A framework for searching encrypted databases

Pedro Geraldo M. R. Alves, Diego F. Aranha

Instituto de Computação – Universidade Estadual de Campinas (Unicamp)
Av. Albert Einstein, 1251 – CEP 13083-852, Campinas/SP – Brazil

{pedro.alves, dfaranha}@ic.unicamp.br

Abstract. *Cloud computing is a ubiquitous paradigm. It has been responsible for a fundamental change in the way distributed computing is performed. The possibility to outsource the installation, maintenance and scalability of servers, added to competitive prices, makes this platform highly attractive to the industry. Despite this, privacy guarantees are still insufficient for data processed in the cloud, since the data owner has no real control over the processing hardware. This work proposes a framework for database encryption that preserves data secrecy on an untrusted environment and retains searching and updating capabilities. It employs order-revealing encryption to provide selection with time complexity in* $\Theta(\log n)$*, and homomorphic encryption to enable computation over ciphertexts. When compared to the current state of the art, our approach provides higher security and flexibility. A proof-of-concept implementation on top of MongoDB is offered and presents an 11-factor overhead for a selection query over an encrypted database.*

1. Introduction

Cloud computing is a ubiquitous paradigm. From mobile to scientific computing, the industry increasingly embraces cloud services and take advantage of their potential to improve availability and reduce operational costs [Hoffa et al. 2008, Dinh et al. 2013]. Despite this, the cloud cannot be blindly trusted since malicious parties may have full access to the server and consequently to data. This includes external entities exploiting vulnerabilities, governmental institutions requesting information, and even curious cloud administrators. The data owner has no real control over the processing hardware and therefore cannot guarantee the secrecy of data [Xiao and Xiao 2013].

The risk of confidentiality breach caused by insecure use of cloud computing is real. As examples, we can mention PRISM, a surveillance program from the USA government that forced cloud companies to provide user data [Greenwald and MacAskill 2013, Weber 2014]; the Ashley Madison case, when personal data of millions of users was leaked by malicious parties exploiting security vulnerabilities [Thomsen 2015]; and the Yahoo data breach, possibly the largest known and affecting about 500 million accounts [BBC News 2016]. These security issues contribute to a growing crisis of confidence and leads domestic and corporate users to slow down the adoption of the cloud. There are estimates that the disclosures about PRISM may imply in a reduction of financial income from U$ 35 billion to U$ 180 billion for the cloud computing market in 2016 [Miller 2014]. This could be avoided if data was encrypted by the user and kept this way all the time, staying secret to the application and the cloud.

The problem of using standard encryption in an entire database is that it eliminates the capability to select records or evaluate arbitrary functions without the cryptographic keys. This reduces the cloud to a complex and huge storage service, discarding several

advantages it offers. Searchable encryption enables the cloud to manipulate encrypted data on behalf of a client without learning information. Hence, it solves both of aforementioned problems, keeping confidentiality in regard to the cloud but retaining some of its interesting features.

This work follows the state of the art and proposes directives to the modeling of a searchable encrypted database [Popa et al. 2011, Bösch et al. 2014]. We define the main primitives of a relational algebra necessary to keep the database functional, while adding enhanced privacy-preserving properties. A set of cryptographic tools is used to construct each of these primitives. It is composed by order-revealing encryption to enable data selection, homomorphic encryption for evaluation of arbitrary functions, and a standard symmetric scheme to protect and add flexibility to the handling of general data. In particular, our proposal of a selection primitive achieves time complexity of $\Theta(\log n)$. Moreover, we provide a security analysis and performance evaluation to estimate the impact on execution time and space consumption, and a conceptual implementation that validates the framework. It works on top of MongoDB, a popular document-based database, and is implemented as a wrapper over its Python driver. Its source code was made available to the community under a GNU GPLv3 license [Alves 2016].

When compared to CryptDB [Popa et al. 2011], our proposal provides stronger security since it is able to keep data confidentiality even in the case of a compromise of the database and application servers. Since CryptDB delegates to the application server the capability to derive users' cryptographic keys, it is not able to provide such security guarantees. Futhermore, our work is database-agnostic. This way, it is not limited to SQL but can be applied on different key-value-like databases.

2. Building blocks

Next, we revisit basic security notions and special properties that make a cryptosystem suitable to a certain context.

2.1. Security notions

Ciphertext indistinguishability is an important property that can be used to analyze the security of a cryptosystem. Two scenarios are considered, when an adversary has and does not have access to an oracle that provides decryption capabilities. Usually these scenarios are evaluated through a game in which an adversary tries to acquire knowledge about ciphertexts generated by a challenger [Bellare et al. 1998].

Indistinguishability under chosen plaintext attack - IND-CPA. In the IND-CPA game the challenger generates a pair (PK,SK) of cryptographic keys, makes PK public and keeps SK secret. An adversary has as objective to recognize a ciphertext created from a randomly chosen message from a known two-element message set. A polynomially bounded number of operations is allowed, including encryption (but not decryption), over PK and the ciphertexts. A cryptosystem is indistinguishable under chosen plaintext attack if no adversary is able to achieve the objective with non-negligible probability.

Indistinguishability under chosen ciphertext attack/adaptive chosen ciphertext attack - IND-CCA and IND-CCA2 This type of indistinguishability differs from IND-CPA due

to the adversary having access to a decryption oracle. In this game the challenge is again to recognize a ciphertext as described before, but now the adversary is able to use decryption results. This new game has two versions, non-adaptive and adaptive. In the non-adaptive version, IND-CCA, the adversary may use the decryption oracle until it receives the challenge ciphertext. On the other hand, in the adaptive version he is allowed to use the decryption oracle even after that event. For obvious reasons, the adversary cannot send the challenge ciphertext to the decryption oracle. A cryptosystem is indistinguishable under chosen ciphertext attack/adaptive chosen ciphertext attack if no adversary is able to achieve the objective with non-negligible probability.

Indistinguishability under an ordered chosen plaintext attack - IND-OCPA Introduced by Boldyreva et al., this notion supposes that an adversary is capable of retrieving two sequences of ciphertexts resulting of the encryption of any two sequences of messages [Boldyreva et al. 2009]. Furthermore, he knows that both sequences have identical ordering. The objective of this adversary is to distinguish between these ciphertexts. A cryptosystem is indistinguishable under an ordered chosen plaintext attack if no adversary is able to achieve the objective with non-negligible probability.

2.2. Order-revealing encryption (ORE)

Order-revealing encryption schemes are characterized by having, in addition to the usual set of cryptographic functions like *keygen* and *encrypt*, a function capable of comparing ciphertexts and returning the order of the original plaintext, as shown by Definition 1.

Definition 1 (Order-revealing encryption) *Let E be an encryption function, C be a comparison function, and m_1 and m_2 be plaintexts from the message space. The pair (E, C) is defined as an encryption scheme with the order-revealing property if:*

$$C(E(m_1), E(m_2)) = \begin{cases} \text{LOWER}, & \text{if } m_1 < m_2, \\ \text{EQUAL}, & \text{if } m_1 = m_2, \\ \text{GREATER}, & \text{otherwise.} \end{cases}$$

This is a generalization of order-preserving encryption (OPE), that fixes C to a simple numerical comparison [Boneh et al. 2015].

Security As argued by Lewi and Wu, the "best-possible" notion of security for ORE is IND-OCPA, what means that it is possible to achieve indistinguishability of ciphertexts. This property implies a much stronger security guarantee than OPE schemes can have [Lewi and Wu 2016]. Furthermore, differently from OPE, ORE is not inherently deterministic [Kolesnikov and Shikfa 2012]. For example, Chenette et al. propose an ORE scheme that applies a pseudo-random function over an OPE scheme, while Lewi and Wu propose an ORE scheme completely built upon symmetric primitives, capable of limiting the use of the comparison function and reducing the leakage inherent to this routine [Chenette et al. 2016, Lewi and Wu 2016]. Nevertheless, any scheme that reveals numerical order of plaintexts through ciphertexts is vulnerable to inference attacks and frequency analysis, as those described by Naveed et al. over relational databases encrypted using deterministic and OPE schemes [Naveed et al. 2015]. Although ORE does not completely discard the possibility of such attacks, it offers stronger defenses.

2.3. Homomorphic encryption (HE)

Homomorphic encryption schemes have the property of conserving some plaintext structure during the encryption process, allowing the evaluation of certain functions over ciphertexts and obtaining, after decryption, a result equivalent to the same computation applied over plaintexts. Definition 2 presents this property in a more formal way.

Definition 2 (Homomorphic encryption) *Let E and D be a pair of encryption and decryption functions, and m_1 and m_2 be plaintexts. The pair (E, D) forms an encryption scheme with the homomorphic property for some operator $\diamond$ if and only if the following holds:*

$$E(m_1) \circ E(m_2) \equiv E(m_1 \diamond m_2).$$

The operation $\circ$ in the ciphertext domain is equivalent to $\diamond$ in the plaintext domain.

Homomorphic cryptosystems are classified according to the supported operations and their limitations. *Partially homomorphic encryption* schemes (PHE) hold on Definition 2 for either unlimited addition or multiplication operations, while *fully homomorphic encryption* schemes (FHE) support both addition and multiplication operations.

Security In terms of security, homomorphic encryption schemes achieve at most IND-CCA-1 [Bellare et al. 1998]. This is a natural consequence of the design requirements, since these cryptosystems allow any entity to manipulate ciphertexts. Most of current proposals, however, reach at most IND-CPA and stay secure against attackers without access to a decryption oracle [Loftus et al. 2012].

3. Searchable encryption

In this section, the problem of searching over encrypted data is formally defined. We present two state-of-the-art solutions to this problem, namely the CryptDB and Arx database systems.

3.1. The problem

Suppose a scenario where Alice stores a set of documents with an untrusted entity Bob. She would like to keep this data encrypted because, as defined, Bob is not trustworthy. Alice also would like to occasionally retrieve a subset of documents accordingly to a predicate without revealing any sensitive information to Bob. Thus, sharing the decryption key is not an option. The problem lies in the fact that communication between Alice and Bob may (and probably will) be constrained. Hence, a naive solution consisting of Bob sending all documents to Alice and letting her decrypt and select whatever she wants may not be feasible. Alice must then implement some mechanism to protect her encrypted data so that Bob will be able to identify the desired documents without knowing their contents or the selection criteria [Song et al. 2000].

An approach that Alice can take is to create an encrypted index as in Definition 3.

Definition 3 (Searching on an encrypted index) *Suppose a database $\mathcal{DB} = (m_1, \ldots, m_n)$ and a list $\mathcal{W} = (W_1, \ldots, W_m)$ of sets of keywords such that W_i contains keywords for m_i. The following routines are needed to build and search on an encrypted index:*

BUILDINDEX$_K(\mathcal{DB}, \mathcal{W})$**:** *The list $\mathcal{W}$ is encrypted using a searchable scheme under a key K and results in a searchable encrypted index $\mathcal{I}$. This process may not be reversible (e.g., if a hash function is used). The routine outputs $\mathcal{I}$.*

TRAPDOOR$_K(\mathcal{F})$**:** *This function receives a predicate $\mathcal{F}$ and outputs a trapdoor $\mathcal{T}$. The latter is defined as the information needed to search $\mathcal{I}$ and find records that satisfy $\mathcal{F}$.*

SEARCH$_{\mathcal{I}}(\mathcal{T})$**:** *It iterates through $\mathcal{I}$ applying T and outputs every record that returns* TRUE *for the input trapdoor.*

This way, Alice is able to keep her data stored with Bob and remain capable of selecting subsets of it without leaking information [Bösch et al. 2014].

3.2. CryptDB

CryptDB is a software layer that provides capabilities to store data in a remote database and query over it without revealing sensitive information to the database management system (DBMS). It introduces a proxy layer responsible to encrypt and adjust queries to the database and decrypt the outcome [Popa et al. 2011].

The context in which CryptDB stands is a typical structure of database-backed applications, consisting of a DBMS server and a separate application server. To query a database, a predicate is generated by the application and processed by the proxy before it is sent to the DBMS server. The user interacts exclusively with the application server and is responsible for keeping the password secret. This password is provided on login to the proxy (via application) that derives all the keys required to interact with the database. When the user logs out, it is expected that the proxy deletes their keys.

Data encryption is done through "onions". These are defined as layers of encryption that are combined to provide different functionalities. Modeling a database involves evaluating the meaning of each attribute and predicting the operations it must support. In particular, keyword-searching as described in Definition 3 is implemented as proposed in Song's work [Song et al. 2000].

The authors address two types of threats with CryptDB: curious database administrators who try to snoop and acquire information about client's data; and an adversary that gains complete control of application and DBMS servers. The first threat is achieved through the encryption of stored data and the ability to query it without any decryption or knowledge about its content. The second threat, as the authors claims, applies only to logged-out clients. In the described scenario, the cryptographic keys relative to data in the database are handled by the application server. Thus, if the application server is compromised, all the keys it possesses at that moment (that are expected to be only from logged-in users) are leaked to the attacker.

3.3. Arx

Arx is a database system alternative to CryptDB [Poddar et al. 2016]. It targets much stronger security properties and claims to protect the database with the same level of regular AES-based encryption[1], achieving IND-CPA security. This is a direct consequence of the

[1]The Advanced Encryption Standard (AES) is a well-established symmetric block cipher enabling high performance implementation in hardware and software [Daemen and Rijmen 1999].

almost exclusively use of AES to construct selection operators, even on range queries. This not only brings strong security but also good performance, due to efficiency of symmetric primitives, sometimes even benefiting from hardware implementations. The building blocks used for searching follow those described previously. Futhermore they apply a different AES key for each keyword when generating the trapdoor.

At its core, Arx introduces two database indexes, ARX-RANGE for range and order-by-limit queries and ARX-EQ for equality queries, both built on top of AES. The former uses an obfuscation strategy to protect the data it compares and does not leak any information, while enabling searches in logarithmic time. The latter embeds a counter into each repeating value. This ensures that the encryption of both are different, protecting it against frequency analysis. Using a small token provided by the client, the database is able to expand it in many search tokens and return all the occurrences desired, allowing an index to be built over encrypted data.

The context in which Arx stands is similar to CryptDB. However, the authors consider the data owner as the application itself. This way, it simplifies the security measures and considers the responsibility to keep the application server secure outside of its scope.

4. Proposed framework

The goal of the proposed framework is to develop a database model capable of storing encrypted records and applying relational algebra primitives on it without the knowledge of any cryptographic keys or the need for decryption. A trade-off between performance and security is desirable, however we completely discard deterministic encryption whenever is possible for security reasons. The applicability of this framework goes beyond SQL databases. Besides the relational algebra hereby used to describe the framework, it can be extended to key-value, document-oriented, full text and several other databases classes that keeps the same attribute structure.

The three main operations needed to build a useful database are insertion, selection and update. Once data is loaded, being able to select only those pieces that correspond to an arbitrary predicate is the fundamental block to construct more complex operations, as grouping and equality joins. This functionality is fundamental when there is a physical separation between the database and the data owner, otherwise high demand for bandwidth is incurred to transmit large fractions of the database records. Furthermore, real data is frequently mutable and thus the database must support updates to remain useful.

We define as *secure* a system model that guarantees that the data owner is the only entity capable of revealing data, which can be achieved by his exclusive possession of the cryptographic keys. Thus, a fundamental aspect of our proposal is the scenario in which the database and the application server handle data with minimum knowledge.

Lastly, the framework does not ensure integrity, freshness or completeness of results returned to the application or the user, since an adversary that compromises the database in some way can delete or alter records. We consider this threat to be outside the scope of this framework.

4.1. Classes of attributes

Records in an encrypted database are composed by attributes. These consist of a name and a value, that can be an integer, float, string or even a binary blob. Values of attributes are

classified according to their purpose:

static An immutable value only used for storage. It is not expected to be evaluated with any function, so there is no special requirement for the encryption.

index Used for building a single or multivalued searchable index. It should enable one to verify if an arbitrary term is contained in a set without the need to acquire knowledge of their content.

computable A mutable value. It supports the evaluation with arithmetic circuits and ensures obtaining, after decryption, a result equivalent to the same circuit applied over plaintexts.

The implementation of each attribute must satisfy the requirements without leaking any vital information beyond those related directly with the attribute objective (i.e.: order for *index* attributes). Since the name of an attribute reveals information, it may need to be protected as well. However, the acknowledgement of an attribute is done using its name, so even anonymous attributes must be traceable in a query. An option for anonymizing the attribute name is to treat it as an *index*.

The aforementioned cryptosystems are natural suggestions to be applied within these classes. Since *static* is a class for storage only and has no other requirements, any scheme with appropriate security levels and performance may be used, as AES. On the other hand, *index* and *computable* attributes are immediate applications of ORE and HE schemes. Particularly, the latter defines the HE scheme according to the required operations. Attributes that require only one operation can be implemented with a PHE scheme, which provides good performance; while those that requires addition and multiplication must use FHE and deal with the performance issues.

Definition 4 (Secure ORE) *Let E and C be, respectively, an encryption and a comparison function. The pair (E, C) forms an encryption scheme with the order-revealing property defined as "secure" if and only if it satisfies Definition 1; the encryption of a message m can be written as $E(m) = (c_L, c_R) = (E_L(m), E_R(m))$, where E_L and E_R are complementary encryption functions; and the comparison between two ciphertexts c_1 and c_2 is done by $C(c_{L1}, c_{R2})$. This way, C may be applied without the complete knowledge of the ciphertexts.*

In order to build a secure and efficient *index*, an ORE scheme that corresponds to Definition 4 should be used. We define the search framework as in Definition 5.

Definition 5 (Encrypted search framework) *Let S be a set of words, sk a secret key, and an ORE scheme (*ENC, CMP*) that satisfies Definition 4. The operations required for an encrypted search over S are defined as follows:*

BUILDINDEX$_{sk}(S)$ *Output the set*

$$S^* = \{c_R \mid (c_L, c_R) = \text{ENC}_{sk}(w), \forall w \in S\}.$$

TRAPDOOR$_{sk}(w)$ *Output the trapdoor*

$$T_w = (c_L \mid (c_L, c_R) = \text{ENC}_{sk}(w)).$$

SEARCH$_{S^*,\,r}(T_w)$ *To select all records in* S^* *with the relation* $r \in \{$LOWER, EQUAL, GREATER$\}$ *to a word* w*, one computes the trapdoor* T_w *and iterates through* S^* *looking for the records* $w^* \in S^*$ *that satisfy*

$$\textsc{Cmp}\,(T_w, w^*) = r.$$

The set $\hat{S}$ *with all the elements in* S^* *that satisfy this equation is returned.*

4.2. Database operations

A relational algebra for database operations can be built over six main operations to query on related sets of data: selection, projection, rename, Cartesian product, union and difference [Codd 1983]. They are defined as follows.

1. **Selection** (σ): The selection of entries in the database is done exclusively using *index* attributes.
   (a) *index*: The user wants all records with the relationship r when compared to an arbitrary word w in an *index* attribute, where $r \in \{$LOWER, EQUAL, GREATER$\}$. The trapdoor $T_w = Trapdoor_{sk}(w)$ is sent to the server that executes SEARCH for the desired r.
2. **Projection** (π): Attribute names may or may not be encrypted.
   (a) *encrypted*: If encrypted, a deterministic scheme is used or they are treated as *index* values.
      i. *deterministic scheme*: The user defines a set A of attribute names and computes $A^* = \{Enc(a) \mid a \in A\}$ using the encryption function related to the deterministic scheme chosen. A^* is sent to the server that returns only the records with names in it.
      ii. *index*: The user defines a set A of attribute names, computes $A^* = \{Trapdoor_{sk}(a) \mid a \in A\}$ and sends it to the server, that selects the projected attributes through the operation SEARCH.
   (b) *unencrypted*: Unencrypted attribute names may be sent to and selected by the server using a standard algorithm.
3. **Rename** (ρ): Attribute names may or may not be encrypted.
   (a) *encrypted*: If encrypted, a deterministic scheme is used or they are treated as *index* values.
      i. *deterministic scheme*: The user picks an attribute A and the new name B and computes $(A^*, B^*) = (Enc(A), Enc(B))$ using the encryption function related to the deterministic scheme chosen. (A^*, B^*) is sent to the server that applies a standard algorithm.
      ii. *index*: The user picks an attribute A and the new name B and computes $(A^*, B^*) = (Trapdoor_{sk}(A), (c_R \mid (c_L, c_R) = \text{Enc}_{sk}(B)))$. (A^*, B^*) is sent to the server, that selects attributes related to A^* as EQUAL through the operation SEARCH and renames the result to B^*.
   (b) *unencrypted*: Unencrypted attribute names may be renamed by the server using a standard algorithm.
4. **Cartesian product** ($\times$): The Cartesian product of two datasets encrypted with the same keys is executed using a standard algorithm.

5. **Difference** ($-$): The difference between two datasets A and B encrypted with the same keys is defined as $A - B = \sigma_{\text{not in B}}(A)$, that means a selection in A of all elements not contained in B.
6. **Union** ($\cup$): The union of two datasets encrypted with the same keys is defined as $A \cup B = A + (B - A)$, where $+$ is the usual set operator.

In addition, three other important operations must be defined to complete the set of operators:

7. **Insert**: Encrypted data is provided and added to the database using a standard algorithm.
8. **Intersect** ($\cap$): The intersection of two sets A and B encrypted with the same keys is defined as $A \cap B = \sigma_{\text{in B}}(A)$.
9. **Update**: An update operation is defined as a selection followed by the evaluation of a *computable* attribute by a supported homomorphic operation.

This set of operators enables operating over an encrypted database without the knowledge of cryptographic keys or acquiring sensitive information from user queries.

4.3. Security analysis

We assume the scenario in which the data owner has exclusive possession of cryptographic keys. This way, insertions to the database must be locally encrypted before being sent to the server. The database or the application never deals with plaintext data. Our framework thus has the advantage over CryptDB of preserving privacy even in the outcome of a compromised database or application server.

Despite being conceptual similar to OPE, ORE is able to address several security limitations of it. ORE does not necessarily generate ciphertexts that reveal their order by design, but allows someone to protect this information and only reveals it through specific functions. ORE is able to achieve the IND-OCPA security notion and adds randomization to ciphertexts. Those characteristics make it much safer against inference attacks [Naveed et al. 2015]. The proposal of Lewi and Wu goes even beyond that and is capable of limiting the use of the comparison function [Lewi and Wu 2016]. Their scheme generates a ciphertext that can be decomposed into left and right components such that a comparison between two ciphertexts requires only a left component of one ciphertext and the right component of the other. This way, the authors argue that robustness against such attacks is ensured since the database dump may only contain the right component, that is encrypted using semantically-secure encryption.

Nonetheless, an eavesdropper is capable of recognizing repeated queries by observing the outcome of a selection. This weakness may still be used for inference attacks, that can breach confidentiality from related attributes. This issue can get worse if the trapdoor is deterministic, when there is no other solution than implementing a key refreshment algorithm. Besides that, the knowledge of the numerical order between every pair of elements in a sequence may leak information depending on the application. This problem manifests itself in our proposal on the σ primitive if it uses a weak index structure, like a naive sequential index. A balanced-tree-based structure, on the other hand, obscures the numerical order of elements in different branches. This way, an attacker is capable of recovering the order of up to $O(\log n)$ database elements and somewhat infer about the others, in a database with n elements.

Finally, BUILDINDEX is not able to hide the quantity of records that share the same index. This way, one is able to make inferences about those by the number of records. There is also no built-in protection for the number of entries in the database. A workaround is to fix the size of each *static* attribute value and round the quantity of records in the database using padding. This approach increases secrecy but also the storage overhead.

4.4. Performance analysis

The application of ORE as the main approach to build a database index provides an extremely important contribution to selection queries. SEARCH does not require walking through all the records testing a trapdoor, but only a logarithmic-smaller subset of it when implemented over an optimal index structure, as an AVL tree or B-tree based structure [Sedgewick 1983]. This characteristic is highlighted on union, intersection and difference operations, that work by comparing and selecting elements in different groups. Moreover, current proposals in the state of the art of ORE enjoys the good performance provided by symmetric primitives and does not require more expensive approaches such as public-key cryptography [Chenette et al. 2016, Lewi and Wu 2016, Boneh et al. 2015]. In particular, although fully homomorphic cryptosystems promises to fulfill this task, it is still prohibitively expensive for real-world deployments [Boneh et al. 2013].

Space consumption is also affected. Ciphertexts are computed as a combination of the plaintext with random data. This way, a non-trivial expansion rate is expected. Differently from speed overheads which are affected by a single attribute type, all attributes suffer with the expansion rate of encryption.

4.5. Capabilities and limitations

Our framework is capable of providing an always-encrypted database that preserves secrecy as long as the data owner keeps the cryptographic keys secure. One is able to select records through *index* and apply arbitrary operations on attributes defined as *computable*. Furthermore, it increases the security of data but maintaining the computational complexity of standard relational primitives, achieving a fair trade-off between security and performance.

Although the framework has no constraints about attributes classified as both *index* and *computable*, there is no known encryption scheme in the literature capable of satisfying all the requirements. This way, the relational model of the database must be as precise as possible when assigning attributes to each class, specially because the costs of a model refactor can be prohibitive.

Some scenarios appear to be more compatible with an encrypted database as described than others. An e-mail service, for example, can be trivially adapted. The e-mails received by a user are stored in encrypted form as *static* and some heuristic is applied on its content to generate a set of keywords to be used on BUILDINDEX. This heuristic may use all unique words in the e-mail, for example. The sender address may be an important value for querying as well, so it may be stored as an *index*. To optimize common queries, a secondary collection of records may be instantiated with, for example, counters. The quantity of e-mails received from a particular sender, how often a term appears or how many messages are received in a time frame. Storing this metadata information in a secondary data collection avoids some of the high costs of searching in the main dataset.

However, our proposal fails when the user wants to search for something that was not previously expected. For example, regular expressions. Suppose a query that searches

for all the sentences that start with “Attack” and end with “dawn”, or all the e-mails on the domain “gmail.com”. If these patterns were not foreseen when the keyword index was built, then no one will be able to correctly execute this selection without the decryption of the entire database. Since the format of the strings is lost on encryption, this kind of search is impossible on our proposal.

5. Implementation

A proof-of-concept implementation of the proposed framework was developed aiming at the popular document-based database MongoDB [Chodorow and Dirolf 2010] and made available to the community under a GNU GPLv3 license [Alves 2016]. It is written in Python and works as a wrapper on its driver. This way, it becomes database-agnostic and limits the database server to dealing with encrypted data. Table 1 provides the schemes used for each attribute class, the parameter size and its security level.

Table 1. Chosen cryptosystems for each attribute presented in Section 4.

Attribute	Cryptosystem	Parameters	Security level
static	AES	128 bits	128 bits
index	Lewi-Wu	128 bits	128 bits
computable - add	Paillier	3072 bits	128 bits
computable - mul	ElGamal	3072 bits	128 bits

Lewi-Wu’s ORE scheme relies on symmetric primitives and achieves IND-OCPA. The authors claim that this is more secure than all existing OPE and ORE schemes which are practical [Lewi and Wu 2016]. Finally, Paillier and ElGamal are known public-key schemes. Both achieve IND-CPA and are based in the integer factorization and discrete logarithm problems, respectively. Paillier supports homomorphic addition, while ElGamal provides homomorphic multiplication. Both are defined as PHE schemes [Paillier 1999, ElGamal 1985]. The implementation of AES was provided by *pycrypto* toolkit [Litzenberger 2016], while Lewi-Wu, Paillier and ElGamal were implemented with our own code.

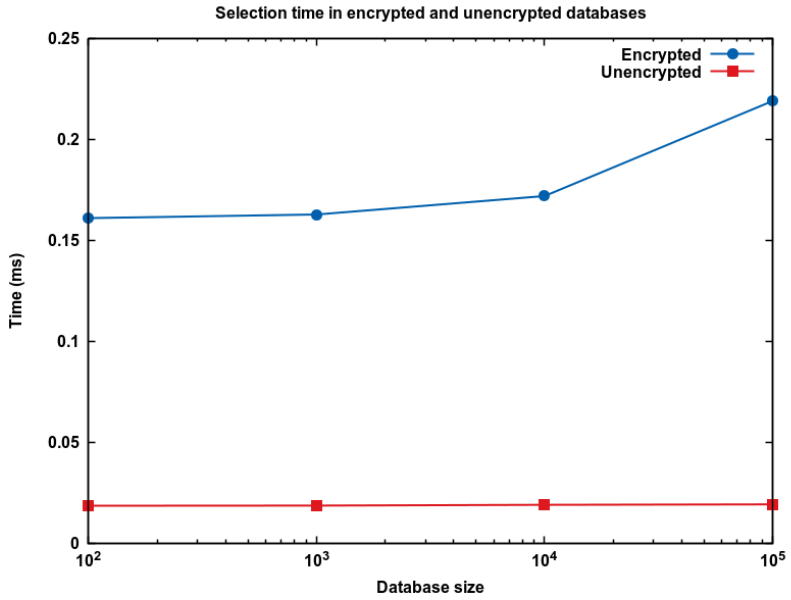
Table 2. Attribute structure of elements in the synthetic dataset.

Name	Value type	Class
e-mail	string	*static*
firstname	string	*static*
surname	string	*static*
country	string	*static*
age	integer	*index*
text	string	*static*

To measure the computational costs of managing an encrypted database, four synthetic variable-sized datasets were generated with the structure described in Table 2. Each one was loaded in encrypted and unencrypted form to a stock MongoDB database using insertion element-by-element by a Python script. An AVL tree was used as index for the documents through the attribute “age”. While it was possible to index the unencrypted database natively, it was not so simple with the encrypted version. MongoDB is not friendly to custom index structures or comparators, so we decided to construct the structure with Python

code and then insert it into the database using pointers based on MongoDB's native identity codes. This way, walking through the index tree depends on a database-external operation at Python-side, calling MongoDB's FIND method to localize documents related to left/right pointers starting from the tree root. Encrypting this document structure took 2.13ms in a Intel Xeon E5-2630 CPU at 2.60GHz. As can be seen in Figure 1, the performance overhead on queries in the encrypted database goes from 7 to 11 times.

Figure 1. Time required to perform a selection query on encrypted and unencrypted databases in the worst case scenario for an AVL tree index. The measures are the average of 100 independent executions.



Besides the need to walk through the AVL tree using database-external operations, these results are comparable to those related to Arx [Poddar et al. 2016] and one magnitude higher than CryptDB [Popa et al. 2011]. This way, it is expected that an efficient implementation, capable of executing searches completely inside MongoDB, will present an expressive speedup compared with the state of the art.

6. Conclusion

We presented the problem of searching in encrypted data and a proposal of a framework that guides the modeling of a database with support to this functionality. This is achieved by combining different cryptographic concepts and using different cryptosystems to satisfy the requirements of each attribute, like order-revealing encryption and homomorphic encryption. Over this approach, six main relational algebra operations were redefined to support encrypted data: selection, projection, rename, Cartesian product, union and difference. An overview of the security provided is discussed, as well as an analysis about the impact in database performance. We present a proof-of-concept implementation in Python over the document-based database MongoDB. A selection query on the worst case scenario was up to 11 times slower on the encrypted database. In comparison with CryptDB our proposal provides higher security, since it delegates exclusively to the data owner the responsibility of encrypting and decrypting data. This way, privacy holds even in a scenario of database or application compromised.

As future work, we intend to focus on performance and pursuit an efficient implementation of this framework.

References

[Alves 2016] Alves, P. (2016). A proof-of-concept searchable encryption backend for mongodb. `https://github.com/pdroalves/encrypted-mongodb`. Last accessed: 09/08/2016.

[BBC News 2016] BBC News (2016). Yahoo 'state' hackers stole data from 500 million users. Last accessed: 23/09/2016.

[Bellare et al. 1998] Bellare, M., Desai, A., Pointcheval, D., and Rogaway, P. (1998). *Advances in Cryptology — CRYPTO '98: 18th Annual International Cryptology Conference Santa Barbara, California, USA August 23–27, 1998 Proceedings*, chapter Relations among notions of security for public-key encryption schemes, pages 26–45. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Boldyreva et al. 2009] Boldyreva, A., Chenette, N., Lee, Y., and O'Neill, A. (2009). Order-preserving symmetric encryption. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5479 LNCS:224–241.

[Boneh et al. 2013] Boneh, D., Gentry, C., Halevi, S., Wang, F., and Wu, D. J. (2013). Private database queries using somewhat homomorphic encryption. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, ACNS'13, pages 102–118, Berlin, Heidelberg. Springer-Verlag.

[Boneh et al. 2015] Boneh, D., Lewi, K., Raykova, M., Sahai, A., Zhandry, M., and Zimmerman, J. (2015). *Semantically Secure Order-Revealing Encryption: Multi-input Functional Encryption Without Obfuscation*, pages 563–594. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Bösch et al. 2014] Bösch, C., Hartel, P., Jonker, W., and Peter, A. (2014). A survey of provably secure searchable encryption. *ACM Comput. Surv.*, 47(2):18:1–18:51.

[Chenette et al. 2016] Chenette, N., Lewi, K., Weis, S. A., and Wu, D. J. (2016). Practical order-revealing encryption with limited leakage. In FSE.

[Chodorow and Dirolf 2010] Chodorow, K. and Dirolf, M. (2010). *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 1st edition.

[Codd 1983] Codd, E. F. (1983). A relational model of data for large shared data banks. *Commun. ACM*, 26(6):64–69.

[Daemen and Rijmen 1999] Daemen, J. and Rijmen, V. (1999). AES Proposal: Rijndael.

[Dinh et al. 2013] Dinh, H. T., Lee, C., Niyato, D., and Wang, P. (2013). A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless Communications and Mobile Computing*, 13(18):1587–1611.

[ElGamal 1985] ElGamal, T. (1985). A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In Blakley, G. and Chaum, D., editors, *Advances in Cryptology*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer Berlin Heidelberg.

[Greenwald and MacAskill 2013] Greenwald, G. and MacAskill, E. (2013). NSA Prism program taps in to user data of Apple, Google and others.

[Hoffa et al. 2008] Hoffa, C., Mehta, G., Freeman, T., Deelman, E., Keahey, K., Berriman, B., and Good, J. (2008). On the Use of Cloud Computing for Scientific Workflows. In *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pages 640–645.

[Kolesnikov and Shikfa 2012] Kolesnikov, V. and Shikfa, A. (2012). On the limits of privacy provided by Order-Preserving Encryption. *Bell Labs Technical Journal*.

[Lewi and Wu 2016] Lewi, K. and Wu, D. J. (2016). Order-revealing encryption: New constructions, applications, and lower bounds. Cryptology ePrint Archive, Report 2016/612.

[Litzenberger 2016] Litzenberger, D. (2016). Python Cryptography Toolkit. `http://www.pycrypto.org/`. Last accessed: 07/03/2016.

[Loftus et al. 2012] Loftus, J., May, A., Smart, N. P., and Vercauteren, F. (2012). On CCA-Secure Somewhat Homomorphic Encryption. In *Proceedings of the 18th International Conference on Selected Areas in Cryptography*, SAC'11, pages 55–72, Berlin, Heidelberg. Springer-Verlag.

[Miller 2014] Miller, C. C. (2014). Revelations of N.S.A. spying cost U.S. tech companies. *The New York Times*. Last accessed: 02/04/2016.

[Naveed et al. 2015] Naveed, M., Kamara, S., and Wright, C. V. (2015). Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 644–655, New York, NY, USA. ACM.

[Paillier 1999] Paillier, P. (1999). Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In Stern, J., editor, *Advances in Cryptology — EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer Berlin Heidelberg.

[Poddar et al. 2016] Poddar, R., Boelter, T., and Popa, R. A. (2016). Arx: A strongly encrypted database system. Cryptology ePrint Archive, Report 2016/591.

[Popa et al. 2011] Popa, R. A., Redfield, C. M. S., Zeldovich, N., and Balakrishnan, H. (2011). Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100, New York, NY, USA. ACM.

[Sedgewick 1983] Sedgewick, R. (1983). *Algorithms*, chapter 15, page 199. Addison-Wesley.

[Song et al. 2000] Song, D. X., Wagner, D., Perrig, A., and Perrig, A. (2000). Practical techniques for searches on encrypted data. *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*, pages 44–55.

[Thomsen 2015] Thomsen, S. (2015). Extramarital affair website Ashley Madison has been hacked and attackers are threatening to leak data online. Last accessed: 25/05/2016.

[Weber 2014] Weber, H. (2014). How the NSA & FBI made Facebook the perfect mass surveillance tool. Venture Beat. Published on 05/15/2014.

[Xiao and Xiao 2013] Xiao, Z. and Xiao, Y. (2013). Security and Privacy in Cloud Computing. *IEEE Communications Surveys Tutorials*, 15(2):843–859.

