be some relatively small number of columns with common domains. In the full version of the paper, we provide a concrete example of the storage overhead of an encrypted database.

6 Black-Box Security and Leakage of SPX

We show that our construction is adaptively-secure with respect to a well-specified leakage profile. Part of the subtlety in our security analysis is that some of the leakage is "black-box" in the sense that it comes from the underlying building blocks and part of it is "non-black-box" in the sense that it comes directly from the SPX construction.

6.1 Setup leakage

The setup leakage of SPX captures what an adversary can learn before performing any query operation. The setup leakage of SPX is

$$\mathcal{L}_{\mathsf{S}}^{\mathsf{spx}}(\mathsf{DB}) = \Big(\mathcal{L}_{\mathsf{S}}^{\mathsf{dx}}(\mathsf{DX}), \mathcal{L}_{\mathsf{S}}^{\mathsf{mm}}\left(\mathsf{MM}_R\right), \mathcal{L}_{\mathsf{S}}^{\mathsf{mm}}\left(\mathsf{MM}_C\right), \mathcal{L}_{\mathsf{S}}^{\mathsf{mm}}\left(\mathsf{MM}_V\right)\Big),$$

where $\mathcal{L}_{\mathsf{S}}^{\mathsf{dx}}$ and $\mathcal{L}_{\mathsf{S}}^{\mathsf{mm}}$ are the setup leakages of Σ_{DX} and Σ_{MM}, respectively. If the latter are instantiated with standard encrypted multi-map constructions, the setup leakage of SPX will consist of the number of rows and columns in DB and the size of the dictionary. Note that standard encrypted dictionary constructions leak only the maximum size of the values they store so the size of the $\mathsf{EMM}_{\mathbf{c}}$'s will be hidden (up to the maximum size).

6.2 Query Leakage

The query leakage is more complex and is defined as follows,

$$\mathcal{L}_{\mathsf{Q}}^{\mathsf{spx}}(\mathsf{DB}, q) = \Big(\mathrm{XPP}(\mathsf{DB}, q), \mathrm{PrP}(\mathsf{DB}, q), \mathrm{SelP}(\mathsf{DB}, q)\Big),$$

where each individual pattern is described next.

Cross product. The first leakage pattern is the *cross product* pattern which is defined as

$$\mathrm{XPP}(\mathsf{DB}, q) = \Big\{ \big(|a_i|\big)_{i\in[f]} \Big\},$$

and includes the size of the virtual attributes.

Projection. The second leakage pattern is the *projection pattern* which is defined as

$$\mathrm{PrP}(\mathsf{DB}, q) = \Big(\mathcal{P}(\mathsf{att}_1), \ldots, \mathcal{P}(\mathsf{att}_h)\Big),$$

where

$$\mathcal{P}(\mathsf{att}_i) = \begin{cases} \Big(\mathtt{out}, \mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}\Big(\mathsf{MM}_C, \chi(\mathsf{att}_i)\Big), \big(|c_j|\big)_{j\in[\#\mathbf{c}_i]}, \mathrm{AccP}(\mathbf{c}_i)\Big) & \text{if } \mathsf{att}_i \in S \setminus I; \\ \Big(\mathtt{in}, \mathsf{att}_i\Big) & \text{otherwise,} \end{cases}$$

where $I = \Big\{ \mathsf{att} \in S : \mathsf{att} \in \bigcup_{j=1}^{t} \mathbb{S}(\mathsf{T}_{i_j}) \Big\}$ and $S \overset{def}{=} \{\mathsf{att}_1, \ldots, \mathsf{att}_h\}$, $\mathbf{c}_i \in \mathsf{DB}^{\mathsf{T}}$ denotes the column with attribute att_i and $\mathrm{AccP}(\mathbf{c}_i)$ indicates the access pattern, i.e., if and when the column $\mathbf{c}_i$ has been accessed before. PrP captures the leakage produced when the server queries MM_C and for every attribute att_i reveals whether the attribute was `in` or `out` of the set composed of the attributes in the predicate Ψ. If it is `out`, it also reveals the size of the items in the projected column and if and when this column has been accessed before. Notice that it also reveals the Σ_{MM} query leakage on the coordinates of the projected attribute. If the attribute is `in`, it just reveals the attribute.[4]

[4] To be more precise, it reveals *only* the position of the attribute in the heuristic normal form. The position, however, is independent of the attribute itself.

Selection. The third leakage pattern is the *selection pattern* which is defined as

$$\mathrm{SelP}(\mathsf{DB}, q) = \Big(\mathcal{Z}(\varphi_1), \ldots, \mathcal{Z}(\varphi_d)\Big).$$

If φ_i has form $\sigma_{\mathsf{att}_{i,1}=a_{i,1}}(\mathsf{T}) \cap \cdots \cap \sigma_{\mathsf{att}_{i,p_i}=a_{i,p_i}}(\mathsf{T})$, then $\mathcal{Z}(\varphi_i)$ is defined as

$$\mathcal{Z}(\varphi_i) = \Bigg(\text{case-1}, p_i, \Bigg(\mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}\Big(\mathsf{MM}_V, \Big\langle X_{i,j}, \chi(\mathsf{att}_{i,j})\Big\rangle\Big), \left\{\mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}\Big(\mathsf{MM}_R, \chi(\mathbf{r})\Big), \mathrm{AccP}(\mathbf{r})\right\}_{\mathbf{r}\in\mathsf{DB}_{\mathsf{att}_{i,j}=X_{i,j}}}\Bigg)_{j\in[p_i]}\Bigg),$$

where $\mathrm{AccP}(\mathbf{r})$ indicates whether the row $\mathbf{r}$ has been accessed before, and *case-1* refers to the first form of φ_i as introduced in Section 5.1. $\mathcal{Z}(\varphi_i)$ captures the leakage produced when the server queries MM_V and uses the resulting row tokens to then query MM_R. It reveals whether the selection term is of case-1, the Σ_{MM} query leakage on the constant a_j, and the coordinates of the attribute $\mathsf{att}_{i,j}$, for all $j \in [p_i]$ where p_i represents the number of attributes $\mathsf{att}_{i,j}$ that are in the same table T. In addition, it also leaks the Σ_{MM} query leakage on the coordinates of the rows in $\mathsf{DB}_{\mathsf{att}_{i,j}=a_{i,j}}$ as well as if and when they have been accessed before, for all $j \in [p_i]$.

If, on the other hand, φ_i has form $\sigma_{\mathsf{att}_{i,1}=\mathsf{att}_{i,2}}(\mathsf{T}_{i,1} \times \mathsf{T}_{i,2})$, then $\mathcal{Z}(\varphi_i)$ is defined as

$$\begin{aligned}\mathcal{Z}(\varphi_i) = \Bigg(&\text{case-2}, \mathcal{L}^{\mathsf{dx}}_{\mathsf{Q}}\Big(\mathsf{DX}, \chi(\mathsf{att}_{i,1})\Big), \mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}(\mathsf{MM}_{\mathsf{att}_{i,1}}), \mathrm{AccP}(\mathsf{EMM}_{\mathsf{att}_{i,1}}),\\ &\mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}\Big(\mathsf{MM}_{\mathsf{att}_{i,1}}, \Big\langle \chi(\mathsf{att}_{i,1}), \chi(\mathsf{att}_{i,2})\Big\rangle\Big), \Big\{\mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}\Big(\mathsf{MM}_R, \chi(\mathbf{r}_1)\Big),\\ &\mathrm{AccP}(\mathbf{r}_1), \mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}\Big(\mathsf{MM}_R, \chi(\mathbf{r}_2)\Big), \mathrm{AccP}(\mathbf{r}_2)\Big\}_{(\mathbf{r}_1,\mathbf{r}_2)\in\mathsf{DB}_{\mathsf{att}_{i,1}=\mathsf{att}_{i,2}}}\Bigg),\end{aligned}$$

where $\mathrm{AccP}(\mathbf{r}_1)$, $\mathrm{AccP}(\mathbf{r}_2)$ and $\mathrm{AccP}(\mathsf{EMM}_{\mathsf{att}_i})$ indicate if and when $\mathbf{r}_1$, $\mathbf{r}_2$ and $\mathsf{EMM}_{\mathsf{att}_{i,1}}$ have been accessed before, and *case-2* refers to the second form of φ_i as introduced in Section 5.1. In this case, $\mathcal{Z}(\varphi_i)$ captures the leakage produced when the server queries EDX to retrieve some $\mathsf{EMM}_{\mathsf{att}_{i,1}}$ which it in turn queries to retrieve row tokens with which to query EMM_R. It reveals whether the selection term is of case-2, the Σ_{DX} query leakage on the coordinates of $\mathsf{att}_{i,1}$, the Σ_{MM} *setup* leakage on $\mathsf{MM}_{\mathsf{att}_{i,1}}$ and if and when $\mathsf{EMM}_{\mathsf{att}_{i,1}}$ has been accessed in the past. In addition, it reveals the query leakage of Σ_{MM} on the coordinates of $\mathsf{att}_{i,1}$ and $\mathsf{att}_{i,2}$ and, for every pair of rows $(\mathbf{r}_1, \mathbf{r}_2)$ in $\mathsf{DB}_{\mathsf{att}_{i,1}=\mathsf{att}_{i,2}}$, their Σ_{MM} query leakage and if and when they were accessed in the past.

6.3 Black-Box Security of SPX

We show that SPX is adaptively semantically-secure with respect to the leakage profile described in the previous sub-section.

Theorem 2. *If* SKE *is RCPA secure,* Σ_{DX} *is adaptively* $\left(\mathcal{L}_{\mathsf{S}}^{\mathsf{dx}}, \mathcal{L}_{\mathsf{Q}}^{\mathsf{dx}}\right)$*-semantically secure and* Σ_{MM} *is adaptively* $\left(\mathcal{L}_{\mathsf{S}}^{\mathsf{mm}}, \mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}\right)$*-secure, then* SPX *is* $(\mathcal{L}_{\mathsf{S}}^{\mathsf{spx}}, \mathcal{L}_{\mathsf{Q}}^{\mathsf{spx}})$*-semantically secure.*

The proof of Theorem 2 is in the full version of the paper.

7 Concrete Security and Leakage of SPX

7.1 With Zero-Leakage Building Blocks

Here, we are interested in the leakage profile of SPX when the underlying building blocks are ZL. By a ZL encrypted structure, we mean that its query operations only reveals information that can be derived from the security parameter or other public parameters. We write this as $\mathcal{L}_{\mathsf{Q}}(\mathsf{DS}, q) = \perp$, for any query q in its corresponding query space. When instantiated with ZL building blocks, the query leakage of SPX decreases considerably but its setup leakage remains the same. Specifically, the projection pattern becomes $\mathrm{PrP}(\mathsf{DB}, q) = \Big(\mathcal{P}(\mathsf{att}_1), \ldots, \mathcal{P}(\mathsf{att}_h)\Big)$, where

$$\mathcal{P}(\mathsf{att}_i) = \begin{cases} \Big(\mathtt{out}, \big(|c_j|\big)_{j\in[\#\mathbf{c}_i]}, \mathrm{AccP}(\mathbf{c}_i)\Big) & \text{if } \mathsf{att}_i \in S \setminus I; \\ \Big(\mathtt{in}, \mathsf{att}_i\Big) & \text{otherwise.} \end{cases}$$

The selection pattern SelP becomes $\mathrm{SelP}(\mathsf{DB}, q) = \Big(\mathcal{Z}(\varphi_1), \ldots, \mathcal{Z}(\varphi_d)\Big)$, where if φ_i has form $\sigma_{\mathsf{att}_{i,1}=a_{i,1}}(\mathsf{T}) \cap \cdots \cap \sigma_{\mathsf{att}_{i,p_i}=a_{i,p_i}}(\mathsf{T})$, then $\mathcal{Z}(\varphi_i)$ is defined as

$$\mathcal{Z}(\varphi_i) = \Bigg(\text{case-1}, p_i, \Big\{\mathrm{AccP}(\mathbf{r})\Big\}_{\mathbf{r}\in \mathsf{DB}_{\mathsf{att}_{i,j}=X_{i,j}}, j\in[p_i]}\Bigg),$$

Otherwise if, φ_i has form $\sigma_{\mathsf{att}_{i,1}=\mathsf{att}_{i,2}}(\mathsf{T}_{i,1} \times \mathsf{T}_{i,2})$, then $\mathcal{Z}(\varphi_i)$ is defined as

$$\begin{aligned}\mathcal{Z}(\varphi_i) = \Bigg(&\text{case-2}, \mathcal{L}_{\mathsf{S}}^{\mathsf{mm}}(\mathsf{MM}_{\mathsf{att}_{i,1}}), \mathrm{AccP}(\mathsf{EMM}_{\mathsf{att}_{i,1}}), \\ &\Big\{\mathrm{AccP}(\mathbf{r}_1), \mathrm{AccP}(\mathbf{r}_2)\Big\}_{(\mathbf{r}_1,\mathbf{r}_2)\in \mathsf{DB}_{\mathsf{att}_{i,1}=\mathsf{att}_{i,2}}}\Bigg).\end{aligned}$$

We are aware of two ZL encrypted multi-map constructions. The first can be derived from an SSE construction of Garg, Mohassel and Papamanthou [20] that itself is based on the TWORAM construction. We note that the SSE scheme proposed in that work is not ZL (since it reveals the response length) but it can be made so with a careful parametrization of its block size. The second construction is FZL by Kamara, Moataz and Ohromenko [29]. Of course, ZL schemes come with an additional efficiency overhead. For example, if the TWORAM-based construction is used in SPX its time and space complexity would incur an additive overhead of

$$\widetilde{O}\Bigg((2\ell + h) \cdot m \cdot \log (n \cdot m) + d \cdot m^2 \cdot \log \Bigg(\sum_{i \in [n]} \|\mathsf{T}_i\|_c \cdot m \Bigg)\Bigg),$$

where n is the number of tables in DB. Note that SPX becomes interactive if it is instantiated with any of the currently-known ZL constructions.

7.2 With Standard Building Bocks

In this section, we describe the leakage profile of SPX when instantiated with encrypted dictionary or multi-map schemes with the "standard" leakage profile [19,30,16,12,11,35]. A standard response-revealing encrypted multi-map or dictionary encryption reveals the search pattern SP and access pattern AP, whereas a standard response-hiding encrypted multi-map or dictionary reveals the search pattern SP and the response length RL. The search pattern reveals if and when a query is repeated, the access pattern reveals the responses, and the response length reveals the length of the response. The query leakage of SPX when instantiated with standard STE schemes is the one detailed in Section 6.2 except that we replace $\mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}$ with the patterns detailed above depending on whether the underlying scheme is response-revealing or response-hiding. In the following, we provide a high level description of both the projection and selection patterns of SPX.

Projection. The projection pattern discloses the frequency of accesses made to a particular attribute. An adversary can learn the size of the accessed columns, and therefore the number of entries that a specific table has. The impact of such leaked information depends on the auxiliary information the attacker possesses. In some settings, just knowing the size of the table can be sufficient for an adversary to know the targeted information, but this is a general problem that can be addressed by padding, for instance.

Selection. Of all the leakage patterns of SPX, the selection pattern is the one that leaks the most. If φ_i is of case-1, then an adversary can know the number of rows that contain the same value at a particular column(s), and this applies to all the p_i attributes in φ_i. The adversary can also learn 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. Otherwise if φ_i is of case-2, then the server learns how many rows are equal to each other in both columns.

7.3 SPX and SSE Attacks

As mentioned, *one* of the possible instantiation SPX makes use of standard SSE to implement the underlying encrypted multi-maps. There are several known attacks that try to exploit the leakage of various SSE schemes such as the inference attacks of Islam et al. [26] and of Cash et al. [10] and the file injection attacks of Cash et al. [10] and Zhang et al. [40]. It is not exactly clear what the impact of these attacks would be to our setting since our construction handles more complex objects and has a different leakage profile than standard SSE schemes. What is clear, however, is that our scheme leaks *more* than standard SSE schemes so presumably the techniques from these works could be extended to apply to our construction.

We note, however, that the attacks in [26,10] rely on strong assumptions including knowledge of a large fraction of the client's data and knowledge of some client queries [5]. Specifically, for IKK, the adversary needs to know about 90% of the client's data in order to recover about 10% of its queries. Similarly, the Count attack from [10] requires the adversary to know 80% of the client's data and 5% of its queries in order to recover 40% of the client's queries (note that the success rate of the counting attack is not linear so knowing even 75% of the client's data is not enough for the adversary to learn even 1% of the client's queries). With 90% of the data and 2% of the queries, the Count attack does not work at all. Another limitation of these attacks is related to how the adversary can recover client data in practice. Recall that in an outsourced storage setting the client is assumed to erase its data after storing it in encrypted form on the server (that is the purpose of outsourcing). It is therefore not clear how the adversary can recover, say 80%, of client data unless the client encrypts publicly-known data—in which case it should use a different primitive like private information retrieval. In a model where the adversary does not know any of the client's data a-priori—which is the standard model for SSE and structured encryption—neither the IKK attack nor the Count attack can recover *any* queries at all.

Unlike the previously mentioned attacks, the file injection attacks of [40] are effective in practice but are only applicable against dynamic SSE schemes and in scenarios where the adversary can inject data into the encrypted structure. This is the case, for example, if one were to use a dynamic SSE scheme to encrypt an email archive since the server/adversary could send the client malicious emails. In our setting, we assume the data is generated by the client and is not publicly modifiable *after the setup*. However, if our dynamic scheme SPX^+ were used in a setting where row injections are possible then, presumably, attacks like those of [40] could be designed and some queries could be disclosed. As suggested in [40], one countermeasure in this case is to use forward-secure constructions. In the full version of this work, we discuss how to make SPX^+ forward-secure.

Recently, Abdelraheem et al. [33] presented an attack on relational databases encrypted with SSE. We stress, however, that the attack of [33] only applies to a very specific and naive SSE-based relational EDB construction described in that work and first used for experiments in [12] (e.g., the construction does not handle any non-trivial SQL query). While it is not clear at all how this attack would apply to our construction, we point out that the attack relies on strong assumptions. In particular, it works only for databases with attributes whose domain sizes are unique. In addition, it relies on the adversary knowing the attributes in the database and their domain sizes. Furthermore, the adversary

[5] While the Count attack is not described as a known-query attack in [10], it has come to our attention that this was an error and will be fixed by the authors.

also needs to know, for each attacked column, which domain element appears the most frequently, the second most frequently etc. Finally, the attack needs to solve an NP-complete problem that can be solved in pseudo-polynomial time only for databases with a small number of rows and small attribute domains (experimental results were conducted for databases with $32,561$ rows and domain sizes that range from 2 to 41 and execution times were not reported).

7.4 Comparison to PPE-based Solutions

As mentioned in Section 1, PPE-based solutions can handle a large class of SQL queries which includes conjunctive queries. To support conjunctive queries, however, these solutions have to rely on deterministic encryption. For example, to handle a case-1 query on a table T, they will reveal a deterministic encryptions of all the accessed attributes $\mathbf{c}$ in T (i.e., every element of every column is encrypted under the same key). To handle a case-2 query between two columns $\mathbf{c}_1$ and $\mathbf{c}_2$, they will reveal deterministic encryptions of both columns (under the same key). In turn, this will provide the *frequency* information on the entire columns to the server. Depending on the setting, frequency patterns can be particularly dangerous, as shown in [34].

SPX leaks considerably less. First, it does not leak any frequency information on entire columns or rows. For case-1 queries, it only leaks information about the attributes in the query and the rows that match the term. For case-2 queries, it only leaks information about the pair of attributes $(\mathsf{att}_{i,1}, \mathsf{att}_{i,2})$ in the select and the rows that match the term. Note that this leakage is only a function of the attributes in the query and of the rows that match it, whereas the leakage in PPE-based solutions is a function of entire columns. Moreover, in the case of SPX, if the underlying multi-map and dictionary schemes are instantiated with standard constructions, the information leaked about the attributes and matching rows is "repetition" type of information, i.e., if and when they have appeared in the past. Analogously, the project operations in SPX only leak information about the attributes in the project and the columns that match it and the information being leaked "repetition" type of information.

Formally, the setup leakage of PPE-based solutions like CryptDB is

$$\mathcal{L}_{\mathsf{S}}^{\mathsf{ppe}}(\mathsf{DB}) = \bigg(\|\mathsf{T}_i\|_c, \|\mathsf{T}_i\|_r\bigg)_{i\in[n]},$$

where n is the number of tables in DB. Given a SQL query q, the query leakage is

$$\mathcal{L}_{\mathsf{Q}}^{\mathsf{ppe}}(\mathsf{DB}, q) = \bigg(\mathrm{XPP}(\mathsf{DB}, q), \mathrm{PrP}(\mathsf{DB}, q), \mathrm{SelP}(\mathsf{DB}, q), \mathrm{FrP}(\mathsf{DB}, q)\bigg),$$

where XPP, PrP and SelP are the cross product, projection and selection patterns (defined as in the leakage profile of SPX), and $\mathrm{FrP}(\mathsf{DB}, q)$ is the frequency pattern which leaks frequency information on all queried columns. It is easy to see that even when SPX is instantiated with non-ZL building blocks, its query leakage is a subset of the query leakage of the PPE-based solutions. Note that, not only is FrP relatively easy to exploit [34], it is also *persistent* in the sense that it is available not only to an adversary that has the query tokens and witnesses or executes the query operation but also to a "snapshot" adversary which only has access to the encrypted DB. This is not the case for SPX.

A remark on leakage. Ideally, one would hope to better understand how significant the leakage of practical encrypted search solutions are but we currently lack any theoretical framework to conduct such an analysis. In other words, the best we can currently do is to give a precise leakage profile and prove that our constructions do not leak anything beyond that profile. For the same reason, the best we can currently do to compare two leakage profiles is to show that one is a subset of the other (and in some cases, this is not even possible).

8 Extensions

In the full version of the paper, we show how to extend SPX to handle additional post-processing operations including Group-by, Order-by and various aggregate functions such as Sum, Average, Median, Count, Mode, Max and Min.

In addition, due to its modularity, SPX can be extended to be dynamic without re-designing it entirely. We refer to the dynamic version of SPX as SPX^+ and describe it in the full version of this work. Note that SPX^+ maintains the same query complexity and query leakage as SPX. We also discuss how to use ORAM to make SPX^+ forward-secure at the cost of a poly-logarithmic overhead for updates and without affecting the query complexity.

9 Future Directions and Open Problems

In this work, we proposed the first encrypted relational database scheme purely based on STE techniques. As such, our construction offers more security than the PPE-based solutions and are more efficient than solutions based on general-purpose techniques like ORAM simulation or FHE. Our work leaves open several interesting questions. The first is whether our techniques can be extended to handle the *full* relational algebra which, effectively, is the entire SQL. To achieve this, our solution would have to be extended to handle negations and disjunctions (set unions) in the Where clause of the SQL query. We believe this to be challenging. A second problem is to handle SQL queries with ranges in the Where clause. A first step towards achieving this would be to improve the state of the art in encrypted range queries. In particular, finding schemes with improved leakage profiles is important since recent work [31,32] has described powerful attacks against the state of the art encrypted search solutions (under some assumptions on the data and queries).

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases: the logical level.* Addison-Wesley Longman Publishing Co., Inc., 1995.
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. G. Asharov, M. Naor, G. Segev, and I. Shahaf. Searchable symmetric encryption: Optimal locality in linear space via two-dimensional balanced allocations. In *ACM on Symposium on Theory of Computing (STOC '16)*, 2016.
5. M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and efficiently searchable encryption. In A. Menezes, editor, *Advances in Cryptology – CRYPTO '07*, 2007.
6. A. Ben-David, N. Nisan, and B. Pinkas. Fairplaymp: a system for secure multi-party computation. In *ACM Conference on Computer and Communications Security (CCS 2008)*, pages 257–266. ACM, 2008.
7. A. Boldyreva, N. Chenette, Y. Lee, and A. O'neill. Order-preserving symmetric encryption. In *Advances in Cryptology - EUROCRYPT 2009*, pages 224–241, 2009.
8. D. Boneh, G. di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Advances in Cryptology – EUROCRYPT '04*, 2004.
9. R. Bost. Sophos - Forward Secure Searchable Encryption. In *ACM Conference on Computer and Communications Security (CCS '16)*, 2016.
10. D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *ACM Conference on Communications and Computer Security (CCS '15)*, pages 668–679. ACM, 2015.
11. 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.
12. 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.
13. D. Cash and S. Tessaro. The locality of searchable symmetric encryption. In *Advances in Cryptology - EUROCRYPT 2014*, 2014.
14. D. D. Chamberlin and R. F. Boyce. SEQUEL: A structured english query language. In *(SIGMOD '74)*, 1974.
15. A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *(STOC '77)*, 1977.
16. M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *Advances in Cryptology - ASIACRYPT '10*, volume 6477, 2010.
17. M. Chase and S. Kamara. Structured encryption and controlled disclosure. Technical Report 2011/010.pdf, IACR Cryptology ePrint Archive, 2010.
18. E. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.
19. 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.
20. 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*, 2016.
21. C. Gentry. *A fully homomorphic encryption scheme.* PhD thesis, Stanford University, 2009.
22. O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.
23. P. Grofig, M. Haerterich, I. Hang, F. Kerschbaum, M. Kohler, A. Schaad, A. Schroepfer, and W. Tighzert. Experiences and observations on the industrial implementation of a system to search over outsourced encrypted data. In *Sicherheit*, pages 115–125, 2014.

24. P. Grubbs, T. Ristenpart, and V. Shmatikov. Why your encrypted database is not secure. In *HotOS*, 2017.
25. H. Hacigümücs, B. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *(SIGMOD'02)*, 2002.
26. M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Network and Distributed System Security Symposium (NDSS '12)*, 2012.
27. S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Outsourced symmetric private information retrieval. In *ACM Conference on Computer and Communications Security (CCS '13)*, pages 875–888, 2013.
28. S. Kamara and T. Moataz. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In *Advances in Cryptology - EUROCRYPT '17*, 2017.
29. S. Kamara, T. Moataz, and O. Ohrimenko. Structured encryption and leakage suppression. In *Advances in Cryptology – CRYPTO 2018*, pages 339–370, 2018.
30. S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *ACM Conference on Computer and Communications Security (CCS '12)*. ACM Press, 2012.
31. G. Kellaris, G. Kollios, K. Nissim, and A. O. Neill. Generic attacks on secure outsourced databases. In *ACM Conference on Computer and Communications Security (CCS '16)*, 2016.
32. M. Lacharité, B. Minaud, and K. G. Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *2018 IEEE Symposium on Security and Privacy, SP*, pages 297–314, 2018.
33. T. A. Mohamed Ahmed Abdelraheem and C. Gehrmann. Inference and record-injection attacks on searchable encrypted relational databases. Technical Report 2017/024, 2017. http://eprint.iacr.org/2017/024.
34. 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.
35. M. Naveed, M. Prabhakaran, and C. Gunter. Dynamic searchable encryption via blind storage. In *IEEE Symposium on Security and Privacy (S&P '14)*, 2014.
36. 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.
37. R. A. 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.
38. D. Song, D. Wagner, and A. Perrig. Practical techniques for searching on encrypted data. In *IEEE Symposium on Research in Security and Privacy*, 2000.
39. J. Van den Bussche and S. Vansummeren. Translating sql into the relational algebra. 2009. http://cs.ulb.ac.be/public/_media/teaching/infoh417/sql2alg_eng.pdf.
40. Y. Zhang, J. Katz, and C. Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security Symposium*, 2016.