Proxy-Mediated Searchable Encryption in SQL Databases Using Blind Indexes

Eugene Pilyankevich
eugene@cossacklabs.com

Dmytro Kornieiev
dmitry@cossacklabs.com

Artem Storozhuk
artem@cossacklabs.com

ABSTRACT

Rapid advances in Internet technologies have fostered the emergence of the "software as a service" model for enterprise computing. The "Database as a Service" model provides users with the power to create, store, modify, and retrieve data from any location, as long as they have access to the Internet. As more and more datasets (including those containing private and sensitive data) are outsourced to remote / cloud storage providers, the data owner, firstly, needs to be certain of the security of data against thefts by outsiders and, secondly, the data owner needs to secure the data not only against external threats but also from untrusted service providers. The same is true for distributed applications with complex microservice architectures. However, the use of standard encryption schemes for data protection also effectively eliminates the search capability of the database service which, in turn, severely constrains the ability of the service to manage large volumes of data.

Searchable encryption (SE) is a class of cryptographic techniques that addresses these issues. SE allows a user to write encrypted data to an untrusted storage provider while retaining the ability to perform queries without decrypting the data. This can be achieved by either encrypting the data in a special way that enables queries to be executed directly on the ciphertext or by introducing a searchable encrypted index which is stored together with the encrypted data on the storage provider.

All reasonably efficient SE schemes have a common problem. They leak the search pattern that reveals whether two search queries were performed for the same keyword or not. Hence, the search pattern provides the information on the frequency of occurrence for each query. This information can be further exploited by statistical analysis, allowing an adversary to gain full knowledge about the plaintext keywords, which significantly decreases the security benefits of encrypting the data.

There is no single best publicly known secure search system or a set of such techniques. The design of SE schemes is a balancing act between security, functionality, performance, and usability. This is especially true since different users will want different database architecture (SQL, NoSQL, NewSQL).

Most progress in the area of SE has been made in the setting of keyword search on encrypted documents. While this has many practical applications (i.e. email, desktop search engines, cloud document storage), much of the data produced and consumed is stored and processed in relational databases queried using SQL.

In this paper, we propose Acra Searchable Encryption (Acra SE) – a solution for secure search in an encrypted SQL database based on the blind indexing approach developing and evolving the original idea of the CipherSweet project.

Keywords: searchable encryption, blind indexing, SQL database security, distributed applications.

1 INTRODUCTION

The wide proliferation of sensitive data in open information and communication infrastructures has increased both the volume of research on secure data management and its relevance. Recent developments in cloud computing have made it possible to store data remotely and access it from any part of the network, and by any service or software subcomponent. However, this freedom also has its downsides, since storage providers (administrators or attackers with root access) have full access to their servers and – consequently – to the plaintext data. A trusted provider may sell its business to an untrusted entity which will then have full access to your data. In short, to store sensitive data securely on an untrusted server, the data needs to be encrypted. This reduces the security and privacy risks by hiding all the information about the plaintext data. Encryption makes it impossible for both insiders and outsiders to access the data without the cryptographic keys, but at the same time, this removes the ability of the data owner to search the data. One trivial approach to searching over encrypted data is to download the whole database, decrypt it locally, and then search for the desired results in the plaintext data. For most applications, this approach is impractical due to the significant size of the datasets (for an enterprise that keeps a historical database of customer facing data like sales, shipments, etc., the size of these databases can be quite large – from Tbytes to Pbytes [1]). Another method lets the server decrypt the data, runs a query on the server side and only returns the user the results. In this case, security is compromised as all records protected by encryption are revealed. Instead, it is desirable to support the fullest possible search functionality on the server side, without decrypting the data, with the smallest possible loss of data confidentiality. This is called searchable encryption (SE). Figure 1 illustrates a typical workflow of SE.

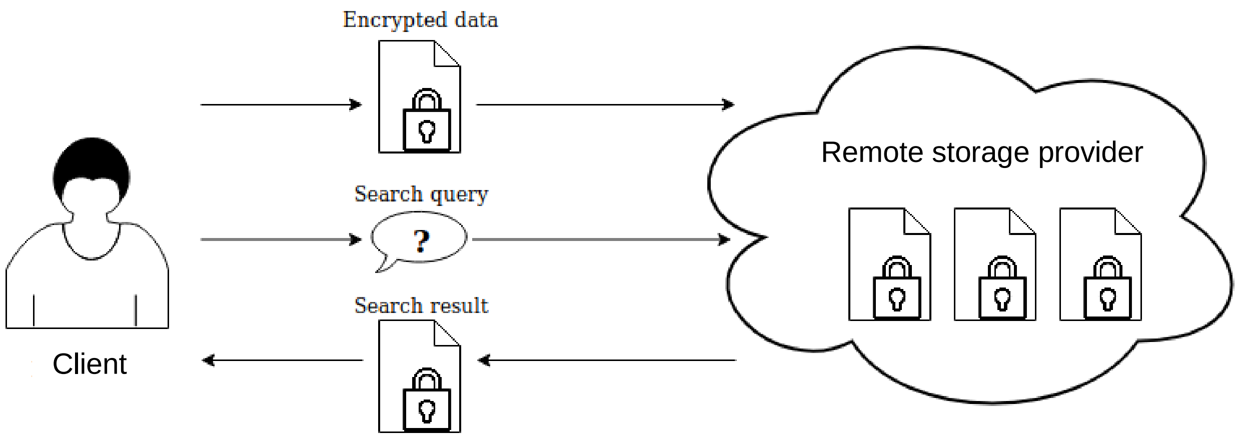

Figure 1: Abstract searchable encryption scheme.

A SE scheme allows remote storage provider to search over encrypted data without learning the information about the plaintext data. Some schemes implement this via ciphertext that allows searching, while most other schemes let the client generate a special searchable encrypted index (subsequently, for this notion, we use the term “*blind index*”) and store it together with encrypted data on remote storage on the provider’s side. To search, the client includes a so-called *trapdoor* into the search query – a predicate function on a searchable encrypted keyword, which can be used by storage provider to determine (as in “check if the predicate is satisfied”) whether the stored encrypted keyword should be included in the search result.

In recent years, the relevance of searchable encryption has significantly increased due to the adoption of GDPR [2], which regulates personal data protection and privacy for all members of the European Union. In particular, GDPR requires pseudonymisation (where applicable) of personal data stored by commercial companies (usually by the means of encryption) [3]. Another example is a requirement to encrypt electronic health records stipulated by legislation around the world. These security demands pose the question of effective and secure search through highly sensitive data.

The first SE scheme (and the first formal scientific definition of searchable encryption scheme) was proposed by Song, Wagner, and Perrig [4] in 2000. The problem of search over encrypted data has since received much interest from governments [5], academia [6 - 9] and industrial sector (Bitglass, CipherCloud, Skyhigh, Crypteron, IQrypt, Kryptnostic, Google's Encrypted BigQuery, Microsoft's SQL Server 2016, Azure SQL Database, PreVeil, SkyHigh, StealthMine).

It's well known [8, 9] that designing a SE scheme is a balance between security, functionality, performance, and usability. When a scheme improves in one aspect, it usually has to make sacrifices in others. Security descriptions focus on the information that is revealed or *leaked* to an attacker with access to the database server. Functionality is primarily characterized by the query types that a protected database can answer. Queries are usually expressed in a standard language, i.e. the structured query language (SQL). Performance and usability are affected by data structures and indexing mechanisms of the database, as well as by the required computational and network costs.

1.1 Searchable encryption security

Any searchable encryption scheme will leak information that can be divided into three groups: blind index metadata, search pattern, and access pattern. We will use definitions from [9], accepted for general-purpose databases (including, as in our case, relational databases).

Blind index metadata (index information in [9]) refers to the information about the keywords contained in the blind index (not to be confused with traditional database indexes for performance improvement). Blind index metadata is leaked from the stored ciphertext / blind index. This information may include the number of keywords per document / database, the number of documents, the documents' length, document IDs, and / or document similarity. Documents can be treated as a set of cells in the table of SQL database.

Search pattern refers to the information that can be derived in the following sense: determining whether two searches use the same keyword / predicate given that two searches return the same results. Accessing the search pattern allows the server to use statistical analysis and (possibly) determine information about the query keywords.

Access pattern refers to the information that is implied by the query results. For example, one query can return a document x, while the other query could return x and another 10 documents. This implies that the predicate used in the first query is more restrictive than that in the second query.

In practice, the following types of objects within SE systems are vulnerable to leakage [8]:

1) Data items and any indexing data structures;
2) Queries;
3) Records returned in response to queries;
4) Access control rules and the results of their application.

Most scientific papers follow the traditional security definition from [10]. Namely, it requires that nothing leaks from the remotely stored ciphertexts and blind indexes beyond the result and pattern of search queries. Meaning, SE schemes should not leak the plaintext keywords in either the trapdoor or the blind index. To capture the concept that neither blind index metadata nor the search pattern is leaked there is a so-called definition of full security of SE scheme [11]. This is an absolutely logical security requirement because the recent works [12 - 17] demonstrated how attackers can recover plaintext data if they observe queries over an encrypted database. And, unfortunately, in real-world scenarios, full security (not to be confused with Shannon's information-theoretic security which cannot be achieved in searchable encryption by definition) for SE scheme is almost unreachable. Most theoretical schemes from [9] leak at least the search pattern and the access pattern. The remaining schemes SSW [11] and BTH+ [18] are fully secure but they are also inefficient in production databases (a single search query may take 47 seconds in an encrypted database with 1000 documents [18]). At the same time, more efficient SE schemes with existing implementations (research prototypes) such as CryptDB [19], Arx [20], Seabed [21], Mylar [22] have rather complicated architecture that reduces real-world usage attractiveness and, moreover, are criticized for using security models that are too abstract and operating under a false assumption that a snapshot attacker is unable to obtain the search pattern [23].

1.2 Current SE solutions for SQL databases

Research on searchable encryption focuses mainly on the scenario of a user who outsources an encrypted collection of documents and would like to further search the keywords over this encrypted dataset. While this theoretical setting is valid, in practice, many organizations store data in relational databases that structure data into tables according to a set of attributes. The popular SQL language (first described back in 1974 by Chamberlin and Boyce) enables users to store, query, and update their data in a user-friendly manner. Consequently, a cryptographic data protection mechanism for searching over encrypted data stored in a SQL database should allow the server to efficiently process the search queries without having an access to the plaintext data. However, creating a method satisfying such constraint for SQL databases is not straightforward. Indeed, existing solutions build an index of keywords. In the case of SQL, data is arranged into tables and records and queries are based on conditions applied over one or more attributes of a record. Therefore, keywords should preserve this notion of an attribute. Moreover, SQL allows comparing the data (range queries), which is not always addressed in the existing work.

As far as we know, the first searchable encryption solution for relational database was proposed by Hacigűműs et al. [6] and was based on quantization. Generally speaking, the attribute space of each column is partitioned into bins and each element in the column is replaced with its bin number. In [19] Popa et al. has proposed CryptDB. It was the first non-quantization-based solution also capable of handling a large subset of SQL. Instead of using quantization, CryptDB relies on property-preserving encryption like deterministic and order-preserving encryption, applied to a column of an SQL table. The CryptDB design influenced the Cipherbase system from Arasu et al. [24] and the SEEED system from Grofig et al [25].

Some SE systems have made the transition to full database solutions. Those systems report performance analysis, perform rule enforcement, and support dynamic data.

It is in this context that we propose Acra SE which adds SE capabilities to the existing Acra database security suite [26] based on the “blind indexing” approach developed by the CipherSweet project [27].

Table 1 summarizes the qualities of available SE schemes. All these schemes have implementations and support relational databases.

Table 1

System	Query expressiveness	Additional features	Open-source
CryptDB	Equality, Boolean, Range, Sum, Join, Update	User authentication, Access control	Yes
Blind Seer	Equality, Boolean, Keyword, Range, Update	Query policy	No
OSPIR-OXT	Equality, Boolean, Keyword, Range, Substring, Wildcard, Update	Query policy	No
SisoSPIR	Equality, Keyword, Range, Substring	Query policy	No
CipherSweet	Equality	Key management	Yes
Acra	Equality	Authentication, Query policy, Intrusion detection, Key management, Monitoring and observability	Acra: Yes Acra SE: No

CryptDB replicates most of the functionality of database management systems with a performance overhead of under 30% [19]. Blind Seer [28] reports slowdown between 20% and 300% for most queries, while OSPIR-OXT [29] report that they occasionally outperform a baseline MySQL 5.5 system with a cold cache and are one order of magnitude slower than MySQL with a warm cache. The SisoSPIR system [30] reports performance slowdown of 500% compared to a baseline MySQL system on keyword equality and range queries. CipherSweet doesn't provide measurable performance characteristics.

At the same time, data leakage types may vary. CryptDB is the fastest and the easiest to deploy. However, once a column is used in a query, CryptDB reveals the statistics on the entire dataset's value on this column. Blind Seer and OSPIR-OXT also leak information to the server but primarily on the data returned by the query. Thus, they are appropriate in setups where a small fraction of the database is queried. Finally, SisoSPIR is appropriate if a large fraction of the data is queried regularly. However, SisoSPIR does not support Boolean queries, which is limiting. CipherSweet only supports equality queries but offers simple and straightforward security model, which is essentially a time-memory trade off that introduces the risk of partially-known plaintext attacks. These attacks are more similar to a crossword puzzle than traditional cryptanalysis techniques [31].

1.3 Our contribution

Our contribution is a new SE system based on the blind indexing approach – storing keyed hashes of keywords on the database side.

Unlike the existing solutions, our system provides strict separation of duties (based on reverse proxying – see Figure 2) which guarantees absence of cryptographic key leakage from application, secure storage and management of cryptographic keys, and a set of additional security features that better correspond to the real-world threats.

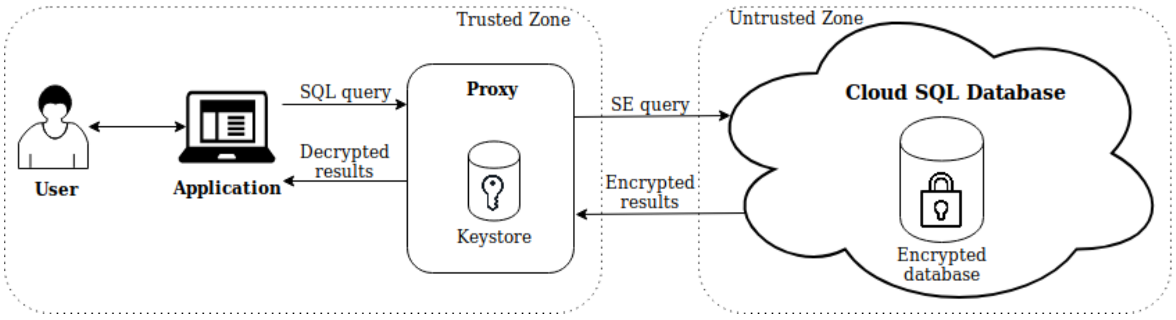

Figure 2: Reverse proxy between application and untrusted storage provider.

1.4 Organization

This paper has the following structure: Section 2 introduces common definitions of cryptographic schemes used as building blocks for our design, details on data flow and cryptographic keys management; Section 3 describes the practical security evaluation of our system; Section 4 provides information about implementation and performance evaluation; Section 5 sets out our conclusions.

2 PRELIMINARIES

In the previous section, we discussed the notion of searchable encryption that allows a client to store encrypted data with an untrusted storage provider, while still being able to query the data securely. Significant progress has been made in this field after more than a decade of research. The main academic efforts were dedicated to improving query expressiveness, efficiency, and security. One can recognize the tradeoffs among these three directions: (1) security versus efficiency, (2) security versus query expressiveness, and (3) efficiency versus query expressiveness. When a scheme tries to be better in one aspect, it usually has to make sacrifices in others [9].

In practice, many organizations such as governments, hospitals, or private companies store data in relational databases [32], where users are able to store, query, and update their data using SQL. As noted, direct application of the existing solutions for search over an encrypted collection of files in SQL databases is not straightforward (due to infrastructural requirements, trust model and / or application demands). Besides, only a few of them have industry-proven implementations with available source code [19, 27, 33].

Based on our analysis of available implementations, one of the most noteworthy secure search frameworks is CipherSweet. It is based on a blind indexing strategy, which relies on storing keyed hashes of search keywords on the database side. In the following sections, we provide a cryptographic design of our approach (proxy-mediated encrypted search with blind indexing) based on the ideas from CipherSweet and implemented in Acra database encryption suite [33].

2.1 Low-level cryptographic schemes

2.1.1 Cryptographic key generation (CKG)

Cryptographic key generation scheme is a pair of algorithms $CKG = (GenKey, GenKeyPair)$, such that:

1) The algorithm $GenKey$ takes no input and returns a symmetric key K: $K = GenKey()$;
2) The algorithm $GenKeyPair$ takes no input and returns an Elliptic Curve Diffie-Hellman (ECDH) [34] key pair (pk, sk) where pk is a public key and sk is a private key: $(pk, sk) = GenKeyPair()$.

The security of $GenKey$ algorithm is defined similarly to the security definition of PRNG with Input object [35], while the security of $GenKeyPair$ can be defined by "cryptographic quality" of the underlying elliptic curve [36]. For simplicity's sake, we assume that the input parameters of $GenKey$ and $GenKeyPair$ are secure and skip them (i.e. key length for $GenKey$, domain parameters of the elliptic curve for $GenKeyPair$, etc.).

2.1.2 Authenticated encryption with associated data (AEAD)

The cryptographic scheme for authenticated encryption with associated data is a pair of algorithms $AEAD = (E, D)$, such that:

1) The algorithm E (encryption algorithm) takes a plaintext R, a key K, and returns the ciphertext: $R^K = E(R, K)$ and the authentication tag (we consider the authentication tag to be a part of the ciphertext).
2) The algorithm D (decryption algorithm) takes a ciphertext R^K, a key K, and returns the decrypted text: $R = D(R^K, K)$ and the indication of the integrity of the plaintext.

The security is defined similarly to the definition of Authenticated Encryption scheme from [37]. The preferable candidates for the *AEAD* scheme would be popular block cipher AES-256 [38] in GCM mode [39] or ChaCha20 [40] with Poly1305 authenticator [41].

2.1.3 Key Encapsulation (KE)

The cryptographic scheme for key encapsulation is a combination of three algorithms: $KE = (GenKeyPair, W, U)$. Let's formally define these algorithms:

1) The algorithm $GenKeyPair$ has been defined earlier (see section 2.1.1);
2) The algorithm W (encapsulation algorithm) takes a public key pk_1 from ECDH key pair (pk_1, sk_1), a private key sk_2 from ECDH key pair (pk_2, sk_2), a symmetric key K, and returns the encapsulated symmetric key: $K^{pk_1} = W(K, pk_1, sk_2)$.

3) The algorithm U (decapsulation algorithm) takes a private key sk_1 from ECDH key pair (pk_1, sk_1), a public key pk_2 from ECDH key pair (pk_2, sk_2), an encapsulated key K^{pk_1}, and returns a decapsulated symmetric key: $K = U(K^{pk_1}, pk_2, sk_1)$.

The definition of key encapsulation scheme security is similar to the common security definition of the Elliptic Curve Integrated Encryption Scheme (ECIES) [42], which is also the most preferable candidate.

2.1.4 Blind index calculation (BIC)

The cryptographic scheme of blind index calculation is a single algorithm T that takes some data R, a symmetric key K, and returns a string $UT_R = T(R, K)$. The security of the blind index calculation is defined similarly to the definition of pseudo-random function (PRF) [43]. The preferable candidates for the *BIC* scheme include HMAC-family algorithms [43].

2.1.5 Secure communication (SC)

The cryptographic scheme of secure communication is used for establishing a secure channel between two entities. Let's denote those entities as A (which possesses a private key sk_A and a public key pk_A from A's long-term ECDH key pair) and *B* (possesses a private key sk_B and a public key pk_B from *B*'s long-term ECDH key pair).

SC includes three phases:

1) Mutual authentication;
2) Key establishment ($KEst$);
3) Secure data transfer.

During the *first* phase, A authenticates B and B authenticates A. This phase works under a standard assumption that long-term public keys pk_A and pk_B have been previously exchanged securely (without tampering), i.e. with a help of public-key infrastructure [44].

During the *second* phase, A and B establish a common session encryption key using ECDH-based protocol $KEst$. Actually, A uses $KEst$ that takes private key sk_A, public key pk_B and returns a shared secret $SS = KEst(sk_A, pk_B)$, while B uses $KEst$ that takes private key sk_B, public key pk_A and returns the same shared secret $SS = KEst(sk_B, pk_A)$.

The first two phases can be represented by the well-known authenticated key agreement protocols [45]. The security of such protocols is defined in [45, 46]. The preferable candidates include Blake-Wilson protocols (2 or 3) [45] or protocol from [46].

During the *third* phase, both A and B use an authenticated encryption scheme (defined above) to protect the data transferred over the communication channel.

2.2 AcraStruct object

AcraStruct is a multipurpose cryptographic container that stores encrypted data (and the associated metadata) using a specific format. AcraStruct can be stored as a binary record in the table of a database or as a blob on a filesystem. AcraStruct is specially designed with respect to high-level practical purpose: encrypt sensitive data by having only public ECDH key of an application.

According to Acra's documentation [48], AcraStruct can be informally defined as a set of five elements (concatenated together, one after another): `Begin_Tag[8]`, `Throwaway_Public_Key[45]`, `Encrypted_Random_Key[84]`, `Data_Length[8]` and, finally, `Encrypted_Data[Data_Length]`. Note that each element is a byte array with its length provided inside the square brackets. Next, let's provide a more formal definition of AcraStruct.

Definition 1. AcraStruct *AS* is a sequence (byte array) of variable length:

$$AS = Tag \,||\, pk_{rand} \,||\, K_{rand}^{pk_A} \,||\, len \,||\, Ciphertext,$$

where *Tag* is a special tag that marks the beginning of an AcraStruct; pk_{rand} is a public key from a randomly generated ECDH key pair (pk_{rand}, sk_{rand}); $K_{rand}^{pk_A}$ is a random symmetric key encapsulated by *KE* scheme: $K_{rand}^{pk_A} = W(pk_A, sk_{rand})$, where pk_A is a public key from ECDH key pair (pk_A,sk_A) that belongs to the client A; len is a little-endian representation (byte array with a length of 8) of an integer variable that defines the length of the encrypted data; $Ciphertext = E_{K_{rand}}(R)$ — is data *R*, encrypted with *AEAD* scheme (see above).

Along with defining AcraStruct, let's introduce two algorithms that abstract the low-level cryptographic logic:

1) The algorithm CreateAS (AcraStruct creation) takes a plaintext R, a public key pk_1 from ECDH key pair $(\mathrm{pk}_1, \mathrm{sk}_1)$, and returns AcraStruct: $\mathrm{AS}^{\mathrm{R}} = \mathrm{CreateAS}(\mathrm{R}, \mathrm{pk}_1)$ by performing the following steps:

   - Generating ephemeral ECDH key pair: $(pk_2, sk_2) = GenKeyPair()$;
   - Generating random symmetric key: $K = GenKey()$;
   - Encrypting R: $R^K = E(R, K)$;
   - Calculating R^K length: $len_{R^K} = LENGTH(R^K)$, where $LENGTH$ object is a function that takes a byte array and calculates its length;
   - Wrapping K: $K^{pk_1} = W(K, pk_1, sk_2)$;
   - Discarding sk_2;
   - Returning AcraStruct: $AS^R = Tag \,||\, pk_2 \,||\, K^{pk_1} \,||\, len_{R^K} \,||\, R^K$.

2) The algorithm DecryptAS (AcraStruct decryption) takes an AcraStruct AS^{R}, a private key sk_1 from ECDH key pair $(\mathrm{pk}_1, \mathrm{sk}_1)$, and returns plaintext R: $\mathrm{R} = \mathrm{DecryptAS}(\mathrm{AS}^{\mathrm{R}}, \mathrm{sk}_1)$ by performing the following steps:

   - Extracting K^{pk_1}, R^K, and pk_2 from AS^R according to the AcraStruct format;
   - Unwrapping K^{pk_1}: $K = U(K^{pk_1}, pk_2, sk_1)$;
   - Decrypting R^K : $R = D(R^K, K)$;
   - Returning R.

2.3 Functional scheme and data flow of Acra SE

The main component of Acra SE scheme is a so-called AcraServer that works as a reverse proxy. It sits between the application and the database. Figure 3 illustrates the typical data flow (including sensitive data) from two different applications (Application A and Application B) to Database and vice versa. Application issues query, AcraServer performs all cryptographic operations (if necessary) and pushes query (which may be modified) further to the Database. Database processes query and send result back. AcraServer got result from Database, performs (if necessary) cryptographic operations and pushes result (which may also be modified) further to Application.

In cases when communication channel between application and AcraServer is not secure, it's critical (!!! – triple exclamation marks in Figure 3) to enable the construction of queries by Application using AcraStructs instead of plaintext (a library that implements *CreateAS* algorithm is supplied [33]). These scenarios may take place only if the infrastructure between Application and AcraServer is trustworthy.

AcraServer supports two operation modes: standard and transparent. In standard mode encryption (creating AcraStructs) is performed on Application side. In transparent mode, encryption is performed on AcraServer. It allows easily integrating Acra SE into existing infrastructure without altering the source code of Application.

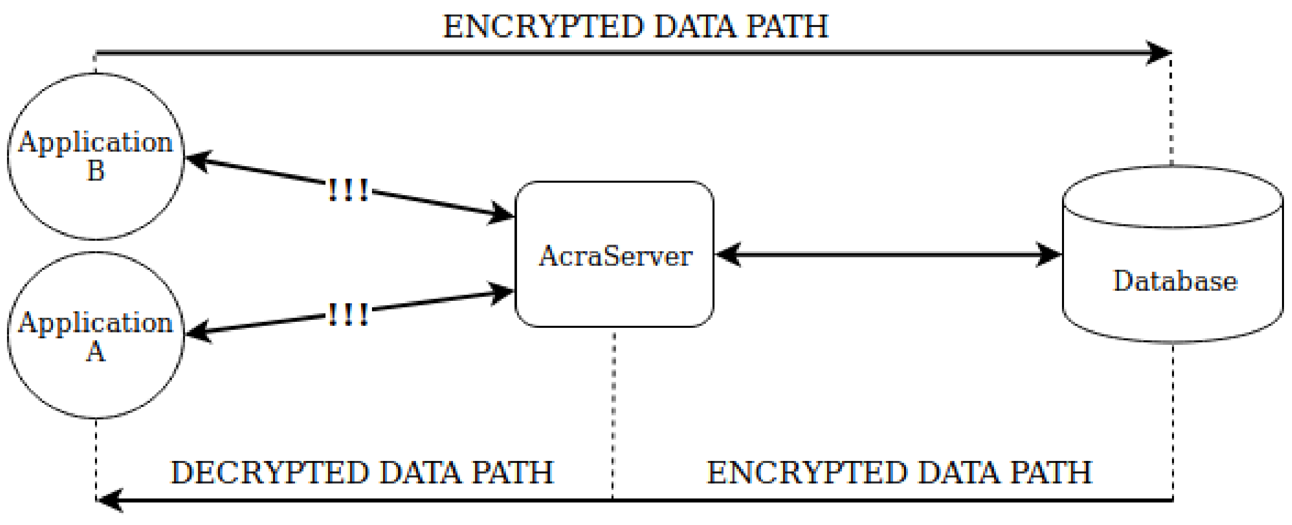

Figure 3: Data roundtrip

It should be noted that one AcraServer instance can work with multiple Applications and only one Database at the time.

2.4 Key Management

We presume that each component stores its own transport encryption keys for establishing secure communication and do not show them in Figure 4 for the simplicity's sake.

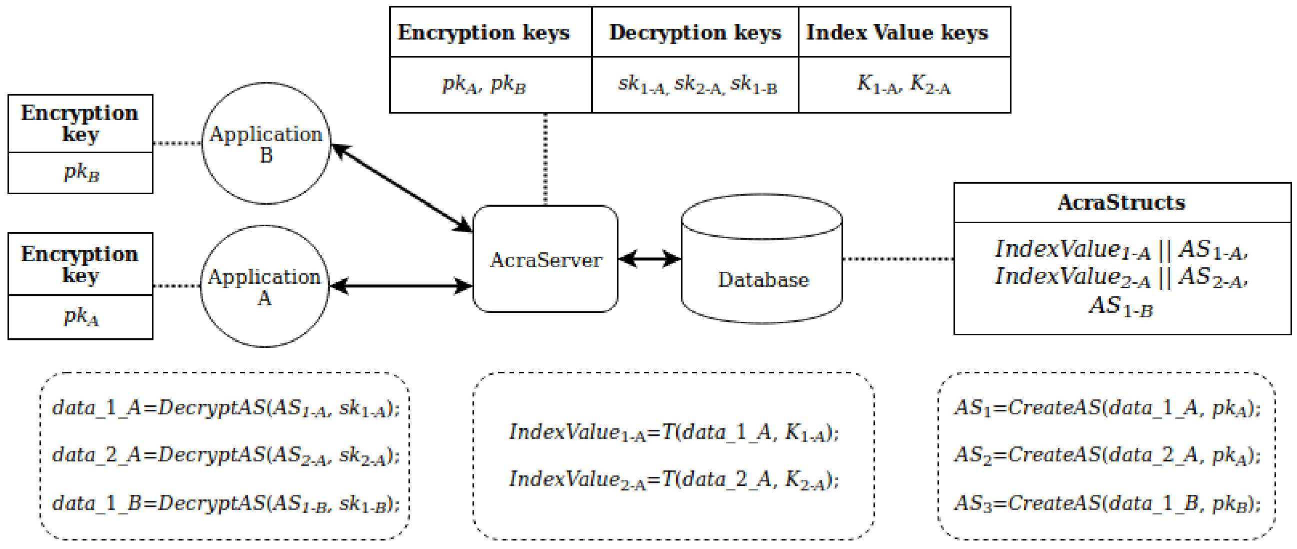
Encryption keys
Decryption keys
Index Value keys
pk_A, pk_B
sk_1-A, sk_2-A, sk_1-B
K_1-A, K_2-A
Encryption key
pk_B
Application B
Encryption key
pk_A
Application A
AcraServer
Database
AcraStructs
IndexValue_1-A || AS_1-A, IndexValue_2-A || AS_2-A, AS_1-B
data_1_A=DecryptAS(AS_1-A, sk_1-A);
data_2_A=DecryptAS(AS_2-A, sk_2-A);
data_1_B=DecryptAS(AS_1-B, sk_1-B);
IndexValue_1-A=T(data_1_A, K_1-A);
IndexValue_2-A=T(data_2_A, K_2-A);
AS_1=CreateAS(data_1_A, pk_A);
AS_2=CreateAS(data_2_A, pk_A);
AS_3=CreateAS(data_1_B, pk_B);

Figure 4: Key management in Acra SE

Note that AcraServer is a single place where the decryption keys are stored. Neither the Application nor the Database can decrypt AcraStructs. Moreover, the Database is unable to execute cryptographic operations of any kind. It can only store pre-encrypted (AcraStructs AS_{1-A}, AS_{2-A} from Application A and AS_{1-B} from Application B) or plaintext data. Only AcraServer is able to decrypt the sensitive data and calculate its blind indexes, but both Application and AcraServer are able to encrypt the sensitive data.

If AcraServer receives some plaintext data (from Application) that is configured as encryptable (i.e. a legitimate INSERT query into the encryptable table), it will encrypt it (transparently for application) and push it further to the Database. When the data (on the way back from database to application) comes to

AcraServer, it is transparently decrypted. Finally, application operates with the data according to the standard flow. Table 2 illustrates the cryptographic abilities of the components related to sensitive data.

Table 2

	Application	**AcraServer**	**Database**
Able to encrypt data (apply *CreateAS* algorithm)	+	+	–
Able to decrypt data (apply *DecryptAS* algorithm)	–	+	–
Able to create blind index (apply T algorithm)	–	+	–

2.5 Blind indexes and filtration

Blind indexes should not be confused with traditional indexes introduced for performance improvement in database management systems. Informally, blind indexes ($IndexValue_{1-A}$, $IndexValue_{2-A}$) are keyed hashes of plaintext data computed on AcraServer with the help of the BIC scheme and stored together with the encrypted data (AS_{1-A}, AS_{2-A}) on the database side (as shown in Figure 4). It is recommended to truncate values of blind indexes for security reasons (see section 4.3.2 for details). Truncated value of blind index can be treated as Bloom filter [49]. This data structure offers a compact probabilistic way to represent a set that can result in false positives, but never in false negatives. In case of data querying, this means that greater blind index truncation leads to greater amount of collisions – when two different records have two equal truncated values of blind indexes. Consequently, non-relevant encrypted records may appear in SELECT query response which can be a problem in non-proxy solutions. In our case, AcraServer (located between application and database) can resolve those issues by filtering non-relevant records from the response. Finally, the application only obtains relevant decrypted data.

Note that filtering functionality is limited by filter configuration. Filtering is currently only supported for PostgreSQL database for SELECT queries with single comparison WHERE expressions.

2.6 Secure search over encrypted data

Let’s suppose, Application A needs to perform a search over its AcraStructs AS_{1-A}, AS_{2-A}. In this case, blind indexes ($IndexValue_{1-A}, IndexValue_{1-B}$) should be provided, in order to enable the Database to distinguish AcraStructs without decrypting them. Table 3 provides an example of the encryption configuration that should be stored in the AcraServer’s memory.

Table 3

Tables in Database	**Columns**	**Encryption**	**Searching**
Employees	emp_no	ENABLED	ENABLED
	first_name		
	last_name		
Departments	emp_no	ENABLED	ENABLED
	from_date	DISABLED	DISABLED
	to_date	DISABLED	DISABLED
Salaries	emp_no	ENABLED	DISABLED
	salary	ENABLED	ENABLED

Acra SE provides secure search over encrypted data by implementing modifications of INSERT, UPDATE, and SELECT queries that were created and sent to the Database by the Application. Note that encryption (and secure search) functionality of AcraServer can be configured on the per-column basis. This means that each table in the Database can be encrypted completely (each column), partially (some columns are encrypted, some are not), or be completely unencrypted.

In configuration above, the “Employees” table is fully encrypted and the search functionality is enabled for all columns, while the “Departments” table has an encrypted column “emp_no” with search functionality enabled along with unencrypted columns “from_date” and “to_date”. The “Salaries” table contains an encrypted “emp_no” column with the search functionality disabled and an encrypted “salary” column with the search functionality enabled.

Secure search requires an introduction of two procedures: 1) secure upload of the encrypted data with indexes to the untrusted storage provider and 2) secure retrieving of the requested data from the storage provider.

2.6.1 Secure Upload (SUpload)

Secure Upload is a modification of typical INSERT and UPDATE queries. It can be expressed by an algorithm *SUpload* that takes: an input query Q_{in}^{data} (where $data$ is a sensitive data), ECDH key pair of the Application (pk, sk), a blind index key K_{index} and returns a threefold: $Q_{out}^{data}, changed, error = SUpload(Q_{in}^{data}, pk, sk, K_{index})$, where Q_{out}^{data} is a query that should be transferred to the Database, binary flag $changed$ that indicates if $Q_{in}^{data} == Q_{out}^{data}$, and error message $error$.

AcraServer evaluates the output from *SUpload* in the following way: if $error$ is not null, AcraServer logs $error$ and notifies Application. Nothing is transferred to the Database. If $changed$ is true, Q_{out}^{data} is transferred to the Database; otherwise, Q_{in}^{data} is transferred to the Database.

Let’s formally define *SUpload* algorithm, step-by-step (for simplicity’s sake, we’ll do it without the $error$ output parameter. If an error occurs on any step, as it was mentioned above, nothing is being transferred to the Database):

1) Check if Q_{in}^{data} is an INSERT or UPDATE query and is addressed to the encryptable column;
2) Go over the Q_{in}^{data} structure (according to the query type) and check if $data$ is a valid AcraStruct.
   If it’s true, perform:
   - $\mathrm{data_{plain}} = \mathrm{DecryptAS}(\mathrm{AS}, \mathrm{sk})$;
   - $\mathrm{UT_{data_{plain}}} = \mathrm{T}(\mathrm{data_{plain}}, \mathrm{K_{index}})$;
   - $\mathrm{newdata} = \mathrm{FuncID} \,||\, \mathrm{UT_{data_{plain}}} \,||\, \mathrm{AS}$

     (where FuncID is an identifier of chosen BIC scheme);
   - $\mathrm{Q_{in}^{data}} = \mathrm{Q_{in}^{newdata}}$.

   Otherwise, perform:
   - $\mathrm{UT_{data}} = \mathrm{T}(\mathrm{data}, \mathrm{K_{index}})$;
   - $\mathrm{AS^{data}} = \mathrm{CreateAS}(\mathrm{data}, \mathrm{pk})$;
   - $\mathrm{newdata} = \mathrm{FuncID} \,||\, \mathrm{UT_{data}} \,||\, \mathrm{AS^{data}}$;
   - $\mathrm{Q_{in}^{data}} = \mathrm{Q_{in}^{newdata}}$;
   - Set $\mathrm{Q_{out}^{data}} = \mathrm{Q_{in}^{data}}$ and return $\mathrm{Q_{out}^{data}}$.

The pseudocode that illustrates the execution of *SUpload* is supplied in Appendix A.

2.6.2 Secure Select (SSelect)

Secure Select is a modification of a typical SELECT query. It can be expressed by the algorithm *SSelect* that takes: query $Q_{in}^{data,colIDs}$(where $data$ is a searchable data, $colIDs$ is a set of searchable column identifiers) from Application, ECDH key pair of the Application (pk, sk), a blind index key K_{index} and returns threefold: $Q_{out}^{data,colIDs}, changed, error = SSelect(Q_{in}^{data,colIDs}, pk, sk, K_{index})$, where $Q_{out}^{data,colIDs}$ is the query that should be transmitted to the Database, binary flag *changed* that indicates if $Q_{in}^{data} == Q_{out}^{data}$ and error message $error$.

AcraServer evaluates the output from *SSelect* in the same way as for *SUpload*.

Let's formally define the *SSelect* algorithm, step-by-step (again, without $error$ output parameter for simplicity):

1) Check if $Q_{in}^{data,colIDs}$ is a SELECT query and is addressed to an encryptable column;
2) Go over the $Q_{in}^{data,colIDs}$ (according to the query structure) and check if $data$is a valid AcraStruct. If it's true, perform:
   1) $data_{plain} = DecryptAS(AS, sk)$;
   2) $UT_{data_{plain}} = T(data_{plain}, K_{index})$;
   3) Change each column identifier in WHERE expression:
      - $\mathrm{newcolIDs_1} = \mathrm{SubStr(colID_1, 1, MAC\_LEN)}$,
      - $\mathrm{newcolIDs_2} = \mathrm{SubStr(colID_2, 1, MAC\_LEN)}$,

        …
      - $\mathrm{newcolIDs_n} = \mathrm{SubStr(colID_n, 1, MAC\_LEN)}$, where n is the length of colIDs and SubStr is a Database-specific function for manipulating strings [50, 51];
   4) $newdata += MacFuncID \,||\, UT_{data_{plain}}$;
   5) $newcolIDs = (newcolIDs_1, newcolIDs_2, \ldots, newcolIDs_n)$;
   6) $Q_{in}^{data,colIDs} = Q_{in}^{newdata,newcolIDs}$.

   Otherwise, perform:
   1) $UT_{data} = T(data, K_{index})$;
   2) Change each column identifier in :
      - $\mathrm{newcolIDs_1} = \mathrm{SubStr(colID_1, 1, MAC\_LEN)}$,
      - $\mathrm{newcolIDs_2} = \mathrm{SubStr(colID_2, 1, MAC\_LEN)}$,

      …
      - $\mathrm{newcolIDs_n} = \mathrm{SubStr(colID_n, 1, MAC\_LEN)}$.
   3) $newdata = MacFuncID \,||\, UT_{data}$;
   4) $newcolIDs = (newcolIDs_1, newcolIDs_2, \ldots, newcolIDs_n)$;
   5) $Q_{in}^{data,colIDs} = Q_{in}^{newdata,newcolIDs}$.
3) Set $Q_{out}^{data,colIDs} = Q_{in}^{data,colIDs}$ and return Q_{out}^{data}.

The pseudocode that illustrates the execution of *SSelect* is in Appendix B.

3. IMPLEMENTATION

Acra suite is written in the Go programming language. Acra is specifically designed for web and mobile applications with centralised data storage, including with distributed, microservice-rich applications. Enabling Acra in an existing infrastructure is simple and requires integration of a single-function client library (which is available for Ruby, Python, Go, C++, NodeJS, PHP, Objective-C, Swift, Java programming languages) with the application and deployment of the AcraServer (via Docker, pre-built binaries, or building from source) with optional components. Acra SE functionality is implemented as a separate module of AcraServer.

3.1 Implementation of low-level cryptographic schemes

We use the crypto/rand package from Go standard library that implements a cryptographically secure pseudorandom number generator [52] and ECDH key generator with elliptic curve NID_X9_62_prime256v1 from Themis cryptographic library [53] as *CKG* scheme.

We use Themis' Secure Cell (in Seal mode) cryptosystem as the *AE* scheme (with AES-256 in GCM mode "under the hood").

We use Secure Message cryptosystem from Themis as *KW* scheme. Secure Message is similar to elliptic curve integrated encryption scheme with the following algorithms: ECDH with elliptic curve NID_X9_62_prime256v1, Key derivation function [54], based on HMAC-SHA256 and AES-256 in GCM mode cipher.

We use HMAC-SHA256 algorithm from Go programming language standard library [55] as *MAC* scheme.

We use the well-known SSL / TLS protocol implementation from the Go programming language standard library [56] for *SC* scheme.

3.2 Additional features

The additional features are the competitive advantages of Acra SE. These features enable a rather easy integration of Acra SE into existing infrastructures and increase the practical security level, making it a complete production-ready security tool.

The list of mentioned features:

- *Authentication* (two-level access control list for application and user authentication);
- *Query policy* (a separate SQL firewall module);
- *Intrusion detection* (poison records [48]);
- *Key management* (key rotation utility);
- *Monitoring and observability* (*logging, metrics,* and *tracing*. There are three different logging formats supported by Acra: plaintext, CEF [57], JSON [58]. Logs are compatible with various external log analysing tools and SIEM systems [59]. Metrics of AcraServer to Prometheus [60] are also supported).

3.3 Performance evaluation

We evaluated the performance of Acra SE with a help of a specially developed benchmarking application [26]. All measurements have been taken on a single desktop machine with Intel Core i7, 4000 MHz (12 cores), 16 GB RAM running Ubuntu 18.04 LTS x64 operating system. The benchmarking application was implemented in Go programming language (v. 1.11). The storage server ran PostgreSQL (v. 11) and was wrapped into a Docker container. The source code of application, Docker Compose file of the database, and data material used for evaluation are stored in Acra GitHub repository [26].

The database consisted of one table with a sequence generator created with the following SQL queries:

1) `DROP TABLE IF EXISTS test_raw;`
2) `DROP SEQUENCE IF EXISTS test_raw_seq;`
3) `CREATE SEQUENCE test_raw_seq START 1;`
4) `CREATE TABLE test_raw (id INTEGER PRIMARY KEY DEFAULT nextval(‘test_raw_seq’), plaintext BYTEA, ciphertext BYTEA).`

A standard functional index used for measurements of different blind index sizes was created with the following SQL pseudo-query:

```
CREATE INDEX IF NOT EXISTS test_raw_ciphertext_secure_index_idx ON test_raw (substr(ciphertext, 1, secureIndexSize));
```

AcraServer was running in transparent mode (when an application issues a query with plaintext keywords and AcraServer encrypts them) and was configured to encrypt ‘ciphertext’ column and perform secure search on it.

An example of INSERT pseudo-query issued by application:

```
INSERT INTO test_raw (plaintext, ciphertext) VALUES (input, input)
```

An example of SELECT pseudo-query issued by application:

```
SELECT * FROM test_raw WHERE ciphertext=input
```

To evaluate the performance, we measured the total time (query latency) taken by the *SUpload* and *SSelect* procedures with a help of the following experiment:

1) We generated input dataset (denoted as *R*) with *n*=50000 items by randomly taking values from the provided data material (a collection of 25000 unique emails);
2) Inserted *R* into the database;
3) Selected a non-existing row with ciphertext value (‘\xFFFFFFFFFFFFFFFFFFFFFF’) from the database and checked that no rows were fetched;
4) Inserted (1 time) a new row r_1 with the same plaintext and ciphertext value (‘\x62614C494E31353247444F43717765727479313233343536373839304068 6F746D61696C2E636F6D’) into the database, then selected r_1 and checked that exactly 1 row was fetched;
5) Inserted (10 times) a new row r_2 with the same plaintext and ciphertext value (‘\x677370723233366 56E747A61732E627261646C6579303938373635343340796168 6F6F2E636F6D’) into the database, then selected r_2 and checked that exactly 10 rows were fetched.

Such experiment was carried out on two different datasets with 350 and 25000 keyword universe size respectively.

Table 4 summarizes the results (the average of 3 runs) of the 2nd step – *SUpload* procedure that used INSERT queries with single value.

Table 4

Keyword universe size	Time PostgreSQL (s)	Blind index size (**bytes**)	Time Acra SE				Relative overhead
			Encryption (calls / time in **s**)	Blind index calculation (calls / time in **ms**)	Other Acra SE operations (time in **s**)	Total (time in **s**)	
350	707.1668	2	50000 / 57.6947	50000 / 620.1667	679.0150	737.3299	~4%
		15	50000 / 57.7494	50000 / 617.1465	736.5025	794.8690	~12%
25000	691.9133	2	50000 / 59.1564	50000 / 619.2667	648.3589	708.1346	~2%
		15	50000 / 58.2443	50000 / 625.7523	670.7465	729.6166	~5%

One can see that encryption takes up 7% – 8% of the time, blind index calculation takes up to 1%, and other operations (network communications with database and client application) take 92% – 93%. It is worth noting that a significant number of issued queries (50000) causes noticeable load on the network subsystem of the operating system. This can explain the spread of relative overhead values obtained in the measurements.

The approximate value of relative overhead is within 2% – 12%.

Table 5 summarizes the results (average of 10 runs) of the 2nd step – *SUpload* procedure that used INSERT queries with 1000 values (so-called BULK INSERT).

Table 5

Keyword universe size	Time PostgreSQL (s)	Blind index size (**bytes**)	Time Acra SE				Relative overhead
			Encryption (calls / time in **s**)	Blind index calculation (calls / time in **ms**)	Other Acra SE operations (time in **s**)	Total (time in **s**)	
350	2.5993	2	50 / 23.7783	50000 / 230.4363	5.2013	29.2100	~1024%
		15	50 / 23.7612	50000 / 232.5844	4.5905	28.5843	~1000%
25000	2.3125	2	50 / 23.6542	50000 / 229.2123	4.7324	28.6158	~1137%
		15	50 / 23.8756	50000 / 240.4962	4.5991	28.7152	~1142%

One can see that encryption takes up 81% – 83% of the time, blind index calculation takes up to 1%, and networking operations take 16 – 18% of the total Acra SE execution time.

Note that in this case only 50 INSERT queries (compared with 50000 queries in the previous measurement) were issued by benchmarking application, so networking operations cause less but still noticeable deviations of the result values.

The relative overhead is approximately one order of magnitude.

Table 6 summarizes the results (an average of 10000 runs) of the 3rd step – *SSelect* procedure that returns nothing.

Table 6

Keyword universe size	Time PostgreSQL (**ms**)	Blind index size (**bytes**)	Time Acra SE					Relative overhead
			Decryption (calls / time in **ms**)	Apply filtration (calls / time in **µs**)	Blind index calculation (calls / time in **µs**)	Other Acra SE operations (time in **ms**)	Total (time in **ms**)	
350	0.1840	2	-	-	1 / 41.9543	0.4933	0.5353	~191%
		15	-	-	1 / 39.1419	0.4623	0.5014	~172%
25000	0.1770	2	-	-	1 / 37.7230	0.4397	0.4774	~170%
		15	-	-	1 / 38.6630	0.4577	0.4964	~180%

One can see that in this case, networking operations take up 92% of time, while blind index calculation takes up the remaining 8% of the total Acra SE execution time. It's obvious that in this case, no decryption and no filtration operations should be performed.

The approximate value of relative overhead is within 170% – 191%.

Table 7 summarizes the results (average of 10000 runs) of the 4th step – *SSelect* procedure that returns exactly 1 relevant row.

Table 7

Keyword universe size	Time PostgreSQL (**ms**)	Blind index size (**bytes**)	Time Acra SE					Relative overhead
			Decryption (calls / time in **ms**)	Apply filtration (calls / time in **µs**)	Blind index calculation (calls / time in **µs**)	Other Acra SE operations (time in **ms**)	Total (time in **ms**)	
350	0.1920	2	1 / 0.4988	-	1 / 32.2792	0.5747	1.1058	~476%
		15	1 / 0.4971	-	1 / 28.9147	0.5700	1.0960	~471%
25000	0.1920	2	3 / 1.3528	2 / 73.6460	1 / 30.6646	0.6666	2.1237	~1006%
		15	1 / 0.500	-	1 / 32.3181	0.5797	1.1120	~479%

One can see that in case of dataset with 25000 keyword universe size and 2 bytes blind index size, decryption takes 64% out of the total Acra SE execution time compared to 45% in other time measurements. Traces show that there were 3 decryption function calls instead of the expected 1.

Also, it's worth pointing out that there were 2 "apply filtration" function calls. This is explained by the fact that, in this case, the result of SELECT query contained 2 non-relevant rows that were filtered by AcraServer. Note that filtering is performed after decryption since AcraServer should evaluate plaintext values of the rows in result.

Finally, we got an approximate value of the relative overhead, which is within 471% – 1006%.

Table 8 summarizes the results (average of 10000 runs) of the 5th step – *SSelect* procedure that returns exactly 10 relevant rows.

Table 8

Keyword universe size	Time PostgreSQL (**ms**)	Blind index size (**bytes**)	Time Acra SE					Relative overhead
			Decryption (calls / time in **ms**)	Apply filtration (calls / time in **µs**)	Blind index calculation (calls / time in **µs**)	Other Acra SE operations (time in **ms**)	Total (time in **ms**)	
350	0.2410	2	10 / 4.2650	-	1 / 29.7211	0.9265	5.2212	~2066%
		15	10 / 4.2619	-	1 / 28.9147	0.9682	5.2590	~2082%
25000	0.2550	2	14 / 5.8788	4 / 69.5195	1 / 30.0456	0.9842	6.9626	~2630%
		15	10 / 4.2538	-	1 / 28.9155	0.9051	5.1878	~1934%

Again, one can see blank decryptions and filtering in a dataset with 25000 keyword universe size and 2 bytes blind index size.

The approximate value of relative overhead is within 1934% – 2630%.

According to results in Tables 4 – 8, it can be reported that Acra SE has 2 orders of magnitude slowdown compared to plain PostgreSQL. Most significant impact is caused by cryptographic operations (encryption in *SUpload* and decryption in *SSelect*). Also, it should be noted that results of SELECT queries may contain non-relevant rows at small sizes of blind index (1 - 2 bytes). This may lead to blank decryptions and consequently to further performance degradation.

4. SECURITY CONSIDERATIONS

This section contains practical security evaluation of our SE scheme implemented in Acra [32]. All Acra's searchable encryption security properties are quite similar to the security properties of CipherSweet [29], which introduces the risk of partially-known plaintext attacks. We expect the presence of similar risks, however, an introduction of a proxy between the application and the database allows for a separation of duties that guarantees that there is no cryptographic key to leak from an application (neither for blind indexes nor for data). Moreover, security functions are added in a transparent and convenient manner at the cost of introduction of an additional infrastructure component.

4.1 Security assumptions

1) The AcraServer is trusted (it works in a separate, computationally isolated environment and isolation mechanism is also trusted);
2) Cryptographic keys (except for public keys) never leave the AcraServer;
3) Communication between the AcraServer and the application is secure (encrypted and authenticated);
4) An attacker is only able to see the ciphertexts and blind indexes.

This implies that the database server is on one piece of physical hardware, the AcraServer that accesses the database is on a different separate piece of physical hardware and that the keys never get transferred to the database server or application. The physical deployment of the application doesn't matter.

Separating the AcraServer, the database and the application in the cloud lowers the security guarantees, because it relies on operating system/cloud vendor execution isolation mechanisms.

4.2 Attacker capabilities

It's assumed that the weakest attackers only have read-only access to the SQL database in question (i.e. via SQL injection of an existing SELECT query). Some attackers MAY have read-write access permissions to the database. Some attackers MAY use the application legitimately (i.e. as normal users) and encrypt some plaintexts to attempt an attack on the encryption. Some attackers MAY use the application on their own (i.e. as a root user). Some attackers MAY use AcraServer (Figure 1) legitimately (i.e. as a normal user) and encrypt some plaintexts to attempt an attack on both the encryption and the blind indexes.

4.3 Practical security evaluation

We've already mentioned that the security model of CipherSweet (and, consequently, of Acra's secure search) introduces the risk of partially-known plaintext attacks. In this attack type, the adversary's goal is to reconstruct as much of the client's indexed encrypted data as possible, primarily by learning the mapping of keywords to their encrypted versions. This keyword recovery can then be used to partially reconstruct the plaintexts of the stored data [12]. Leakage inference (LI) attacks [8, 12] are the most powerful examples of partially-known plaintext attacks because the adversary (i.e. honest-but-curious or malicious database server [61]) appears in a very favourable position – they can see (to various degrees) the search and the access patterns. The only way to mitigate those attacks is to hide the search and access patterns from an adversary, but as it's described in [23], in practice, it's hard to achieve due to the real-world threats like stealing a disk, performing an SQL injection, or compromising the OS. All this may lead to revealing of the previous queries because each modern database management system keeps logs, caches, data structures, and other metadata to adapt the system to the workload and help manage its performance. We can state that this (search / access pattern availability to the adversary) is one of the main searchable encryption threats which can be mitigated by blind index truncation which will be discussed below.

In the next subsections, we provide an informal security evaluation with a respect to the current academic understanding of searchable encryption security. We also try to analyze the real consequences of leakage-inference attacks on SQL databases.

4.3.1 Leakage inference attacks

Current leakage inference (LI) attacks try to recover either a set of queries or the data stored in the database. All the current LI attacks assume the possession of some prior knowledge by an attacker. Usually it is the information about the stored data but it may include information about the past queries. Note that some attacks (i.e. Active Attacks) have a linear complexity and significant efficacy. For example, Count Attack yields a 40% keyword recovery rate with 80% of dataset known to attacker. It performs well if the keyword universe size does not exceed 5000. A Hierarchical-Search Attack is an extension of the Count Attack, which yields the same 40% keyword recovery rate under a condition that (at least) 40% of the data leaks. The cost of such reduction is the possibility of an attacker injecting a set of constructed records. The detailed descriptions of the mentioned attacks can be found in [12, 14, 16, 17, 62, 63].

Table 9 (taken from [8]) summarizes the most effective LI attacks.

Table 9

Attack name	**Required leakage**	**Required conditions**		**Attack efficacy**	
		Data injection	**Prior knowledge**	**Runtime (in number of keywords)**	**Tested keyword universe**
Communication Volume Attack [12]	L_1	–	K_2	> Quadratic	< 500
Binary Search Attack [14]	$L_1 L_2$	+	K_1	Linear	< 500
Access Pattern Attack [12]	$L_1 L_2$	–	K_2	Quadratic	< 500
Partially Known Documents [16]	$L_1 L_2$	–	K_4	> Quadratic	> 1000
Hierarchical-Search Attack [14]	$L_1 L_2$	+	K_4	Quadratic	> 1000
Count Attack [16]	$L_1 L_2$	–	K_5	Quadratic	> 1000
Graph Matching Attack [17]	$L_1 L_2$	–	K_3	> Quadratic	500 … 1000
Frequency Analysis [62]	L_3	–	K_3	Linear	< 500
Active Attacks [16]	L_3	+	K_3	Linear	> 1000
Known Document Attacks [16]	L_3	–	K_4	Linear	> 1000
Non-Crossing Attacks [63]	L_4	–	K_3	Linear	> 1000

The required leakage is ranked (from least to most damaging) as follows:
L_1 – structure leakage (properties of an object; i.e. string length, cardinality of a set);
L_2 – identifier leakage (pointers to object so that their past/future accesses are identifiable);
L_3 – equality leakage (information whether two objects have the same value);
L_4 – order / contents leakage (numerical or lexicographical order of objects).

The prior knowledge is ranked (from least to most damaging) as follows:
K_1 – keyword universe;
K_2 – distributional knowledge of queries;
K_3 – distributional knowledge of dataset;
K_4 – contents of a subset of dataset;
K_5 – contents of full dataset;

To sum it up, the existence of LI attacks introduces a very important security parameter – keyword universe size (i.e. data entropy). If the stored data pieces have a very small keyword universe size (i.e. HIV status of a patient in a database, which can be either “positive” or “negative”) they are more leakage-prone. In this case, the common strategy (if such data still requires indexing) may be to create a compound index that could point to a composite of low-entropy sensitive data concatenated with other high-entropy data.

4.3.2 Security of blind indexes

The approach of blind index truncating is one of the most effective methods of leakage-inference attack mitigation, since it hides the search pattern in adjustable manner at the cost of query accuracy. When blind index is truncated to a specified number of bits, it can be treated as Bloom filter [49] for database lookups. Bloom filter was conceived by Burton H. Bloom in 1970. This data structure supports adding element to the set and querying for element membership in the probabilistic set representation. Bloom filter may claim an element to be part of the set when it was not inserted but never report an inserted element to be absent from the set. The more elements are added to a Bloom filter, the higher the probability that the query operation reports false positives.

As soon as Bloom filters allow false positives (i.e. by partial hash collisions on the non-truncated part of the hash), truncating index leads to non-relevant records presence in the query result. Thus, these prefix collisions cease to be collisions that reveal the fact of plaintexts probable duplication [29]. There is an exact formula for determining the safe upper bounds for the amounts of information that one can safely leak without also revealing duplicate plaintexts (and allowing the attacker to rule out false positives). Let’s provide a formal definition:

Definition 2. The main security parameter of blind indexes is an upper bound plaintext leakage (average number of rows returned) that can be expressed as:

$$C = R \,/\, 2^{-S},$$

where $S = \sum_{i=0}^{n} min(L_i, K_i)$, n – is a number of blind indexes, L_i – a blind index length (in bits), K_i – the keyspace of the input domain (in bits), R – a number of encrypted records that use those blind indexes.

A practical recommendation is to choose parameters in such a way that $2 \le C < \sqrt{R}$. If $C < 2$, attacker is able to infer that some plaintexts are identical (which breaks the standard security notion of the scheme). Otherwise, if $C > \sqrt{R}$, too much collisions will be introduced, and no performance benefit will be achieved.

A safe upper limit for the plaintext leakage is, actually, the highest number of the expected coincidences. Generally, the number n and the length of each index L_i should be minimized. The more indexes are created, the more confidence an attacker gains. At the same time, larger indexes are more useful than shorter indexes.

4.4 Risk modelling

Now, we evaluate the potential security consequences of components' compromising. As a result, the compromised component becomes malicious (worst case scenario). Table 10 demonstrates the worst (in our opinion) consequences and a variety of possible attacks, depending on the component.

We use following abbreviations: SQLi — SQL injection; CPA — chosen-plaintext attack; DoS — denial of service; LIA — leakage inference attack; COA — ciphertext-only attack, CMA — chosen-message attack.

Table 10

Malicious component	Consequences	Attacks
Application	Access to application's flow	SQLi, CPA, DoS
Database	Access to stored blind blind indexes, encrypted sensitive data, search pattern and access pattern	LIA, COA, DoS
Application + Database	Access to application flow, stored blind indexes, encrypted sensitive data, search pattern and access pattern	SQLi, CPA, LIA, COA, DoS, CMA
AcraServer	Access to all plaintext sensitive data (worst case)	Revealing / Leaking of all the sensitive data and decryption keys

4.4.1 SQLi mitigation

SQL injection is an old but still relevant and prevalent attack [64]. Acra provides security against SQL injections through an introduction of a firewall mechanism (AcraCensor) as part of AcraServer. AcraCensor enables transparent filtration (based on denylist / allowlist approach) of SQL queries received from the application. This way, we achieve a strictly expected behaviour of an application and prevent the possible damage if it becomes malicious.

4.4.2 CPA mitigation

Chosen plaintext attack is a standard attack model for cryptanalysis. It presumes that the attacker can obtain the ciphertexts for arbitrary plaintexts. Modern ciphers (we use AES in GCM mode) aim to provide semantic security, also known as IND-CPA, and therefore by design generally immune to chosen-plaintext attacks if correctly implemented.

4.4.3 DoS mitigation

Out of scope. Should be achieved by proper infrastructure deployment.

4.4.4 LIA mitigation

Mitigation of leakage inference attacks can be achieved through using blind index truncation according to the formula from Definition 2. In this case, the amount of revealed statistical information about the sensitive data will be insufficient for an attacker. But the cost is performance degradation because a truncated blind index (which can be treated as Bloom filter) leads to false positives – non-relevant records appear in the query result. AcraServer receives a response on query earlier than application and is able to filter those false positives.

Other vectors of LIA mitigation are described in the security recommendations (see subsection 4.5 below).

4.4.5 COA mitigation

Ciphertext-only attack is a standard attack model for cryptanalysis where it’s assumed that an attacker only has access to a set of ciphertexts. We rely on the ability of chosen ciphers to provide protection against COA. Moreover, modular scheme implies prompt substitution of any detected vulnerable cipher with non-vulnerable one.

4.4.6 CMA mitigation

Chosen-message attack is a standard attack model on MAC schemes. It’s assumed that an attacker is able to choose messages and obtain corresponding message authentication codes. Modern keying hash functions for message authentication (we use HMAC-SHA256) aim to provide security against such attacks even if the underlying hash function is not collision-free.

4.5 Practical security recommendations

The security assumptions of Acra (and usually of any other encrypted database system) can engender over-cautiousness. We provide a list of practical recommendations of our searchable encryption tool for manageable and safe usage:

1) Don’t create blind indexes for the extremely sensitive data (data that should not be exposed at all cost).
2) Create only the minimal number of blind indexes. The more indexes — the more metadata is leaked.
3) If the data that needs to be indexed is extremely sensitive and too low-entropy to be safely put in a blind index (i.e. HIV status of a patient), it can be hashed together with some other data (i.e. with a SSN). In this case, SELECT queries to records with a given HIV status and a given SSN may be supported (compound index).
4) Blind index can be truncated in order to reduce the information leak at the cost of increasing the chances of collisions (wrong SELECT results that should be filtered after decryption). It’s recommended that for any given value of R, the expression $2 \leq C < \sqrt{R}$ always remains true (see Definition 2).
5) At the very least, secure and authenticated communication must be enabled between application and AcraServer.
6) Use standard mode (instead of transparent mode) of Acra SE operating – enable the encryption on the application side straight away (instead of relying on AcraServer) to minimize plaintext lifecycle and separate duties.
7) Enable all additional security features (AcraCensor, Poison Records, encryption/decryption keys rotation).
8) Use hardware secure module to store the master key of AcraServer.

5 FUTURE WORK / WORK IN PROGRESS

5.1 N-Gram search

The current scheme only provides search capabilities for EQUAL statements. As discussed in section 4.3.1, building cryptographic primitives that accommodate range queries usually cause greater leakage and consequently may lead to more powerful leakage inference attacks.

Instead of reflecting plaintext properties in encrypted text, another approach is to split the indexed value into smaller tokens akin to full-text-search approaches — so-called N-gram search. It imposes new threats: since the overall entropy of N-Grams is limited, reconstructing protected records on a malicious database becomes easier. To mitigate that, N-Grams have to provide comparable security against existing threat model.

Currently, the authors' intuitive assumption is that if the entropy level of indexed string is N and the acceptable level of entropy against brute force is $K < N$, then we can automatically split the indexed string into smaller substrings, whose length / variability satisfy acceptable level of entropy K, and index each one of them separately.

The ultimate goal for such string splitting is secure and effective LIKE operator support which is actually a tradeoff. The number of substrings should be enough for per-letter evaluation of Levenshtein distance between encrypted string stored in the database and query keyword. Wherein, the entropy of each substring should be as high as possible for security reason (since each substring should have own blind index).

Aside from security considerations, this introduces sufficient performance / storage trade-offs and is subject to further research.

5.2 Extending EVAL's without N-Grams

Another approach would be to implement special plaintext transformation procedures to enable a broader match syntax. For example, case-sensitive matches, normalize all data to lowercase and normalize all queries to lowercase, too. For matches where variety between query and original text is slow regular expression matching: in proxy-mediated design, matching regular expressions in WHERE clauses is possible but seems abnormally slow. Acra has to render all possible values of regular expression, generate hashes and request database with WHERE = list of hashes. This seems to replicate some of the basic database functionality, but for cases where cost of retrieval of queries is not important / sufficient computing resources are devoted towards Acra, such design is possible.

These efforts require additional formal security analysis before actual implementation.

5.3 Improving entropy control on application side

ORM: Acra provides deeper integration scenario into application flow that improves overall security. Having integration library on client-side, integrated into high-level language, Acra could automatically detect the expected level of entropy via detecting variable type in ORM and, upon evaluating potential variable type, deciding whether it can hold sufficient entropy or not, freeing user from making security-critical decisions.

Data learning: by running a large amount of database queries / responses through simple analytical application, complete field of possible values, and their differences can be mapped out, providing exact entropy level detected. In such case low-entropy data can be detected and managed appropriately.

Blind indexes learning: by running special planner that detects 1) number of blind indexes for a given field and 2) boundaries for truncation of the newly created blind index (to follow security recommendations from 4.3.2 and 4.5).

All these approaches would enable Acra to reject low-entropy data as a source for blind index. The latter approach also allows mapping the balance between the number of collisions in bloom filter cutoff and entropy programmatically, thus providing manageable collision level: sufficient to make brute-force useless, but not penalizing the performance too much.

Understanding of the sensitive data's nature is especially important, since many long numeric identifiers, such as SSN or credit card number, follow an algorithm to compose one, sometimes including not just composition rules (where various facts of the real world are encoded deterministically into ID), but also checksum byte / bit. Entropy of such identifiers is further limited, and security of blind indexing them should be assessed separately.

6 CONCLUSIONS

In this paper, we have proposed a searchable encryption scheme for SQL databases. The proposed solution builds upon the already existing searchable encryption by CipherSweet, which is adapted to a proxy-mediated scheme. Along with the secure search, our solution provides strict separation of duties that guarantee no leakage of cryptographic key from application, proper key management and additional security features that corresponds to real-world threats to sensitive data stored externally.

7 ACKNOWLEDGEMENTS

The authors would primarily like to thank Scott Arciszewski and his project CipherSweet for the implementation of the blind indexing approach to searchable encryption. The authors also note that the shaping up and completion of this paper describing Acra SE would be impossible without the individual contributions and collective work of the following individuals: Anastasiia Voitova (*anastasi@cossacklabs.com*), Karen Sawrey (aka Alona Ivanova) (*karen@cossacklabs.com*), Dmytro Shapovalov (*shad@cossacklabs.com*). Special thanks to Ignat Korchagin (*ignat.korchagin@gmail.com*) and Anton Tolchanov for the insightful input on security and comments. Thank you to cryptography & software engineer Ruslan Kiyanchuk (*ruslan.kiyanchuk@gmail.com*) and cryptographer Jean-Philippe Aumasson (*jeanphilippe.aumasson@gmail.com*) for the overview and improvement suggestions.

8 REFERENCES

1) Enterprise Database Applications and the Cloud: A Difficult Road Ahead / M. Stonebraker, A.Pavlo, R. Taft, M. L. Brodie / http://people.csail.mit.edu/rytaft/cloud.pdf.

2) General Data Protection Regulation (GDPR) home / https://eugdpr.org/.

3) GDPR. Article 32 / http://www.privacy-regulation.eu/en/article-32-security-of-processing-GDPR.htm.

4) Practical Techniques for Searches on Encrypted Data / D. X. Song, D. Wagner, A. Perrig / https://people.eecs.berkeley.edu/~dawnsong/papers/se.pdf.

5) R. Powers and D. Beede, "Fostering innovation, creating jobs, driving better decisions: The value of government data," Office of the Chief Economist, Economics and Statistics Administration, US Department of Commerce, July 2014.

6) Executing SQL over Encrypted Data in the Database-Service-Provider Model / H. Haciǧűműs, B.Iyer, C. Li, S. Mehrotra / https://www.ics.uci.edu/~chenli/pub/sigmod02.pdf.

7) Cryptographically Enforced Search Pattern Hiding / C. Bosch / https://ris.utwente.nl/ws/portalfiles/portal/6031163/thesis_C_Boesch.pdf.

8) SoK: Cryptographically Protected Database Search / B. Fuller, M. Varia, A. Yerukhimovich, E.Shen, A. Hamlin, V. Gadepally, R. Shay, J. D. Mitchell, R. K. Cunningham / https://arxiv.org/pdf/1703.02014.pdf.

9) A Survey of Provably Secure Searchable Encryption / C. Bosch, P. Hartel, W. Jonker, A. Peter / https://dl.acm.org/citation.cfm?id=2636328.

10) Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions / R. Curtmola, J. Garay, S. Kamara, R. Ostrovsky / https://eprint.iacr.org/2006/210.pdf.

11) Predicate Privacy in Encryption Systems / E. Shen, E. Shi, B. Waters / https://www.iacr.org/archive/tcc2009/54440456/54440456.pdf.

12) Leakage-abuse attacks against searchable encryption / D. Cash, P. Grubbs, J. Perry, T. Ristenpart / https://eprint.iacr.org/2016/718.pdf.

13) Breaking web applications built on top of encrypted data / P. Grubbs, R. McPherson, M. Naveed, T.Ristenpart, V. Shmatikov / https://eprint.iacr.org/2016/920.pdf.

14) Leakage-abuse attacks against order-revealing encryption / P. Grubbs, K. Sekniqi, V.Bindschaedler, M. Naveed, T. Ristenpart / https://eprint.iacr.org/2016/895.pdf.

15) Access pattern disclosure on searchable encryption: Ramification, attack and mitigation / M.S.Islam, M. Kuzu, M. Kantarcioglu / https://pdfs.semanticscholar.org/9614/87973d4b33f96406fddbfcf1235dc587571f.pdf.

16) Generic attacks on secure outsourced databases / G. Kellaris, G. Kollios, K. Nissim, A. O'Neill / https://privacytools.seas.harvard.edu/files/privacytools/files/generic.pdf.

17) Inference attacks on property-preserving encrypted databases / M. Naveed, S. Kamara, C.V.Wright / https://cs.brown.edu/~seny/pubs/edb.pdf.

18) Selective Document Retrieval from Encrypted Database / C. Bosch, Q. Tang, P. Hartel, W. Jonker / https://ris.utwente.nl/ws/files/5335662/Bosch12SDR.pdf.

19) CryptDB: Protecting confidentiality with encrypted query processing / R. A. Popa, C. Redfield, N.Zeldovich, H. Balakrishnan / https://dl.acm.org/citation.cfm?id=2043566.

20) Arx: A strongly encrypted database system / R. Poddar, T. Boelter, and R. A. Popa / https://eprint.iacr.org/2016/591.pdf.

21) Big data analytics over encrypted datasets with Seabed / A. Papadimitriou, R. Bhagwan, N.Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, S. Badrinarayanan / https://www.usenix.org/system/files/conference/osdi16/osdi16-papadimitriou.pdf.

22) Building web applications on top of encrypted data using Mylar / R. A. Popa, E. Stark, S. Valdez, J.Helfer, N. Zeldovich, H. Balakrishnan / https://www.usenix.org/system/files/conference/nsdi14/nsdi14-paper-popa.pdf.

23) Why Your Encrypted Database Is Not Secure / P. Grubbs, T. Ristenpart, V. Shmatikov / https://eprint.iacr.org/2017/468.pdf.

24) Orthogonal Security With Cipherbase / A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, R. Venkatesan / http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.362.7093&rep=rep1&type=pdf.

25) Experiences and observations on the industrial implementation of a system to search over outsourced encrypted data / P. Grofig, M. Haerterich, I. Hang, F. Kerschbaum, M. Kohler, A. Schaad, A. Schroepfer, W.Tighzert / https://subs.emis.de/LNI/Proceedings/Proceedings228/115.pdf.

26) Acra project GitHub / https://github.com/cossacklabs/acra/.

27) CipherSweet project GitHub / https://github.com/paragonie/ciphersweet/.

28) Blind seer: A scalable private DBMS / V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. D. Keromytis, S. Bellovin / https://www.cs.columbia.edu/~angelos/Papers/2014/blind_seer.pdf.

29) Highly-scalable searchable symmetric encryption with support for Boolean queries / D. Cash, S.Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner / https://eprint.iacr.org/2013/169.pdf.

30) Rich queries on encrypted data: Beyond exact matches / S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M.-C. Rosu, M. Steiner / http://sprout.ics.uci.edu/pubs/rich-queries-ESORICS15.pdf.

31) CipherSweet security question and answers on security.stackexchange.com / https://security.stackexchange.com/questions/196833/how-secure-is-the-ciphersweet-library-for-searchable-encryption-and-why-is-a-du.

32) Framework for Searchable Encryption with SQL Databases / M. Azraoui, M. Onen, R. Molva / https://www.scitepress.org/papers/2018/66661/66661.pdf.

33) Arx project GitHub / https://github.com/arx-deidentifier/arx/.

34) The Elliptic Curve Diffie-Hellman (ECDH) / R. Haakegaard, J. Lang / https://koclab.cs.ucsb.edu/teaching/ecc/project/2015Projects/Haakegaard+Lang.pdf.

35) Security Analysis of Pseudo-Random Number Generators with Input: /dev/random is not Robust / Y. Dodis, D. Pointcheval, S. Ruhault, D. Vergnaud, D. Wichs / https://eprint.iacr.org/2013/338.pdf.

36) Selecting Elliptic Curves for Cryptography: An Efficiency and Security Analysis / J. W. Bos, C.Costello, P. Longa, M. Naehrig / https://eprint.iacr.org/2014/130.pdf.

37) Authenticated Encryption in Theory and in Practice / J. P. Degabriele / http://www.isg.rhul.ac.uk/~kp/theses/JPDthesis.pdf.

38) Announcing the ADVANCED ENCRYPTION STANDARD (AES) / FIPS publication 197 / https://csrc.nist.gov/csrc/media/publications/fips/197/final/documents/fips-197.pdf.

39) The Security and Performance of the Galois/Counter Mode (GCM) of Operation (Full Version) / D.A. McGrew, J. Viega / https://eprint.iacr.org/2004/193.pdf.

40) The ChaCha family of stream ciphers / D. J. Bernstein / https://cr.yp.to/chacha.html.

41) ChaCha20 and Poly1305 for IETF protocols / https://tools.ietf.org/html/rfc7539.

42) Evaluation report on the ECIES cryptosystem / J. Stern / https://www.cryptrec.go.jp/exreport/cryptrec-ex-1016-2002.pdf.

43) Keying Hash Functions for Message Authentication / M. Bellare, R. Canetti, H. Krawczyk / https://cseweb.ucsd.edu/~mihir/papers/kmd5.pdf.

44) Public Key Infrastructure Implementation and Design / S. Choudhury, K. Bhatnagar, W. Haque / Transworld; London, New York, 2002.

45) Authenticated Diffie-Hellman Key Agreement Protocols / S. Blake-Wilson, A. Menezes / https://link.springer.com/content/pdf/10.1007/3-540-48892-8_26.pdf.

46) Special Signature Schemes and Key Agreement Protocols / C. J. Kudla / http://www.isg.rhul.ac.uk/~kp/theses/CKthesis.pdf.

47) A secure key agreement protocol using elliptic curves / C. Popescu / https://pdfs.semanticscholar.org/fa6b/2179671ab9c2c7a58bf6b4f57628585c3022.pdf.

48) Documentation Server of Cossack Labs Limited / https://docs.cossacklabs.com/.

49) Theory and Practice of Bloom Filters for Distributed Systems / S. Tarkoma, C.E. Rothenberg, E. Lagerspetz / http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.457.4228&rep=rep1&type=pdf.

50) MySQL project documentation (string functions) / https://dev.mysql.com/doc/refman/8.0/en/string-functions.html.

51) PostgreSQL project documentation (string functions) / https://www.postgresql.org/docs/9.1/functions-string.html.

52) The Go Programming Language (rand Package) / http://golang.ir/pkg/crypto/rand/.

53) Themis project GitHub / https://github.com/cossacklabs/themis/.

54) ZRTP: Media Path Key Agreement for Unicast Secure RTP/ https://tools.ietf.org/html/rfc6189.

55) The Go Programming Language hmac Package / https://golang.org/pkg/crypto/hmac/.

56) The Go Programming Language tls Package / https://golang.org/pkg/crypto/tls.

57) Common Event Format / ArcSight, Inc / https://kc.mcafee.com/resources/sites/MCAFEE/content/live/CORP_KNOWLEDGEBASE/78000/KB78712/en_US/CEF_White_Paper_20100722.pdf.

58) Understanding JSON schema / M. Droettboom / http://json-schema.org/understanding-json-schema/UnderstandingJSONSchema.pdf.

59) The Absolute Guide to SIEM / ManageEngine Log360 / https://download.manageengine.com/log-management/the-absolute-guide-to-siem.pdf.

60) The Prometheus Methodology / Lin Padgham, Michael Winikoff / https://pdfs.semanticscholar.org/34a3/77d06ec0ba72caca94b36bfa138f6c68d222.pdf.

61) Techniques in Computing on Encrypted Data / W. Chen / https://inst.eecs.berkeley.edu/~cs261/fa17/scribe/08_28_encdata.pdf.

62) All your queries are belong to us: The power of file-injection attacks on searchable encryption / Y.Zhang, J. Katz, C. Papamanthou / https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_zhang.pdf.

63) The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption / D. Pouliot, C. V. Wright / http://web.cecs.pdx.edu/~dpouliot/p1341-pouliot.pdf.

64) 2018 Verizon Data Breach Investigations Report / Verizon / http://www.documentwereld.nl/files/2018/Verizon-DBIR_2018-Main_report.pdf.

APPENDIX A

```
INPUT:
      string: input_Query
OUTPUT:
      string: output_Query
      bool: changed

changed := false
parsed_input_query := Parse(input_Query)
switch statement := parsed_input_query.(type) {
case *Insert:
  table := GetTable(statement)
    columns := GetColumns(statement)
    rows := GetRows(statement)
    for i from 0 to len(rows) {
       for j from 0 to len(rows[i]) {
          if column[j].Encryptable() {
             if !ValidAcraStruct(rows[i][j]) {
                if column[j].IsSearchable() {
                   EncryptWithSearch(rows[i][j])
                } else {
                   // secure searching is disabled for this column
                }
                changed = true
             } else {
                if column[j].IsSearchable() {
                  decryptedData := Decrypt(rows[i][j])
                  index_value := CreateIndexValue(decryptedData)
                  rows[i][j] = index_value + rows[i][j]
           } else {
              // secure searching is disabled for this column
           }
        }
     }
   statement.Rows = rows
   }
case *Update:
   // Performing data encryption according to UPDATE query structure
case *Replace:
   // Performing data encryption according to REPLACE query structure
}
output_query := statement
returnoutput_query, changed
```

`Parse` function transforms the input SQL query string into AST (abstract syntax tree) that can be further analyzed; `GetTable` function extracts the table node from the query structure; `GetColumns` function extracts a set of columns; `GetRows` function extracts a set of rows; `IsEncryptable` function checks whether a column is encryptable according to the encryption configuration; `IsSearchable` function checks whether a column is searchable according to the encryption configuration; `ValidAcraStruct` function validates the actual data in the row and checks if it's a valid AcraStruct created by Application (remember that data can come from the Application in both encrypted or unencrypted forms); `EncryptWithSearch` function encrypts data and enables the further secure search; `Decrypt` function decrypts AcraStruct; `CreateIndexValue` function calculates the UT for specified data (applies *T* algorithm) which is a blind index value.

APPENDIX B

```
INPUT:
      string: input_Query
OUTPUT:
      string: output_Query
      bool: changed
      error: error

changed := false
parsed_input_query := Parse(input_Query).(*Select)
exprs := GetComparisonExpressions(select_query.Where)
for i from 0 to len(exprs) {
   column := exprs[i].Left
   if column.IsEncryptable() {
      if column.IsSearchable() {
         exprs[i].Left = SubstrExpr{Name: column, From: 1, To: MAC_LEN }
         if ValidAcraStruct(exprs[i].Right) {
            Decrypt(exprs[i].Right)
         }
      exprs[i].Right = CreateIndexValue(exprs[i].Right)
      changed = true
      }
   }
}
select_query.Where = exprs
output_query := select_query
returnoutput_query, changed, nil
```

`Parse`, `IsEncryptable`, `IsSearchable`, `Decrypt` and `CreateIndexValue` functions work as defined in Appendix A. `GetComparisonExpressions` function extracts a set of comparison expressions from the WHERE part of the input SELECT query; `SubstrExpr` is a constructor that creates a new column identifier.