Framework for Searchable Encryption with SQL Databases

Monir Azraoui, Melek Önen and Refik Molva
EURECOM, Sophia-Antipolis, France

Keywords: Searchable Encryption, SQL Database, Data Confidentiality.

Abstract: In recent years, the increasing popularity of outsourcing data to third-party cloud servers sparked a major concern towards data breaches. A standard measure to thwart this problem and to ensure data confidentiality is data encryption. Nevertheless, organizations that use traditional encryption techniques face the challenge of how to enable untrusted cloud servers perform search operations while the actually outsourced data remains confidential. Searchable encryption is a powerful tool that attempts to solve the challenge of querying data outsourced at untrusted servers while preserving data confidentiality. Whereas the literature mainly considers searching over an unstructured collection of files, this paper explores methods to execute SQL queries over encrypted databases. We provide a complete framework that supports private search queries over encrypted SQL databases, in particular for PostgreSQL and MySQL databases. We extend the solution for searchable encryption designed by Curtmola et al., to the case of SQL databases. We also provide features for evaluating range and boolean queries. We finally propose a framework for implementing our construction, validating its practicality.

1 INTRODUCTION

Outsourcing data storage and processing to third-party servers, such as cloud servers, has become a common procedure. However, when it comes to storing personal or business confidential data, the usage of cloud services is currently questioned as as potentially malicious cloud servers may try to learn information from the data they store and the queries they process or even leak them to some unauthorized parties. Encryption is a standard approach to ensure the confidentiality of data outsourced at *honest-but-curious* cloud servers. However, "traditional" encryption schemes deprive the users of functionalities over the data, such as searching. In this case, to search words in encrypted outsourced data, the users are required to retrieve the entire dataset from the cloud, decrypt it and operate the search locally. This naive approach poses serious performance concerns which cancel out the benefit of outsourcing. Therefore, cloud servers should be able to perform the search operation in outsourced databases, while the actual databases are encrypted.

The problem of searching over encrypted data has received much interest from both academia (Song et al., 2000; Hacigümüş et al., 2002; Agrawal et al., 2004; Curtmola et al., 2006; Chase and Kamara, 2010; Popa et al., 2012; Cash et al., 2013; Kamara and Papamanthou, 2013) and industry (Bitglass[1], CipherCloud[2] or Skyhigh[3]). Research on this topic mainly focuses on the scenario of a user who outsources an encrypted collection of documents (such as e-mails, medical records, etc.) and would like to further search keywords over this encrypted dataset. While this theoretical setting is valid, in practice, many organizations such as governments, hospitals or companies store data in *relational databases* which structure data into tables according to a set of attributes. The popular SQL language (Chamberlin and Boyce, 1974) enables users to store, query and update their data in a user-friendly manner. SQL databases ensure fast record search and retrieval provided that the SQL server is able to read data content. As mentioned before, encryption typically impedes the server from reading data which makes search over encrypted databases challenging. In particular, a cryptographic data protection mechanism for searching over encrypted data stored in a SQL database should allow the server to efficiently process search queries without having access to the plaintext data. Besides, querying an encrypted SQL database should be user-friendly: protected que-

[1]https://www.bitglass.com/ [Accessed Oct. 09, 2017].
[2]https://ciphercloud.com/ [Accessed Oct. 09, 2017].
[3]https://www.skyhighnetworks.com/ [Accessed Oct. 09, 2017].

Azraoui, M., Önen, M. and Molva, R.
Framework for Searchable Encryption with SQL Databases.
In *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018)*, pages 57-67
ISBN: 978-989-758-295-0

ries should adopt the SQL syntax and simulate the search functionality as if data was not encrypted. A direct application of solutions for searching an encrypted collection of files to SQL databases is not straightforward. Indeed, existing solutions (Curtmola et al., 2006; Chase and Kamara, 2010; Cash et al., 2013; Kamara and Papamanthou, 2013) build an index of the keywords. In the case of SQL, data is arranged into tables, and records are queried based on a condition over one or more attributes. Therefore, keywords should preserve this notion of attribute. Furthermore, SQL allows performing comparisons of data (range queries) which are not always addressed in existing work.
To solve these challenges, in (Popa et al., 2012), the authors proposed CryptDB, a framework that supports SQL queries over encrypted data. This solution relies on layers of different property-preserving encryptions, such as deterministic (DET) and order-preserving encryption (OPE), applied to a column of an SQL table. To query the encrypted database, CryptDB converts the plaintext SQL query into its encrypted equivalent and decrypts the targeted layers. The major drawback of CryptDB lies in the fact that whenever one layer is removed, the encryption scheme becomes weak. In light of this, the major problem is to deliver a practical solution for searching over encrypted databases that does not suffer from the leakage occurring in CryptDB and that enables transparent processing of complex queries over encrypted SQL databases.

In this work, we tackle this problem and propose a practical construction for searching over encrypted SQL databases which limits information leakage. Our solution builds upon the searchable encryption technique designed in (Curtmola et al., 2006) which applies to unstructured documents. This mechanism builds an inverted search index of keywords in the database to enable keyword search queries over encrypted data. We make the following contributions:

- We define a solution that is SQL-compatible, so as to empower user and server to use SQL's operations of database creation, data insertion and search. We propose a framework for implementing our solution on the PostgreSQL backend;
- Our solution achieves practicality thanks to the use of the cuckoo hashing technique which renders the search in the index efficient;
- Our solution supports complex queries, namely boolean and range queries;
- We evaluate the practicality of our solution by testing it over an encrypted e-health database.

The rest of the paper is organized as follows. Section 2 defines searchable encryption and SQL databases. We describe our solution in Section 3. Section 4 presents the proposed implementation framework and a performance evaluation. We review existing work in Section 5. We conclude the paper and give some direction for future work in Section 6.

2 PRELIMINARIES

2.1 Symmetric Searchable Encryption

We consider the following scenario: a user $\mathcal{U}$ wants to store a set $\mathcal{D} = \{d_1, d_2, ..., d_n\}$ of confidential data at a server $\mathcal{S}$. $\mathcal{D}$ contains a list of searchable keywords $\mathbb{W} = \{\omega_1, \omega_2, ..., \omega_N\}$. To preserve data confidentiality, $\mathcal{U}$ encrypts $\mathcal{D}$ to obtain $\mathcal{C} = \{c_1, c_2, ..., c_n\}$, which is outsourced to $\mathcal{S}$. A symmetric searchable encryption mechanism allows searching the keywords directly in $\mathcal{C}$, without compromising data confidentiality nor query privacy. In (Curtmola et al., 2006), the authors formally define symmetric searchable encryption (SSE) schemes by the following algorithms:

$\mathsf{KeyGen}(1^\kappa) \rightarrow \mathcal{K}$: It is a probabilistic algorithm executed by user $\mathcal{U}$ and, given a security parameter κ, outputs a secret key $\mathcal{K}$.

$\mathsf{BuildIndex}(\mathcal{D}, \mathcal{K}) \rightarrow I$: This algorithm is run by user $\mathcal{U}$ to generate a search index. It takes key $\mathcal{K}$ and dataset $\mathcal{D}$ as inputs and outputs index I. When the execution of $\mathsf{BuildIndex}$ is completed, index I is stored at server $\mathcal{S}$.

$\mathsf{Trapdoor}(\omega, \mathcal{K}) \rightarrow \tau_\omega$: This algorithm, executed by user $\mathcal{U}$, issues a search token, called *trapdoor*, to search for a given keyword ω. It takes secret key $\mathcal{K}$ and keyword ω as inputs and generates τ_ω. This trapdoor allows the cloud to perform the search over the encrypted database for the corresponding keyword ω.

$\mathsf{Search}(I, \tau_\omega) \rightarrow \mathcal{D}(\omega)$: Run by server $\mathcal{S}$, this algorithm processes the search query on ω and generates the search result. It takes index I and trapdoor τ_ω as inputs and outputs $\mathcal{D}(\omega)$, the list of identifiers of documents containing ω.

2.2 SQL Databases

Our work focuses on data structured in SQL databases such as MySQL[4] or PostgreSQL[5]. An SQL database $\mathcal{D}$ organizes the data in a set of two-dimensional *tables* $(T_1, T_2, .., T_n)$. Each table row is a *record*, the columns are *attributes*, and each table entry a *value*.

[4]https://www.mysql.com/ [Accessed Oct. 09, 2017].
[5]https://www.postgresql.org/ [Accessed Oct. 09, 2017].

We assume the existence of a *view* V, resulting from a query applied to the data in $\mathcal{D}$, which represents a subset of the records, selected from different tables in $\mathcal{D}$, in such a way that queries can directly be processed on V. We mainly consider the `SELECT` operation that can be expressed as follows:

`SELECT attributes FROM table WHERE conditions.`

This query retrieves from an SQL table (or a view) the records, and more precisely, the attributes specified after `SELECT`, that match `conditions`. In this work, we are interested in conditions that can be written as a boolean expression $Q : r \rightarrow \{0,1\}$, which, evaluated on a record r of $\mathcal{D}$, returns $Q(r) = 1$, if r satisfies the expression or $Q(r) = 0$ otherwise. Q consists of several predicates P_i that express conditions over one or more attributes in $\mathcal{D}$ and are combined with boolean operators: `OR` and `AND`. Each of the P_i involves operators such as $=, >, <, \geq, \leq$ and relates a database attribute to a search criterion.

Table 1: Example of table: `PATIENT`.

ID	Name	Surname	Age	Gender	Occupation
001	David	Smith	38	M	Farmer
002	Mary	Grant	27	F	Lawyer
003	Julie	David	19	F	Student
004	Daniel	Farmer	45	M	Lawyer

For example, let us consider table `PATIENT` (see Table 1). An example of predicate P_1 can be $\langle$ `Name='Mary'` $\rangle$ where `Name` is an attribute and `Mary` the value to be searched for. In this paper, we consider the following queries, listed in Table 2:

Simple keyword queries: They contain only one predicate $Q = P_1 = \langle$`attr=val`$\rangle$. The query is expressed as `SELECT attributes FROM table WHERE attr=val`, `attr` is an attribute and `val` any possible value.

Conjunctive queries (`AND`): Predicates are combined with an `AND` operator: $Q = P_1 \wedge P_2 \wedge ... \wedge P_m$.

Disjunctive queries (`OR`): Predicates are combined with an `OR` operator: $Q = P_1 \vee P_2 \vee ... \vee P_m$.

Range queries: They involve one or two predicates combined with an `AND` operator, and apply to numeric data to find value within a range.

Complex queries: They combine conjunctions, disjunctions and range queries.

3 OUR SSE CONSTRUCTION FOR SQL DATABASES

To clarify the need for an SSE scheme dedicated to SQL databases, we first present an e-health use case that involves medical data. From this use case, we identify the gaps between the theoretical aspect of an SSE scheme and the practical requirement for an encrypted database.

3.1 Use Case

Let us consider a hospital that collects data from their patients. This data may consist of their personal information (names, age, etc.), their biological data (height, weight, body temperature, etc.) or the care they received (surgery operation, etc.). This data is managed in an SQL database (such as in Table 1). The hospital's information technology (IT) department would like to pay the service of a cloud provider to store and enable access to the data.

However, since the e-health data is confidential and because the hospital must comply with strong regulations with respect to the storage and processing of medical data (such as the recent European General Data Protection Regulation (GPDR, 2016)), the IT staff encrypts the data before its outsourcing. Nevertheless, the hospital does not want to lose the functionalities offered by the SQL language. Therefore, the obfuscation of the e-health database should preserve (i) the format and the arrangement of the data, and (ii) the SQL functionalities such as creating and storing the database, or searching and retrieving particular records based on one or more attributes. It is clear that this use case falls into the model of searchable encryption as defined in Section 2.1.

3.2 Background

We base our work on the scheme described in (Curtmola et al., 2006). Unlike our use case, the authors do not consider SQL databases but focus on an unstructured collection of files $\mathcal{D}$. To enable user $\mathcal{U}$ to efficiently search for a keyword ω in the encrypted dataset $\mathcal{C}$, this scheme builds an inverted search index I, that links each keyword ω with $\mathcal{D}(\omega)$, the set of identifiers of documents containing ω. I is also obfuscated so that it does not disclose any information about the data. Only the holder of the secret key can issue a search query. The search time is linear in the size of $\mathcal{D}(\omega)$, which is believed to be optimal (Curtmola et al., 2006). Besides, the authors proved that their scheme is secure under IND-CKA1 (security against chosen-keyword attack). In a nutshell, this security definition requires that an adversary should learn nothing from the outsourced data and the index beyond what can be inferred from the search results and the access pattern, even if the adversary chooses the keywords. In particular, the search tokens do not leak information about the keywords.

Table 2: Example of queries over table `PATIENT`.

Query	Conditions	Results
Simple	`Surname='Grant'`	⟨002, Mary, Grant, 27, F, Lawyer⟩
Conjunctive	`Name='Mary' AND Surname='Grant'`	⟨002, Mary, Grant, 27, F, Lawyer⟩
Disjunctive	`Gender='F' OR Occupation='Lawyer'`	⟨002, Mary, Grant, 27,F, Lawyer⟩
		⟨003, Julie, David, 19, F, Student⟩
		⟨004, Daniel, Farmer, 45, M, Lawyer⟩
Range	`Age<45 AND Age>=35`	⟨001, David, Smith, 38, M, Farmer⟩
	`Age<20`	⟨003, Julie, David, 19, F, Student⟩
Complex	`(Gender='F' OR Occupation='Lawyer') AND Age<20`	⟨003, Julie, David, 19, F, Student⟩

In what follows, we outline this SSE scheme:

$\mathsf{KeyGen}(1^\kappa) \rightarrow \mathcal{K}$**:** Key $\mathcal{K}$ consists of three pseudo-randomly generated keys K_ψ, K_π and K_φ used for two pseudo-random permutations[6] (ψ and π) and one pseudo-random function[6] φ respectively.

$\mathsf{BuildIndex}(\mathcal{D}, \mathcal{K}) \rightarrow I$**:** As shown in Algorithm 1, it constructs three data structures:

- ▷ Linked Lists $\mathcal{L}_i$: For each keyword $\omega_i \in \mathbb{W}$, $1 \leq i \leq N$, the list $\mathcal{L}_i$ is an encrypted form of $\mathcal{D}(\omega_i)$. Each node $N_{i,j}$ of $\mathcal{L}_i$ ($1 \leq j \leq |\mathcal{D}(\omega_i)|$) contains a record identifier in $\mathcal{D}(\omega_i)$, the key $k_{i,j}$ used to encrypt node $N_{i,j+1}$ and the address in array $\mathcal{A}$ of $N_{i,j+1}$. Each node is encrypted using any semantically secure symmetric encryption[7] algorithm denoted $\mathcal{E}$. Node $N_{i,1}$ is encrypted using key $k_{i,0}$.
- ▷ Array $\mathcal{A}$: It stores nodes $\{N_{i,j}\}$ in a pseudo-random order. $\mathcal{U}$ uses key K_ψ as input of ψ to compute the position of each $N_{i,j}$ in $\mathcal{A}$.
- ▷ Look-up table $\mathcal{T}$: Each entry in $\mathcal{T}$ is associated to a keyword $\omega_i \in \mathbb{W}$ and stores information about $N_{i,1}$, namely its address in $\mathcal{A}$ and the key $k_{i,0}$. This information is encrypted using a XOR operation with a key $K_i = \varphi(K_\varphi, \omega_i)$. $\mathcal{T}$ stores the encrypted information in a pseudo-random order, using π seeded with key K_π. In (Curtmola et al., 2006), the authors suggest, without giving further details, to implement $\mathcal{T}$ via a FKS dictionary (Fredman et al., 1984).

At the end of $\mathsf{BuildIndex}$, index $I = (\mathcal{A}, \mathcal{T})$ is stored at server $\mathcal{S}$.

$\mathsf{Trapdoor}(\omega, \mathcal{K}) \rightarrow \tau_\omega$**:** τ_ω consists of the position pos of the entry in table $\mathcal{T}$ associated with ω and the key K_ω to decrypt $\mathcal{T}[\mathsf{pos}]$. We recall that, if ω exists in $\mathcal{D}$ (without loss of generality, we assume that this keyword is ω_i, for $1 \leq i \leq N$), then $\mathcal{T}[\mathsf{pos}_i]$ stores the address in array $\mathcal{A}$ of $N_{i,1}$ and the key $k_{i,0}$. Therefore, τ_ω consists of two values: $\mathsf{pos} = \pi(K_\pi, \omega)$ and $K_\omega = \varphi(K_\varphi, \omega)$.

$\mathsf{Search}(I, \tau_\omega) \rightarrow \mathcal{D}(\omega)$**:** As depicted in Algorithm 2, this algorithm parses τ_ω as pos and K_ω. Then it locates entry $\mathcal{T}[\mathsf{pos}]$ and XOR-decrypts it using K_ω. It obtains the address in array $\mathcal{A}$ of $N_{i,1}$ and the key $k_{i,0}$. Starting from $N_{i,1}$, algorithm Search can decrypt all the nodes $N_{i,j}$, $1 \leq j \leq |\mathcal{L}_i|$ of $\mathcal{L}_i$. Finally, the algorithm outputs the list $\mathcal{D}(\omega_i)$. If one of these steps fails, then algorithm Search aborts and indicates that the keyword was not found.

Algorithm 1: $I \leftarrow \mathsf{BuildIndex}(\mathcal{D}, \mathcal{K})$.

```
Inputs : Database $\mathcal{D}$, Secret Key $\mathcal{K}$
Output: Index $I$
Create dictionary $\mathbb{W}$
for $\omega_i \in \mathbb{W}$ do determine $\mathcal{D}(\omega_i)$

Set global counter $\mathsf{ctr} = 1$
// Array $\mathcal{A}$ creation
Initialize an empty array $\mathcal{A}$ of size $m$
for $\omega_i \in \mathbb{W}$ do
    // Build a linked list $\mathcal{L}_i$ with nodes $N_{i,j}$
    Generate random key $k_{i,0}$
    for $1 \leq j \leq |D(\omega_i)|$ do
        Generate random key $k_{i,j}$
        $N_{i,j} = \langle \mathsf{recordID}_j || k_{i,j} || \psi_{K_\psi}(\mathsf{ctr}+1) \rangle$
        Encrypt $N'_{i,j} = \mathcal{E}_{k_{i,j-1}}(N_{i,j})$
        Store in $\mathcal{A}[\psi_{K_\psi}(\mathsf{ctr})]$
        Set counter $\mathsf{ctr} = \mathsf{ctr} + 1$
    end
    For the last node of $\mathcal{L}_i$, set address of next
      node to NULL
end
Let $m' = \sum_{\omega_i \in \mathbb{W}} |D(\omega_i)|$
if $m' < m$ then fill the $(m - m')$ remaining entries
  of $\mathcal{A}$ with random values

// Look-up table $\mathcal{T}$ creation
for $\omega_i \in \mathbb{W}$ do
    $\mathcal{T}[\pi_{K_\pi}(\omega_i)] = \langle \mathsf{addr}[\mathcal{A}(N_{i,1})] || k_{i,0} \rangle \oplus \varphi_{K_\varphi}(\omega_i)$

Output $I = (\mathcal{A}, \mathcal{T})$
```

[6] A pseudo-random function is a polynomial-time function such that any probabilistic polynomial-time adversary cannot distinguish it from a truly random function.

[7] Informally, a semantically symmetric encryption scheme $(\mathcal{E}, \mathcal{D})$ is a non-deterministic encryption algorithm which yields different ciphertexts for the same plaintext message. $(\mathcal{E}, \mathcal{D})$ denotes the pair of encryption and decryption functions.

Algorithm 2: $\mathcal{D}(\omega) \leftarrow \mathsf{Search}(\mathcal{I}, \tau_\omega)$.

```
  Inputs : Index $\mathcal{I}$, Trapdoor $\tau_\omega$
  Output: Search Results $\mathcal{D}(\omega)$
1 Parse $\tau_\omega = (\mathsf{pos}, K_\omega)$
2 Retrieve $\theta = \mathcal{T}[\mathsf{pos}]$
3 Parse $\langle \alpha || k \rangle = \theta \oplus K_\omega$
4 Decrypt linked list $\mathcal{L}$ whose first node is in $\mathcal{A}[\alpha]$
    and encrypted under key $k$
5 Output each decrypted record ID
```

3.3 Requirements for SQL-compatibility

Given the scenario described in Section 3.1, we identify five requirements for designing an SSE solution that would support SQL databases:

R1: Dictionary Creation. The first requirement concerns the choice of the keywords $\mathbb{W}$. In (Curtmola et al., 2006), the authors only mention to "scan $\mathcal{D}$ to build the set of distinct words". In SQL databases, the data is structured into tables, where each record contains several attributes that take specific values. Hence, one can imagine that the dictionary would contain all the possible values in $\mathcal{D}$. We show in Section 3.5 that such a dictionary does not work in the scope of SQL databases. Therefore, we specify an algorithm CreateDictionary to create an appropriate $\mathbb{W}$.

R2: Practical Look-up Table $\mathcal{T}$. The authors in (Curtmola et al., 2006) mention the use of an FKS dictionary (Fredman et al., 1984), without giving much details about its implementation. FKS dictionaries are known to be memory expensive and finding the *good* hash that prevents hash collisions (as a matter of fact, a perfect hash function) when inserting a new item is a demanding process. For these reasons, we opt for another data structure, simpler to implement, more practical and more flexible, namely a cuckoo hash table. We give an overview of this data structure in Section 3.4.

R3: Structured-data Encryption. In the case of unstructured datasets, each document is encrypted independently. In the case of SQL databases, the organization of data into SQL tables requires that each value in the tables is encrypted independently, using a semantically secure encryption algorithm. This will preserve the functionality of selecting specific attributes from an SQL table using the instruction `SELECT attributes`. Therefore, we will define two additional algorithms EncryptTable and DecryptResults.

R4: Transparent SQL Queries. In (Curtmola et al., 2006), communication details between user and server are omitted. With SQL databases, querying data is a well-defined process that involves specific SQL instructions such as `CREATE`, `INSERT` or `SELECT`. In this paper, we require that a user *transparently* queries the database, meaning that no change is needed in the user's application program. Even in the case of encrypted data, our SSE scheme should respect the SQL language expressiveness, available in the case of plain databases.

R5: Complex Queries. Last but not least, the SSE in (Curtmola et al., 2006) does not tackle the problem of complex queries that contain range, conjunctive or disjunctive queries. We extend this solution with this functionality.

Before delving into the details of our proposal for a symmetric searchable encryption scheme, we outline in the next section the definition of cuckoo hashing.

3.4 Cuckoo Hash

In a cuckoo hash table $\mathcal{T}$ (Pagh and Rodler, 2004), an item v can be stored in one of two possible entries, each located in two different tables, associated with two independent hash functions and a key x. The look-up operation, later denoted $\mathsf{CuckooLookup}(\mathcal{T}, x)$, only requires to examine these two locations. The insertion of a new element v with key x, operation denoted $\mathsf{CuckooInsert}(\mathcal{T}, x, v)$, evaluates the first hash function over x to give the first possible slot in the first table. If this slot is already assigned, then the item currently occupying this location is "kicked out" and v can be inserted in the emptied slot. The removed item is then moved to its alternative location assigned with the second hash function. This move may encounter another collision, thus requiring another element to be kicked out from its current location. This procedure is repeated until the last kicked-out item finds a free slot or when an endless loop is detected (or when the number of kick-outs reaches a predefined maximum). In the latter case, the authors in (Pagh and Rodler, 2004) suggest to *rehash* the data structure, that is, to use two new hash functions.

3.5 Our Construction for Simple Keyword Queries

Our solution is built upon the technique described in (Curtmola et al., 2006), but implements several modifications in order to satisfy the previously mentioned requirements. Let us consider a database $\mathcal{D} = \{T\}$. Table T contains t attributes and R records. Without loss of generality, for databases with multiple tables, we assume that we can reason on a view V extracted from a subset of these tables.

Our first proposal provides a solution to satisfy requirement R1. We define an algorithm CreateDictionary to construct a dictionary $\mathbb{W}$ from $\mathcal{D}$. As required by algorithm BuildIndex, the search index, built upon $\mathbb{W}$, stores information about the records that contain each of the keywords. Besides, a basic SQL query contains a clause `WHERE` of the form $\langle$`attribute=value`$\rangle$, meaning that we are interested in the records where the `attribute` has the specified `value`. Therefore, the search index must indicate that information. Hence, the dictionary $\mathbb{W}$ contains keywords under the form `attribute=value`. We denote n the size of $\mathbb{W}$ ($n \leq t \times R$). Let us take the example of the table `PATIENT` presented in Table 1 and consider the query `SELECT * FROM PATIENT WHERE Name='David'`. If $\mathbb{W}$ only listed the distinct words in the table, as it is the case in (Curtmola et al., 2006), then the index would have lost the information that `David` is a possible value for the attribute `Name`. Note that `David` is also a value for the attribute `Surname`. In this case, the search over the index would have returned all the records that contain the word `David`, regardless of the attributes. To cope with this issue, we suggest to store the keywords ω of the form `Name='David'` in order to preserve the information on the attribute and to prevent the search from returning wrong results. Algorithm 3 depicts the operations to build dictionary $\mathbb{W}$.

Algorithm 3: $\{\mathbb{W}, \{\mathcal{D}(\omega)\}_{\omega\in\mathbb{W}}\} \leftarrow$ CreateDictionary($\mathcal{D}$).

```
   Inputs : Database $\mathcal{D}$
   Output: Dictionary $\mathbb{W}$, Sets $\{\mathcal{D}(\omega)\}_{\omega\in\mathbb{W}}$
1  $\mathbb{W} = \emptyset$
2  for record $\in \mathcal{D}$ do
3      for attribute do
4          $\omega \leftarrow \langle$attribute=value$\rangle$
5          if $\omega \notin \mathbb{W}$ then $\mathbb{W} = \mathbb{W} \cup \{\omega\}$
6
7      end
8  end
9  for $\omega_i \in \mathbb{W}$ do determine $\mathcal{D}(\omega_i)$
10
```

To build index $I = (\mathcal{A}, \mathcal{T})$, we first permute the rows in the database and then we follow the procedure described in Algorithm 4. In our case, we consider the record identifier, denoted as `RecordID`, as an additional attribute of the considered SQL table. To improve the efficiency of (Curtmola et al., 2006) and meet requirement R2, we construct a cuckoo hash table $\mathcal{T}$ for efficient lookups as depicted in Algorithm 4.

At the end of BuildIndex, I is stored in an SQL table called `INDEX` as a binary object (for example, using the type `bytea` in PostgreSQL).

Algorithm 4: $I \leftarrow$ BuildIndex($\mathbb{W}$, $\mathcal{K}$).

```
   Inputs : Keywords $\mathbb{W}$, Secret Key $\mathcal{K}$
   Output: Index $I$
1  Set global counter ctr = 1
   // Array $\mathcal{A}$ creation: same operations as in Algorithm 1
   // Cuckoo Hash Table $\mathcal{T}$ creation:
2  for $\omega_i \in \mathbb{W}$ do
3      value = $\langle \mathsf{addr}[\mathcal{A}(N_{i,1})] || k_{i,0} \rangle \oplus \varphi_{K_\varphi}(\omega_i)$
4      CuckooInsert($\mathcal{T}, \pi_{K_\pi}(\omega_i)$, value)
5  end
6  Output $I = (\mathcal{A}, \mathcal{T})$
```

The next operation in our SSE construction encrypts the SQL table with a semantically secure encryption scheme $\mathsf{E} = (\mathcal{E}, \mathcal{D})$. As opposed to (Curtmola et al., 2006), we define two encryption and decryption algorithms[8] (Algorithm 5). Algorithm EncryptTable encrypts each entry in the table separately, and because of the semantically secure property of the encryption scheme, each encrypted entry is indistinguishable from the others. Note that the attributes and the table names are also encrypted, but the extra attribute `RecordID` is left unencrypted, which does not impact the security of our scheme. Algorithm DecryptResults decrypts the encrypted records returned by the server after the execution of Search. These two algorithms take as input secret key $\mathcal{K}$, output by a modified algorithm KeyGen, that generates an additional key K_{enc}.

Algorithm 5: $\mathcal{C} \leftarrow$ EncryptTable($\mathcal{D}$, $\mathcal{K}$).

```
   Inputs : Table $T$, Secret Key $K_{\mathsf{enc}}$
   Output: Encrypted table $\mathcal{C}$
1  encrypted_table = $\mathcal{E}(K_{\mathsf{enc}}$, table name)
2  for attribute do
3      encrypted_attribute =
         $\mathcal{E}(K_{\mathsf{enc}}$, attribute)
4  end
5  for record in $T$ do
6      for attribute do
7          encrypted_value = $\mathcal{E}(K_{\mathsf{enc}}$, value)
8      end
9  end
```

To retrieve the records from the encrypted database based on a search criterion denoted ω, the user translates a plain SQL query such as `SELECT * FROM PATIENT WHERE Name='David'` (here ω corresponds to `Name='David'`) into an encrypted SQL query, denoted *SE query*. This SE query requests the

[8]Algorithm DecryptResults is the inverse of EncryptTable showed in Algorithm 5, where encryption function $\mathcal{E}$ is replaced by decryption function $\mathcal{D}$.

server to execute algorithm Search over the search index I using the search token generated with algorithm Trapdoor. Therefore, the search procedure operates in a challenge-response interaction between user $\mathcal{U}$ and server $\mathcal{S}$ with the following steps:

1. $\mathcal{U}$ executes algorithm Trapdoor to generate the search token τ_ω for keyword ω. Then, $\mathcal{U}$ forms the following SE query:

   ```
   SELECT * FROM encrypted_table WHERE
         RecordsID IN Search(I, τω) (1)
   ```

   The most important and challenging part of this SE query is `RecordsID IN` Search$(I,\ \tau_\omega)$. The `WHERE` condition requests the execution of algorithm Search **by the SQL server**. Since this algorithm outputs $\mathcal{D}(\omega)$, the `WHERE` condition collects the records whose `RecordID` is in the output of algorithm Search.
2. On execution of the SE query (1), $\mathcal{S}$ runs algorithm Search with the specified trapdoor τ_ω on the indicated index. Algorithm Search is identical to the one defined in (Curtmola et al., 2006), except the fact that we resort to algorithm CuckooLookup to look-up table $\mathcal{T}$, as depicted in Algorithm 6.

The SE query (1) outputs the encrypted records that match the search criterion ω. The last step of our protocol decrypts the obtained records by calling algorithm DecryptResults.

Algorithm 6: $\mathcal{D}(\omega) \leftarrow$ Search(I, τ_ω)

```
Inputs : Index $I$, Trapdoor $\tau_\omega$
Output: Search Results $\mathcal{D}(\omega)$
1 Parse $\tau_\omega = (\gamma, \eta)$
2 Retrieve $\theta =$ CuckooLookup$(\mathcal{T}, \gamma)$
3 Parse $\langle \alpha || k \rangle = \theta \oplus K_\omega$
4 Decrypt linked list $\mathcal{L}$ whose first node is in $\mathcal{A}[\alpha]$
    and encrypted under key $k$
5 Output each decrypted record ID
```

Proposed Framework. In addition to the algorithms we specified in the abovementioned paragraphs, we suggest that our SSE construction follows the framework depicted in Figure 2. Specifically, this framework allows to run the secure search function, namely algorithm Search, on a widely adopted SQL server, namely PostgreSQL.

The architecture of the proposed framework consists of two zones:

Trusted Zone: It includes user $\mathcal{U}$ with an unmodified SQL client application. We define a proxy between $\mathcal{U}$ and $\mathcal{S}$, which executes all the user-side functions of our protocol. The keys generated with KeyGen are stored in a keystore.

Untrusted Zone: It consists of the untrusted cloud server $\mathcal{S}$ running PostgreSQL that stores the encrypted database and the secure search index. This PostgreSQL server is in charge of executing the function Search defined in Algorithm 6.

As shown in Figure 1, our implemented scheme is divided into three phases:

Upload: The user outsources her database by executing the `CREATE` and `INSERT INTO` queries. The proxy receives them, generates keys, runs CreateDictionary, BuildIndex and EncryptTables and converts the queries into an *SE query*. This SE query includes the encrypted version of `CREATE` and `INSERT INTO` queries for the data and the `CREATE` and `INSERT INTO` queries for storing the index into an additional table denoted `INDEX`.

Query: The user executes a `SELECT-FROM-WHERE` query to retrieve some records based on some criteria (the `WHERE` conditions). The proxy receives this query, extracts the criteria, retrieves the keys and executes Trapdoor for each of these criteria. Then, the proxy forms an SE query that converts the SQL query into its “encrypted” version: The proxy replaces the `FROM` argument with the encrypted table name and replaces the `WHERE` conditions with the clause `RecordID IN` Search`(index, trapdoor)`, where Search is the algorithm defined in Algorithm 6, `trapdoor` is the token generated by Trapdoor and `index` is retrieved with `SELECT index FROM INDEX`.

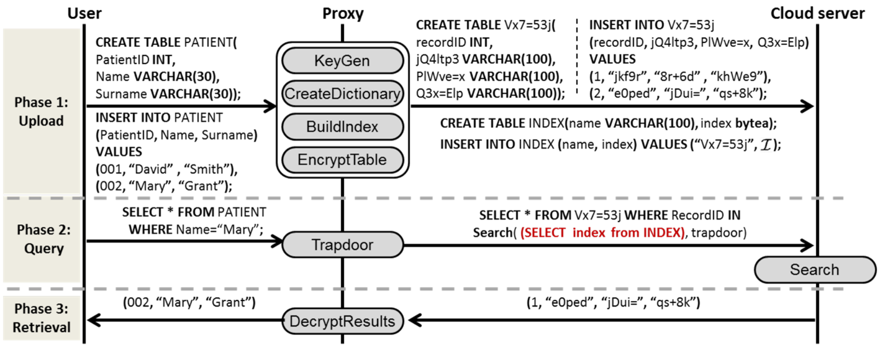
Retrieval: Upon reception of this SE query, the server executes Search over the index and the trapdoor specified in the query. Thanks to the `FROM` and `WHERE` parameters, the server retrieves the encrypted records whose identifiers are output by Search. The proxy then calls DecryptResults and sends to the user the decrypted search results.

3.6 Boolean Queries

We propose to transform boolean queries into several subqueries. The server operates the intersection (in case of conjunction of keywords) or the union (in case of disjunction of keywords) of the search results and returns them to the user. Formally, our solution transforms a plain query $Q : r \to \{0,1\}$ into an SE query $Q' : r \to \{0,1\}$, where Q' consists of the predicates P'_i which are the “obfuscated versions” of the predicates P_i defining Q. The P'_i are of the form `RecordID IN` Search(index, trapdoor$_i$) and are combined with the same boolean operators as the P_i. Here trapdoor$_i$ denotes the token generated by algorithm Trapdoor for the search criteria P_i. Let us consider the query

```
SELECT * FROM PATIENT WHERE Name='Mary'
         AND Surname='Grant'.
```

It contains two predicates $P_1 = \langle$`Name='Mary'`$\rangle$ and



```
CREATE TABLE PATIENT(
PatientID INT,
Name VARCHAR(30),
Surname VARCHAR(30));
INSERT INTO PATIENT
(PatientID, Name, Surname)
VALUES
(001, “David” , “Smith”),
(002, “Mary”, “Grant”);
```

```
CREATE TABLE Vx7=53j(
recordID INT,
jQ4ltp3 VARCHAR(100),
PlWve=x VARCHAR(100),
Q3x=Elp VARCHAR(100));
```

```
INSERT INTO Vx7=53j
(recordID, jQ4ltp3, PlWve=x, Q3x=Elp)
VALUES
(1, “jkf9r”, “8r+6d” , “khWe9”),
(2, “e0ped”, “jDui=”, “qs+8k”);
```

```
CREATE TABLE INDEX(name VARCHAR(100), index bytea);
INSERT INTO INDEX (name, index) VALUES (“Vx7=53j”, 𝓘);
```

```
SELECT * FROM PATIENT
WHERE Name=“Mary”;
```

```
SELECT * FROM Vx7=53j WHERE RecordID IN
Search( (SELECT index from INDEX), trapdoor)
```

```
(1, “e0ped”, “jDui=”, “qs+8k”)
```

```
(002, “Mary”, “Grant”)
```

Figure 1: Workflow of our scheme.

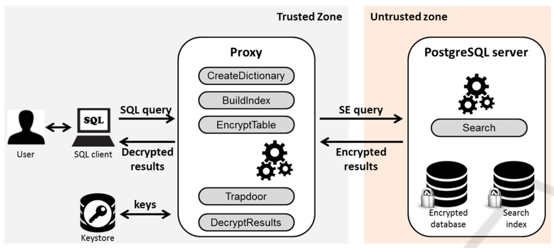


Figure 2: Architecture of our SSE solution.

$P_2 = \langle$`Surname='Grant'`$\rangle$, linked with the boolean operator AND. To transform this SQL query into an SQL-SE query, the proxy generates the two following trapdoors: $\mathsf{trapdoor}_1 = \mathsf{Trapdoor}($`Name='Mary'`$)$ and $\mathsf{trapdoor}_2 = \mathsf{Trapdoor}($`Surname='Grant'`$)$ and executes the SE query:

```
SELECT * FROM encrypted_table WHERE
RecordsID IN Search(index, trapdoor₁) AND
RecordsID IN Search(index, trapdoor₂).
```

3.7 Range Queries

Let us consider the attribute `Age` from the database example depicted in Table 1 and a range query that requests to retrieve the records where `Age` > 40 and `Age` ≤ 45. Using the SSE solution depicted in Section 3.6, we can simply generate trapdoors for each of the values between 41 and 45, so five trapdoors for the keywords $\mathsf{Age} = 41$, $\mathsf{Age} = 42$, etc. For each trapdoor, the server calls algorithm Search and computes and sends back to the user the union of the search results. In case of wider ranges (such as `Age` > 18 and `Age` ≤ 75), this solution would not be efficient because of the execution of several instances of Trapdoor and Search.

The idea of our solution for range queries is to reduce a range query problem into a keyword search problem. To a particular value of an attribute (for example: value 45 for attribute `Age`), we associate a value (for example: $[45-50[$) to a new attribute `Range_Age`. Therefore, we define a new keyword $\langle$`Range_Age` $= [45-50[\rangle$ that will be inserted in index $\mathcal{I}$ during the execution of BuildIndex. As a matter of fact, we extend the initial database $\mathcal{D} = T$ with a new attribute[9] `Range_attribute`. For each record r in $\mathcal{D}$, we let the user specify the range value of the corresponding attribute. Table 3 shows table `PATIENT` extended with the attribute `Range_Age`. Note that the user selects its preferences on the range interval and the initial value, based on the content of the database and the possible queries she may issue, depending on the use cases[10].

Table 3: Table `PATIENT` with Range column.

ID	Name	Surname	Age	Range_Age	Gender	Occupation
001	David	Smith	38	[35;40[	M	Farmer
002	Mary	Grant	27	[25;30[	F	Lawyer
003	Julie	David	19	[15;20[	F	Student
004	Daniel	Farmer	45	[45;50[	M	Lawyer

When user $\mathcal{U}$ wants to retrieve the records that match a range query, the proxy issues an SE query that contains the trapdoor(s) associated with the specified range. Different cases can be encountered:

1. The range query exactly corresponds to one of the range included in the search index. If we consider the example of Table 3, this case applies to search queries such as `SELECT * FROM PATIENT WHERE Age` ≥ 35 `AND Age` < 40. The range query $\langle$`Age` ≥ 35 `AND Age` $< 40\rangle$ exactly matches the range we specified for the record dealing with patient `David Smith`. In this case, the proxy calls algorithm Trapdoor for the keyword $\langle$`Range_Age` $= [35;40[\rangle$. This case is reduced to a simple keyword search query.
2. The range query is different from the ranges

[9]Practically, this column will not be stored at the server.

[10]The specified ranges can have different interval length depending on the frequency of items in the interval. We can use statistical tools for defining the range intervals but this topic is out of scope of this paper.

included in the search index. Let us take the following range query: `SELECT * FROM PATIENT WHERE Age` ≥ 18 `AND Age` < 25. The interval $[18;25[$ is not one of the ranges we included in Table 3. Therefore, the proxy *factorizes* the search range $[18;25[$ into a union of ranges (inserted in the search index during the execution of BuildIndex) and singletons of discrete values. In our example $[18;25[= \{18\} \cup \{19\} \cup [20;25[$. Hence, the proxy calls algorithm Trapdoor for the two singletons $\langle \texttt{Age} = 18 \rangle$ and $\langle \texttt{Age} = 19 \rangle$ as well as for $\langle \texttt{Range\_Age} = [20;25[\rangle$. Therefore, instead of 7 search tokens, our solution only generates 3 trapdoors and the corresponding SE query is a disjunction of clauses `RecordID IN` Search(index, trapdoor).

4 IMPLEMENTATION AND PERFORMANCE EVALUATION

Implementation. Our SSE scheme is implemented according to the architecture shown in Figure 2. All the functions at the proxy are implemented in Java. The server runs PostgreSQL server (version 9.5). Search is coded in Java with PL/Java[11], an add-on for server-side procedures. The code is packed into a JAR that is further loaded into the PostgreSQL backend, so as to execute Search directly in PostgreSQL. We tested our prototype on a single machine that simulates both the proxy and the server. The machine has four 3.20GHz Intel Core$^{\text{TM}}$i5-3470 processors, 32GB of RAM and runs Ubuntu 14.04 LTS.

E-Health Database. This work is done in the context of a collaborative project named CLARUS[12], which collected data from anonymized medical records of a hospital. This database consists of eight tables and queries are performed over a view (in the SQL terminology) called `LAB`. This view lists 64.440 records and comprises eight attributes, including a patient identifier (`pat_id`), a patient name (`pat_name`), a patient last name (composed of two parts `pat_last1` and `pat_last2`) and an identifier of an episode of care (`ep_id`).

Results. We measured the time consumed by the phases described above over the e-health database. We let our program automatically add range attributes for `pat_id`. The length of the ranges is 10: we insert `Range_pat_id` values of the form $[0,9], [10,19]$, etc. We ran 100 executions of the functions, and computed the average time, listed in Tables 4 and 6. We compare these measurements with the case where upload (resp. search) is performed without the application of our SSE mechanism in Table 5 (resp. Table 7).

[11] https://tada.github.io/pljava/. [Accessed Oct. 09, 2017]

[12] http://www.clarussecure.eu

Table 4: Upload with SSE.

Operation	Time (s)
Index Creation	22.1 s
Encryption	5,60 s
Server storage	4.60 s

Table 5: Upload without SSE.

Operation	Time (s)
Upload	9.735 s
Server storage	1.238 s

The time needed by the server for storage is greater than the case where no encryption is performed, since the server is required to store the index together with the encrypted data. We evaluated single keyword queries (keywords with different numbers of occurrences) and range queries. The Retrieval phase is fast: less than 1 second to decrypt the search results. We recall that Search first finds the entry in table $\mathcal{T}$ associated with the searched keyword and then decrypts the corresponding linked list. In average, Search has acceptable costs, given that the data is encrypted. For the most frequent keyword, Search decrypts 4275 nodes which yields the time reported in Table 6. In the case of range queries, we tested the query `pat_id` ≥ 50 `AND pat_id` < 150, which generates 10 trapdoors of the form $\langle \texttt{Range\_pat\_id} = [50,59] \rangle$, ..., $\langle \texttt{Range\_pat\_id} = [140,149] \rangle$ (since we configured the attribute `Range_pat_id` with intervals of size 10). Thus, Search is executed 10 times. The Upload phase is relatively long (22 s) since BuildIndex scans the database value by value to create the index and involve a high amount of permutations. Nevertheless, these operations are performed only once for an unbounded number of queries.

Table 6: Query and Retrieval.

Query	Trapdoor	$\lvert\mathcal{D}(\omega)\rvert$	Search	Decryption
Single Keyword				
`pat_last2='GARCIA'`	2 ms	4275	122 s	0,37 s
`pat_name='RAUL'`	1 ms	178	5.6 s	0,02 s
`pat_name='MORAD'`	1 ms	11	0.8 s	0,002 s
`pat_last1='DUC'`	9 ms	0	0.5 s	–
Range Query	2,0 ms	198	9.87 s	0,04 s

Table 7: SQL search without SSE.

Query	SQL search
Single Keyword	
`pat_last2='GARCIA'`	3.5 ms
`pat_name='RAUL'`	8.0 ms
`pat_name='MORAD'`	7 ms
`pat_last1='DUC'`	7 ms
Range Query	45 ms

5 RELATED WORK

We reviewed the literature in the area of searchable encrypted database systems. The design of such systems always trades off between the level of security, the query functionalities preserved in spite of encryption and the performance.

Boolean Queries. In (Hacigümüş et al., 2002), the authors proposed a heuristic to execute SQL queries over encrypted data where the querier user is required to perform a heavy processing of the search results, whereas in our case, query and retrieval are lightweight operations. Besides, this solution encrypts records as a whole whereas our construction encrypts records attribute-wise. The authors do not provide any security guarantee. In (Cash et al., 2013), an SSE scheme for boolean queries, named OXT, is tested with a MySQL database. This solution builds two indexes, which increases the storage overhead at the server. This scheme supposes the knowledge of the frequency of the keywords to perform efficient search. The user first queries the least frequent keyword of a conjunctive keyword query and then filters the results for the other keywords. In (Pappas et al., 2014), Blind Seer is proposed as a system which enables boolean SQL queries via a tree-based search index. It resorts to Yao's garbled circuit (Yao, 1986), which requires the user and the server to jointly parse the tree to process the search query. Our construction only incurs a single round of interaction between the user and the server.

Range Queries. In (Agrawal et al., 2004), the authors defined order-preserving encryption (OPE) which enables range queries, but reveals the order of numeric data. Our scheme avoids this leakage by using a semantically secure encryption algorithm. ARX (Poddar et al., 2016) is based on a tree to evaluate range queries over encrypted data, and uses Yao's garbled circuit to traverse the tree to respond to range queries. Whenever such a query is computed, nodes of the tree must be updated, which incurs complexity overhead at the user-side. Besides, the authors tested this solution on a NoSQL database, which does not store structured data as in MySQL or PostgreSQL.

CryptDB. Proposed by (Popa et al., 2012), this system allows equality, range and boolean queries over encrypted SQL databases, thanks to onion encryption, that encrypts each attribute with one or more onion layers. Each of the encryption layers preserves a particular functionality. For simple keyword search, CryptDB applies a deterministic encryption, disclosing to the server the occurrences of a particular keyword. CryptDB also preserves the functionality of range queries by adding a layer of OPE, which also leaks some information to adversaries. As opposed to CryptDB, even if the same data is queried several times, our solution preserves data and query privacy.

6 CONCLUSION

This paper proposes an SSE scheme for SQL databases. The proposed solution builds upon the already existing searchable encryption proposed by Curtmola et al. (Curtmola et al., 2006), which is transformed to an SQL-compatible scheme. Our solution supports several query functionalities including range and Boolean queries. Fruthermore, thanks to the use of cuckoo hashing, the search operation becomes more efficient. We finally present a framework for implementation which embeds the search algorithm into PostgreSQL and that converts plain SQL queries into "encrypted" SQL queries directly executable by the Postgres server. This framework is evaluated in terms of performance using an e-health database.

Our future work consists in developing an optimized system for databases with million of records. We plan to conduct an in-depth performance evaluation including comparison with existing -comparable- implementations, if any, to show its practicality in real-world scenarios.

ACKNOWLEDGMENTS

This work was partially funded by European Commission through the H2020 project CLARUS (grant No. 644024). We particularly thank the Fundació Clínic per a la Recerca Biomèdica (FCRB) for providing our test e-health database. The authors also would like to thank Mr. Yiadh Tlijani for his help in simulations.

REFERENCES

Agrawal, R., Kiernan, J., Srikant, R., and Xu, Y. (2004). Order Preserving Encryption for Numeric Data. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 563–574. ACM.

Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.-C., and Steiner, M. (2013). Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *Advances in Cryptology–CRYPTO 2013*, pages 353–373. Springer.

Chamberlin, D. D. and Boyce, R. F. (1974). SEQUEL: A Structured English Query Language. In *Proceedings* *of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '74, pages 249–264. ACM.

Chase, M. and Kamara, S. (2010). Structured Encryption and Controlled Disclosure. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 577–594. Springer.

Curtmola, R., Garay, J., Kamara, S., and Ostrovsky, R. (2006). Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. Cryptology ePrint Archive, 2006:210.

Fredman, M. L., Komlós, J., and Szemerédi, E. (1984). Storing a Sparse Table with 0(1) Worst Case Access Time. *J. ACM*, 31(3):538–544.

GPDR (2016). General Data Protection Regulation. *Official Journal of the European Union*, L119:1–88.

Hacigümüş, H., Iyer, B., Li, C., and Mehrotra, S. (2002). Executing SQL over Encrypted Data in the Database-service-provider Model. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 216–227. ACM.

Kamara, S. and Papamanthou, C. (2013). Parallel and Dynamic Searchable Symmetric Encryption. In *International Conference on Financial Cryptography and Data Security*, pages 258–274. Springer.

Pagh, R. and Rodler, F. F. (2004). Cuckoo Hashing. *Journal of Algorithms*, 51(2):122–144.

Pappas, V., Krell, F., Vo, B., Kolesnikov, V., Malkin, T., Choi, S. G., George, W., Keromytis, A., and Bellovin, S. (2014). Blind Seer: A Scalable Private DBMS. In *2014 IEEE Symposium on Security and Privacy (SP)*, pages 359–374. IEEE.

Poddar, R., Boelter, T., and Popa, R. A. (2016). Arx: A Strongly Encrypted Database System. *IACR Cryptology ePrint Archive*, 2016:591.

Popa, R. A., Redfield, C., Zeldovich, N., and Balakrishnan, H. (2012). CryptDB: Processing Queries on an Encrypted Database. *Communications of the ACM*, 55(9):103–111.

Song, D. X., Wagner, D., and Perrig, A. (2000). Practical Techniques for Searches on Encrypted Data. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S P 2000*, pages 44–55.

Yao, A. C.-C. (1986). How to Generate and Exchange Secrets. In *27th Annual Symposium on Foundations of Computer Science, 1986*, pages 162–167. IEEE.