Alves and Aranha *Journal of Internet Services and Applications* (2018) 9:1
DOI 10.1186/s13174-017-0073-0

Journal of Internet Services and Applications

RESEARCH Open Access

A framework for searching encrypted databases

Pedro G. M. R. Alves* and Diego F. Aranha

Abstract

Cloud computing is a ubiquitous paradigm 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 computing 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 perform 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 the MongoDB system is offered and applied in the implementation of some of the main predicates required by the winning solution to Netflix Grand Prize.

Keywords: Cryptography, Functional encryption, Homomorphic encryption, Order revealing encryption, Searchable encryption, Databases

1 Introduction

The massive adoption of cloud computing is 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 service highly attractive [1, 2]. From mobile to scientific computing, the industry increasingly embraces cloud services and takes advantage of their potential to improve availability and reduce operational costs [3, 4]. However, the cloud cannot be blindly trusted. Malicious parties may acquire full access to the servers and consequently to data. Among the threats there are external entities exploiting vulnerabilities, intrusive governments requesting information, competitors seeking unfair advantages, and even possibly malicious system administrators. The data owner has no real control over the processing hardware and therefore cannot guarantee the secrecy of data [5]. The risk of confidentiality breaches caused by inadequate and insecure use of cloud computing is real and tangible.

The importance of privacy preservation is frequently underestimated, as well as the damage its failure represents to society, as the unfolding of a privacy breach may be completely unpredictable. A report from Javelin Advisory Services found a distressing correlation between individuals who were victims of data breaches and later victims of financial fraud. About 75% of total fraud losses in 2016 had this characteristic, corresponding to U$ 12 billion [6]. This could be avoided with the use of strong encryption at the user side, never revealing data even to the application or the cloud.

The problem of using standard encryption in an entire database is that it eliminates the capability of selecting records or evaluating arbitrary functions without the cryptographic keys, reducing the cloud to a complex and huge storage service. For this reason, alternatives have been proposed to solve this problem, starting from anonymization and heuristic operational measures which do not provide formal privacy guarantees. Encryption schemes tailored for databases such as searchable encryption are a promising solution with perhaps more

*Correspondence: pedro.alves@ic.unicamp.br
This is the extended version of a paper by the same name that appeared in XVI Brazilian Symposium on Information and Computational Systems Security in November, 2016.
Institute of Computing, University of Campinas, Albert Einstein Ave. 1251, 13083-852 Campinas/SP, Brazil

clear benefits [7–10]. 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.

1.1 The frustration of data anonymization

In 2006, Netflix shared their interest in improving the recommendation system offered to their users with the academic community. This synergy was directed to an open competition during 3 rounds which offered financial prizes for the best recommendation algorithms. An important feature of Netflix's commercial model is to efficiently and assertively guide subscribers in finding content compatible to their interests. Doing this correctly may reinforce the importance of the product for leisure activities, consolidate Netflix's commercial position, and ensure clients' loyalty [11].

The participants of the contest received a training set with anonymized movie ratings collected from Netflix subscribers between 1999 and 2005. There are approximately half million customers and about 17 thousands movies classified in the set, totalling over 100 million ratings. This dataset is composed by movie titles, the timestamp when the rating was created, the rating itself, and an identification number for relating same-user records. No other information about customers was shared, such as name, address or gender. The objective of the participants was to predict with good accuracy how much someone would enjoy a movie based on their previously observed behavior in the platform.

In the same year, America Online (AOL) took a similar approach and released millions of search queries made by 658,000 of its users with the goal of contributing to the scientific community by enabling statistical work over real data [12]. As Netflix, AOL applied efforts on anonymizing the data before publishing. All the obviously sensitive data, such as usernames and IP addresses, were suppressed, being replaced by unique identification numbers.

The ability to understand user's interests and predict their behavior based on collected data is desirable in several commercial models and consequently a hot topic in the scientific literature [13–15]. However, the importance of privacy-preserving practices is still underestimated, a challenge to overcome. For instance, despite the anonymization efforts of Netflix, Narayanan and Shmatikov brilliantly demonstrated how to break anonymity of the Netflix's dataset by cross-referencing information with public knowledge bases, as those provided by the Internet Movie Database (IMDB) [16]. Using a similar approach, New York Times' reporters were capable of relating a subset of queries to a particular person by joining apparently innocent queries to non-anonymous real state public databases [17].

1.2 "Unexpected" leaks

These events raised a still unsolved discussion about how to safely collect and use data without undermining user privacy. As remarked by Narayanan and Felten, "data privacy is a hard problem" [18]. Even when data holders choose the most conservative practice and never share data, system breaches may happen.

In 2013, a large-scale surveillance program of the USA government was revealed by Edward Snowden, a former NSA employee. Named PRISM, it was structured as a massive data interception effort to collect information for posterior analysis. Their techniques arguably had support of the US legal system and were frequently applicable even without knowledge of the data-owner companies [19, 20].

Two years later, in 2015, stolen personal data of millions of users of the website Ashley Madison was leaked by malicious parties exploiting security vulnerabilities [21]. As consequence, several reports of extortion and even a suicide, illustrating how increasingly sensitive data breaches are becoming.

In the same year, the Sweden's Transport Agency decided to outsource its IT operations to IBM. To fulfill the contract, the latter chose sites in Eastern Europe to place these operations. This resulted in Swedish confidential data being stored in foreign data centers, in particular Czech Republic, Serbia and Romania. As expected, this decision led to a massive data leak, containing information about all vehicles throughout Sweden, including police and military vehicles. Thus, names, photos and home addresses of millions of Swedish citizens, military personal, people under the witness relocation program, were exposed [22].

In 2016, Yahoo confirmed that a massive data breach, possibly the largest known, affected about 500 million accounts and revealed to the world a dataset full of names, addresses, and telephone numbers [23].

These occurrences take us to the disturbing feeling that, despise all efforts, the risk of data deanonymization increases in worrying ways following how much of it is made available [24, 25]. Hence, a seemingly obvious strategy to avoid such issue is to simply stop any kind of dataset collection.

1.3 Privacy by renouncing knowledge

History has proven that the task of collecting and storing data from third parties should be treated as risky. The chance of compromising user privacy by accident is too big and possibly with extreme consequences. This way, the concept of security by renouncing knowledge has attracted adepts, as the search engine DuckDuckGo that states in a blog post that "the only truly anonymised data is no data", and because of that claims to forego the right to store their users' data [26, 27].

A more financial-realistic approach for dealing with this issue is not to give up completely of knowledge but reduce the entities with access by keeping it encrypted during all its lifespan: transportation, storage, and processing, staying secret to the application and the cloud. Thus, a new security fence is set, tying data secrecy to formal guarantees.

1.4 Our contributions

This work follows the state of the art and proposes directives to the modeling of a searchable encrypted database [28]. We detect 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 proposed selection primitive achieves time complexity of $\Theta(\log n)$ on the dataset size. 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. The source code was made available to the community under a GNU GPLv3 license [29].

When compared to CryptDB [7], our proposal provides stronger security since it is able to keep 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. Furthermore, our work is database-agnostic, it is not limited to SQL but can be applied on different key-value databases.

This work is structured as follows. Section 2 describes the cryptographic building blocks required for building our proposed solution. Sections 3 and 4 define searchable encryption, discuss related threats, and present existing implementations. Section 5 proposes our framework, while Section 6 discusses its suitability in a recommendation system for Netflix. Our experimental validation results are presented in Sections 7 and 8 concludes the paper.

2 Building blocks

The two main classes of cryptosystems are known as symmetric and asymmetric (or public-key) and defined by how users exchange cryptographic keys. Symmetric schemes use the same secret key for encryption and decryption, or equivalently can efficiently compute one from the other, while asymmetric schemes generate a pair of keys composed by public and private keys. The former is distributed openly and is the sole information needed to encrypt a message to the key owner, while the latter should be kept secret and used for decryption.

Besides this, cryptosystems that produce always the same ciphertext for the same message-key input pair are known as deterministic. The opposite, when randomness is used during encryption, are known as probabilistic. We next recall basic security notions and special properties that make a cryptosystem suitable to a certain application. Later, we shall make use of these concepts to analyze the security of our proposal.

2.1 Security notions

Ciphertext indistinguishability is a useful property 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 are evaluated through a game in which an adversary tries to acquire information from ciphertexts generated by a challenger [30].

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 and adaptive chosen ciphertext attack – IND-CCA1 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-CCA1, 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 chosen keyword attack and adaptative chosen keyword attack – IND-CKA and IND-CKA2 This security notion is specific to the context of keyword-based searchable encryption [31]. It considers a scenario in which a challenger builds an index with keyword sets from some documents. This index enables someone to use a value $\mathcal{T}_w$, called trapdoor, to verify if a document contains the word w. This game imposes that no information should be leaked from the remotely stored files or index beyond the outcome and the search pattern of the queries. The adversary has access to an oracle that provides the related trapdoor for any word. His objective is to use this oracle as training to apply the acquired knowledge and break the secrecy of unknown encrypted keywords. As well as in the IND-CCA1/IND-CCA2 game, the non-adaptative version, IND-CKA, of this game forbids the adversary to use the trapdoor oracle once the challenge trapdoor is sent by the challenger. On the other hand, the adaptative version allows the use of the trapdoor oracle even after this event.

A cryptosystem is indistinguishable under chosen keyword attack if every adversary has only a negligible advantage over random guessing.

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 [32]. 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 Functional encryption

Cryptographic schemes deemed "functional" receive such name because they support one or more operations over the produced ciphertexts, hence becoming useful not only for secure storage.

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 plaintexts, as shown by Definition 1.

Definition 1 (ORE) *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:*

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

This is a generalization of order-preserving encryption (OPE), that fixes C to a simple numerical comparison [33].

Security As argued by Lewi and Wu, the "best-possible" notion of security for ORE is IND-OCPA, which means that it is possible to achieve indistinguishability of ciphertexts and with a much stronger security guarantee than OPE schemes can have [34]. Furthermore, differently from OPE, ORE is not inherently deterministic [35]. 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 [34, 36]. Their solution works by defining ciphertexts composed by pairs (ct_L, ct_R). To compare ciphertexts ct_A and ct_B, it requires ct_{A_L} and ct_{B_R}. This way, the data owner is capable of storing only one side of those pairs in a remote database being certain that no one will be able to make comparisons between those elements. 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 [37]. Although ORE does not completely discard the possibility of such attacks, it offers stronger defenses.

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 (HE) *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 addition or multiplication operations, while *fully homomorphic encryption* schemes (FHE) support both addition and multiplication operations.

PHE cryptosystems have been known for decades [38, 39]. However, the most common data processing applications, as those arising from statistics, machine learning or genomics processing, frequently require support for both addition and multiplication operations simultaneously. This way, such schemes are not suitable for general computation.

Nowadays, FHE performance is prohibitive, so weaker variants, such as SHE[1] and LHE[2], have the stage for solving computational problems of moderate complexity [40, 41].

Security In terms of security, homomorphic encryption schemes achieve at most IND-CCA1, which means that the scheme is not secure against an attacker with arbitrary access to a decryption oracle [30]. 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 [42].

3 Searchable encryption

We now formally define the problem of searching over encrypted data. We present three state-of-the-art implementations of solutions to this problem, namely the CryptDB, Arx, and Seabed database systems.

3.1 The problem

Suppose a scenario where Alice keeps a set of documents in untrusted storage maintained by an also untrusted entity Bob. She would like to keep this data encrypted because, as defined, Bob cannot be trusted. 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 [43].

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

Definition 3 (Encrypted indexing) *Suppose a dataset* $\mathcal{DB} = (m_1, \ldots, m_n)$ *and a list* $\mathcal{W} = (W_1, \ldots, W_n)$ *of sets of keywords such that* W_i *contains keywords for* m_i. *The following routines are required 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 the trapdoor* $\mathcal{T}$ *and outputs every record that returns* TRUE *for the input trapdoor.*

This way, if the searchable cryptosystem used is IND-CKA then Alice is able to keep her data with Bob and remain capable of selecting subsets of it without revealing information [28].

3.2 Threat modeling

The development of efficient and secure solutions for management of datasets depends on the awareness of the threats we intent to mitigate. For such, this work follows Grubbs' definitions of adversaries for a database [44].

Active attacker The worst case scenario is when the attacker acquires full control over the server, being capable of performing arbitrary operations. Thus, he is not committed to follow any protocol.

Snapshot attacker The adversary obtains a snapshot of the dataset containing the primary data and indexes but no information about issued queries and how they access the encrypted data.

Persistent passive attacker Another possibility is a scenario in which the attack cannot interfere with the server functionality but can observe all of its operations. We do not consider only attackers that inspect issued queries in real-time, but also those that are able to recover them later. As demonstrated by Grubbs, the data contained in a real-world database goes far beyond the primary dataset (names, addresses, ...). It also includes logs, caches, and auxiliary tables (as MySQL's diagnostic tables) used, for instance, to guarantee *ACID*[3] and enable the server to undo incomplete queries after a power-break. It is very likely that an attacker competent to subjugate the security protocols of the system will be capable to also recover these secondary datasets.

The idea of a *snapshot attacker* is very popular among solutions and researchers intended to develop encrypted databases. Nevertheless, it underestimates the attacker and the many side-attacks a motivated adversary can execute. As Rogaway remarks, we cannot make the mistake to reduce the adversary to the lazy and abstract Bob, but we must remember that it can go far beyond that and take the form of a military-industrial-surveillance program with a billionaire budget and capability to surpass the obvious [45].

4 Related work

The management of a dataset is made by a database management system (DBMS). It is composed by several layers responsible for coordinating read and write operations, guarantee data consistency and integrity, and user access. The engineering of such a system is a complex task and requires smart optimizations to be able to store data, process queries and return the outcome with minimum latency and good scalability.

This way, searchable encryption solutions usually are implemented not inside but on top of these systems as a middleware to translate encrypted queries to the DBMS without revealing plaintext data and decrypt the outcome, as shown in Fig. 1. This strategy enables the use of decades of optimizations incorporated in nowadays DBMSs and portable to encrypted data. It is important to state that, ideally, security features should be designed in conjunction with the underlying database. Long-term solutions are expected to assimilate those strategies internally in the DBMS core.

4.1 CryptDB

CryptDB is a software layer that provides capabilities to store data in a remote SQL database and query over it without revealing sensitive information to the DBMS. It introduces a proxy layer responsible to encrypt and adjust queries to the database and decrypt the outcome [7].

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 his password secret. This is provided on login to the proxy (via application) that derives all the cryptographic keys required to interact with the database. When the user logs out, it is expected that the proxy deletes its keys.

Data encryption is done through the concept of "onions", which consist of layers of encryption that are combined to provide different functionalities, as shown in Fig. 2. Such layers are revealed as necessary to process the queries being performed. 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 [43]. The performance overhead over MySQL measured by the authors is up to 30%.

Two types of threats are treated in CryptDB: curious database administrators who try to snoop and acquire information about client's data but respect the established protocols (a *persistent passive attacker*); and an adversary that gains complete control of application and DBMS servers (an *active attacker*). The authors state that the first threat is mitigated through the encryption of stored data and the ability to query it without

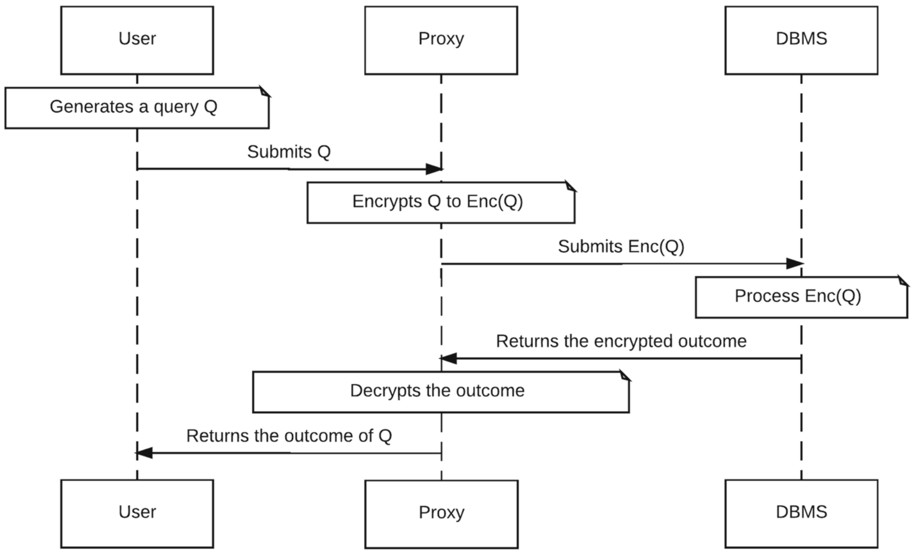

Fig. 1 Sequence diagram representing the process of generating and processing an encrypted query. The proxy is positioned between the user and the DBMS in a trusted environment. Its responsibility is to receive a plaintext query, apply an encryption protocol, submit the encrypted query to the DBMS, and decrypt the outcome

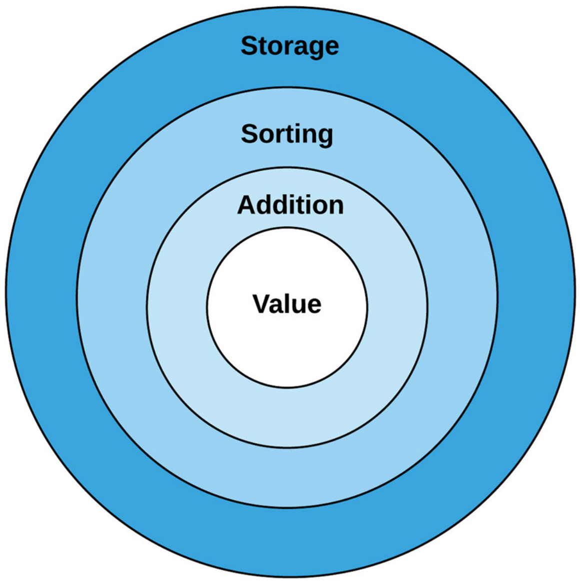

Fig. 2 Representation of the data format used by CryptDB. The current value to be protected lies in the center, and a new encryption layer is overlapped to it according to the need of a particular functionality

any decryption or knowledge about its content; while the second applies only to logged-in clients. In the considered 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. Such arguments were revisited after works by Naveed and Grubbs et al. demonstrated how to explore several weaknesses of the construction, such as the application of OPE [46, 47].

4.2 Arx

Arx is a database system implemented on top of MongoDB [8]. It targets much stronger security properties and claims to protect the database with the same level of regular AES-based encryption[4], achieving IND-CPA security. This is a direct consequence of the almost exclusively use of AES to construct selection operators, even on range queries, and not only brings strong security but also good performance due to efficiency of symmetric primitives, sometimes even benefiting from hardware implementations. The authors report a performance overhead of approximately 10% when used to replace the database of ShareLatex. The building blocks used for searching follow those described in Definition 3. Furthermore, they apply a different AES key for each keyword when generating the trapdoor, requiring the client to store counters, as explained in the next paragraph.

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 and using chained garbled circuits. The former uses an obfuscation strategy to protect data, 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 them against frequency analysis. Using a 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.

4.3 Seabed

Seabed was developed by Papadimitriou et al. and aims at Business Intelligence (BI) applications interested in keeping data secure on the cloud [48]. As well as CryptDB and Arx, Seabed was built consisting of a client-side query translator (to SQL), a query planner, and a proxy that connects to a Apache Spark instance [49]. Its main foundations are two new cryptographic constructions, *additively symmetric homomorphic encryption* (ASHE) and *Splayed ASHE* (SPLASHE). The former is used to replace Paillier as the additively homomorphic encryption scheme, stating that their construction is up to three orders of magnitude faster. The latter is used to protect the database against inference attacks [37].

SPLASHE works by splitting sensitive data into multiple attributes, obscuring the low-entropy of deterministic encryption. Formally, let C be a sensitive attribute of a dataset that can be filled with d possible discrete values. The approach taken by SPLASHE is to replace this attribute in the encrypted database by $\{C_1, C_2, \cdots, C_d\}$ such that $C_v = 1$ and $C_t = 0$ for $t \neq v$ if $C = v$. When encrypted by ASHE the ciphertexts will look random to the adversary.

Seabed's authors argue that SPLASHE is strong enough to mitigate frequency analysis, enabling the use of deterministic encryption whenever it is required in the database model. However, Grubbs states that SPLASHE's protection may be deflected through the auxiliary data stored by the database [44]. Their work demonstrates how state-of-the-art databases store metadata that can be used to reconstruct issued queries and, this way, recognize access patterns on the attributes. Such patterns leak the information that SPLASHE intended to hide. Considering this, the only threat really mitigated by SPLASHE against the deterministic encryption of Seabed is from a *snapshot attacker*.

5 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 possible for security reasons. The only exception are contexts with unique records, which avoid by definition weaknesses intrinsic to deterministic encryption. 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 keep 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 basic block to construct more complex operations, such 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.

5.1 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 its 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 its 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 (e.g.: 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, which has no other requirements, any scheme with appropriate security level 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 require arbitrary additions and multiplications 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$_{sk}(S)$*: Outputs the set*

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

Trapdoor$_{sk}(w)$*: Outputs the trapdoor*

$$\mathcal{T}_w = (c_L \mid (c_L, c_R) = \textsc{Enc}_{sk}(w)).$$

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

$$\mathrm{CMP}\left(\mathcal{T}_w, w^*\right) = r.$$

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

5.2 Database operations

Let us consider a model composed by an encrypted dataset stored in a remote server and a user that possesses the secret cryptographic keys. The latter would like to perform queries on data without revealing sensitive information to the server, as defined in Section 3.1.

In 1970, Codd proposed the use of a relational algebra as a model for SQL [50]. This consists of a small set of operators that can be combined to execute complex queries over the data.

Through the functions defined in Definition 5, a relational algebra for encrypted database operations can be built. The basic operators for such algebra are defined as follows.

1. **Projection** (π_A): Returns a subset A of attributes from selected records. This subset may be defined by attribute names that may or may not be encrypted.
   (a) *encrypted*: If encrypted, a deterministic scheme is used or they are treated as *index* values.
   deterministic scheme: The user computes $A^* = \{Enc_{Det}(a) | a \in A\}$. A^* is sent to the server, which picks the projected attributes using a standard algorithm.
   index attributes: The user computes $A^* = \left\{Trapdoor_{sk}(a) | a \in A\right\}$. A^* is sent to the server, which picks the projected attributes using SEARCH.
   (b) *unencrypted*: Unencrypted selectors are sent to and selected by the server using a standard algorithm.
2. **Selection** (σ_φ): Given a predicate φ, returns only the records satisfying it.
   - Handles exclusively *index*, hence φ must be equivalent to a combination of comparative operators supported by SEARCH.
   - Let $w \diamond x \leftarrow \varphi$, where $\diamond$ is a compatible comparative operator, w an *index* attribute, and x the operand to be compared (e.g.: $\sigma_{age>30}$ signals for records which the attribute named "age" value is greater than 30). The trapdoor $\mathcal{T}_\varphi = Trapdoor_{sk}(\varphi)$ is sent to the server that executes SEARCH.
3. **Cartesian product** ($\times$): The Cartesian product of two datasets is executed using a standard algorithm.
4. **Difference** ($-$): The difference between two datasets A and B encrypted with the same keys is defined as $A - B = \{x \mid x \in A \text{ and } x \notin B\}$.
5. **Union** ($\cup$): The union of two datasets A and B encrypted with the same keys is defined as $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$.

Union and difference are defined over datasets with the same set of attributes. The opposite is expected for Cartesian product, so that no attribute may be shared between operands.

Ramakrishnan calls these "basic operators" in the sense that they are essential and sufficient to execute relational operations [51]. Additional useful operators can be built over those. For instance: rename, join-like, and division. The same observation applies in the encrypted domain, and complex operators can be constructed given basic operators defined over the encrypted domain.

6. **Rename** ($\rho_{a,b}$): Renames attributes. Their names may or may not be encrypted.
   (a) *encrypted*: Encryption shall be executed using a deterministic cryptosystem or names treated as *index* values.

   deterministic scheme: Let a be an attribute name to be replaced by b. The user computes $a^* \leftarrow Enc_{Det}(a)$ and $b^* \leftarrow Enc_{Det}(b)$, and sends the output to the server, which applies a standard replacement algorithm.

   index attributes: Let a be an attribute name to be replaced by b. The user computes $a^* \leftarrow Trapdoor(a)$ and $b^* \leftarrow c_R \mid (c_L, c_R) = \mathrm{Enc}_{index}(b)$ and sends the output to to the server, which 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.
7. **Natural join** ($\bowtie$): Let A and B be datasets with a common subset of attributes. The natural join between A and B is defined as the selection of all elements that lies in A and B and match all the values in those attributes. More formally, let $c_1, c_2, \ldots, c_n$ be attributes common to A and B; $x_1, x_2, \ldots, x_n$ attributes not contained in A or in B; $a_1, a_2, \ldots, a_m$

be attributes unique to A; $b_1, b_2, \ldots, b_k$ be attributes unique to B; and $\mathbb{K} = \mathbb{N}^*_{n+1}$. We have that,

$$A \bowtie B \equiv \sigma_{c_i = x_i}\left(\rho_{(c_i,x_i)}(A) \times B\right), \forall i \in \mathbb{K}.$$

8. **Equi-join** ($\bowtie_\theta$): Let A and B be datasets. The equi-join between A and B is defined as the selection of all elements that lie in A and B and satisfy a condition θ. More formally, $A \bowtie B = \sigma_\theta(A \times B)$.
9. **Division** ($/$): Let A and B be datasets and C the subset of attributes unique to A. The division operator joins the operands by common attributes but projects only those unique to the dividend. Hence, $A/B = \pi_C\,(A \bowtie B)$.

Finally, it is important to define data insertion and update despite these cannot be properly defined as relational operators.

- **Insert**: Encrypted data is provided and inserted into the database using a standard algorithm.
- **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.

5.3 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 deal 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 conceptually similar to OPE, ORE is able to address several of its security limitations. ORE does not necessarily generate ciphertexts that reveal their order by design, but allows someone to protect this information by only revealing 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 [37]. The proposal of Lewi and Wu goes even beyond that and is capable of limiting the use of the comparison function [34]. 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. Their scheme satisfies our notion of a *Secure ORE* and, therefore, provides strong defenses against *Snapshot attackers*.

An eavesdropper (*Active* or *Persistent passive attacker*) is not capable of executing comparisons by himself in a *Secure ORE*. However he may learn the result of these and recognize 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 infer about the others, in a database with n elements.

Schemes used with *computable* attributes are limited to IND-CCA1, and typically reach only IND-CPA. Moreover, homomorphic ciphertexts are malleable by design. Thus, an attacker that acquires knowledge about a ciphertext can use it to predictably manipulate others.

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.

5.4 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 subset of it when implemented over an optimal index structure, such as an AVL tree or B-tree based structure [52]. This characteristic is highlighted on union, intersection and difference operations, which work by comparing and selecting elements in different groups. Moreover, current proposals in the state of the art of ORE enjoy good performance provided by symmetric primitives and does not require more expensive approaches such as public-key cryptography [33, 34, 36]. In particular, although fully homomorphic cryptosystems promise to fulfill this task and progress has been made with new cryptographic constructions [53], it is still prohibitively expensive for real-world deployments [54].

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.

5.5 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 "mail.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 in our proposal.

Lastly, relational integrity is a desired property for a relational database. It connects two or more sets using same-value attributes in both sets (e.g.: every value of a column in a table *A* exists in a column in table *B*), and establishes a *primary-foreign* key relationship. This way, the existence of a record in an attribute classified as *foreign* key depends on the existence of the related record on the other set, in which the *primary* key is equal to that *foreign* key. To implement such feature one must provide to the DBMS capabilities to reinforce relational integrity rules. In other words, the server must be able to recognize such a relationship to guarantee it will be respected by issued queries.

An example of the applicability of this concept is an e-commerce database. Best practices dictate that user data should be stored separately from products and orders. Thus, one may model it as in Fig. 3. When a new order arrives, it is clear that a user chose some product and informed the store about his intent to buy it. Users and products are concrete elements. However, a sale is an abstract object and cannot happen without a buyer and a product. This way, to maintain the consistency of the database the DBMS must assure that no sale record will exist without relating user and product. This can be achieved by constructing the sales table such

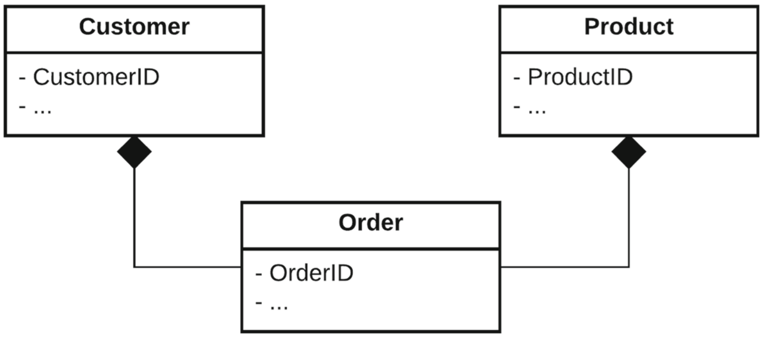

Fig. 3 Simple diagram describing the interaction between users and products composing the information regarding an order. Notice that the existence of users and products is independent, but there is a dependence for orders

that records contain *foreign* keys for the user and product tables (implying that these contain attributes classified as *primary* keys). By definition this feature imposes an inherent requirement that the DBMS has knowledge about this relationship between records on different tables. Any approach to protect the attributes against third parties will affect the DBMS itself and will never really achieve the needed protection. Thus, any effort on implementing secure relational integrity is at best security through obscurity[5].

6 The winner solution of Netflix's prize

The winner of Netflix Grand Prize was BellKor's Pragmatic Chaos team, who built a solution over the progress achieved in the 2007 and 2008 Progress Prizes [55]. Several machine learning predictors were combined in the final solution with the objective of anticipating the suitability of Netflix content for some user considering previous behavior in the platform. The foundation used for this considered diverse factors, such as:

- What is the general behavior of users when rating? What is the average rating?
- How critic is a user and how this changes over time?
- Does the user demonstrate preference for a specific movie or gender?
- Does the user demonstrate preference for blockbusters or non-mainstream content?
- What property of a movie affects the rating? Is there a correlation between the rating of a user and the presence of a particular actor in a particular gender?

The strategy used to combine these factors (and many others) escapes the scope of this work. We should attend only to the necessity of extracting data from the dataset to feed the learning model.

6.1 Searching the encrypted Netflix's database

An interesting application of our framework is enabling an entity to maintain an encrypted database on third party hardware with a similar structure of Netflix's dataset and being able to implement a prediction algorithm with minimum data leakage to the DBMS. The database should be capable of answering the requested predicates regarding user behavior.

Two scenarios must be considered: the recommendation system running on Netflix's infrastructure, and the dataset becoming public. The former offers an execution environment apparently honest (no one would share data with an openly malicious party) but that can be compromised at some point. To mitigate the damage, the data owner can implement different strategies to reduce the usefulness of any leakage that might happen. Thus, data being handled exclusively in encrypted form on the server is a natural option, since security breaches would reveal nothing but incomprehensible ciphertexts. This is the best case scenario since the data owner has as much control of the execution machine as possible, so our framework proposal can be applied in its full capacity.

As an example of the latter, an important feature required for running the Netflix's prize is the capability of in-dataset comparisons. This time any security solution should find the balance between protecting data secrecy and offering conditions for experimentation. Moreover, we must consider that the execution environment cannot be considered honest anymore. This way, the suitability of our framework depends on the relaxation of the indexing method. *index* values must be published to enable comparisons. For instance, both sides of Lewi-Wu's ciphertexts should be published, or even an OPE scheme may be used on the encryption of the index. From the perspective of the secrecy of ciphertexts, if a IND-OCPA scheme is used then there will be no security reduction beyond what the corresponding threat model expects, as discussed in Section 2.1. The adversary learns the ciphertext order but has restricted ability to make inferences using information acquired from public databases. The only strategy that can be applied uses the data distribution in the dataset (that can be retrieved by enabling comparisons), which puts an attacker in this scenario in a very similar position than the persistent passive attacker.

Given the boundary conditions for privacy preservation, we cannot precisely state the robustness of our framework in the context of the Netflix prize. It clearly increases the hardness against an inference attack, since the adversary is unable to observe the plaintext, but the distribution leaked will give him hints about its content. For instance, the correlation of age groups and most watched (or better rated) movies. It is a fact that all these are expressed as ciphertexts, but as previously stated, a motivated adversary may be able to combine such hints and defeat our security barriers.

Our framework performs much better in the more conservative scenario, where a production server provides recommendations to users with comparisons controlled by the data owner through the two-sided *index* attributes. The impossibility for arbitrary comparisons makes snapshot attacks completely infeasible.

As previously discussed, a motivated adversary with access to the database may be able to also retrieve logs and auxiliary collections. Consequently, previous queries may leak the second side of *index* ciphertexts and recall the danger of persistent passive attacks. So, an important feature for future work is the development of a key refreshment algorithm to nullify the usefulness of such information.

6.2 Data structure

The dataset shared by Netflix is composed by more than 100 million real movie ratings from 480,000 users about 17,000 movies, made between 1999 and 2005, and formatted as a training test set [11, 55]. It contains a subset of 4.2 million of those ratings, with up to 9 ratings per user. It consists of:

- *CustomerID:* A unique identification number per user,
- *MovieID:* A unique identification number per movie,
- *Title:* The English title of the movie,
- *YearOfRelease:* The year the movie was released,
- *Rating:* The rating itself,
- *Date:* The timestamp informing when the rating happened.

6.3 Constructing queries of interest over encrypted data

Following we rewrite some of the main predicates required for BellKor's solution using the relational algebra of Section 5.2, thus enabling their execution over an encrypted dataset.

Let

- $\mathcal{DB}$ be a dataset as described in Section 6.2,
- AID be the CustomerID related to a particular user (that we shall call Alice),
- BID be the CustomerID related to a particular user different to Alice (that we shall call Bob),
- MID be the MovieID related to an arbitrary movie in the dataset (that we should refer as $\mathcal{M}$),
- $\mathcal{T} = (\mathcal{T}_{\text{start}}, \mathcal{T}_{\text{end}})$ be a time interval of interest,
- $\mathcal{T}_{\text{first-alice}}$ be the timestamp of the first rating Alice ever made,
- $\mathcal{C}()$ be a function that receives a set and returns the quantity of items contained,
- r_{H} and r_{L} be thresholds for extreme ratings characterizing users that hated or loved a movie,
- $\sigma_{Date \in \mathcal{T}}(\mathcal{DB}) \equiv \sigma_{Date\ \geq\ \mathcal{T}_{\text{start}}}(\mathcal{DB}) + \sigma_{Date\ <\ \mathcal{T}_{\text{end}}}(\mathcal{DB})$,
- $f(X) = \frac{\sum_{x \in X} \pi_{Rating}(x)}{\mathcal{C}(X)}$.

Then, some of the required predicates for BellKor's solution are:

- **Movies rated by Alice:** Returns all movies that received some rating from Alice. For

  $$\text{U}(X) = \sigma_{CustomerID=X}(\mathcal{DB}),$$

  we have the query

  $$\pi_{\text{MovieID}}(\mathcal{U}(\text{AID})). \tag{1}$$

- **Users who rated M:** Returns all users that sent some rating for MID. For

  $$M(X) = \sigma_{MovieID=\text{MID}}(\mathcal{DB}),$$

  we have the query

  $$\pi_{CustomerID}(\mathcal{M}(\text{MID})). \tag{2}$$

- **Average of Alice's ratings over time:** Computes the average of all rates sent by Alice during a particular time interval $\mathcal{T}$. For

  $$\mathcal{A}_{\text{AID},\mathcal{T}} = \sigma_{Date \in \mathcal{T}}(\mathcal{U}(\text{AID})),$$

  we have that

  $$\text{avg}(\text{AID}, \mathcal{T}) = \begin{cases} f(\mathcal{A}_{\text{AID},\mathcal{T}}) \text{ if } \mathcal{C}(\mathcal{A}_{\text{AID},\mathcal{T}}) > 0, \\ 0, \text{otherwise.} \end{cases} \tag{3}$$

- **Average of ratings for a particular movie M in a timeset:** Computes the average of all rates sent by all users during a particular time interval $\mathcal{T}$ for a movie $\mathcal{M}$. For

  $$\mathcal{M}_{\text{MID},\mathcal{T}} = \sigma_{Date \in \mathcal{T}}(M(\text{MID}))$$

  we have that

  $$\text{avg}(\text{MID}, \mathcal{T}) = \begin{cases} f(\mathcal{M}_{\text{MID},\mathcal{T}}) \text{ if } \mathcal{C}(\mathcal{M}_{\text{MID},\mathcal{T}}) > 0, \\ 0, \text{otherwise.} \end{cases} \tag{4}$$

- **Number of days since Alice's first rating:** Computes how many days have been since the Alice submitted the first rating of movie, relative to a moment $\mathcal{I}$.

  $$\text{dsf}(AID, \mathcal{I}) = \mathcal{I} - \pi_{Date}(\sigma_{min(Date)}(\mathcal{U}(\text{AID}))). \tag{5}$$

- **Quantity of users who hated $\mathcal{M}$:** Counts the quantity of very bad ratings $\mathcal{M}$ received since its release.

  $$\mathcal{C}_H(\mathcal{M}) = \mathcal{C}\left(\sigma_{\text{MovieID=MID}}(\sigma_{\text{Rating} \leq r_{\text{H}}}(\mathcal{DB}))\right). \tag{6}$$

- **Quantity of users who loved $\mathcal{M}$:** Counts the quantity of very good ratings $\mathcal{M}$ received since its release.

  $$\mathcal{C}_L(\mathcal{M}) = \mathcal{C}\left(\sigma_{MovieID=\text{MID}}(\sigma_{\text{Rating} \geq r_{\text{L}}}(\mathcal{DB}))\right). \tag{7}$$

- **Users that are similar to Alice:** The similarity assessment between users require the derivation of a specific metric according to the boundary-conditions. The winning solution developed a sophisticated strategy, building a graph of neighborhoods considering similar movies and users and computing a weighted mean of the ratings. For simplicity, we shall condense two factors that can be used for this objective: the set of common rated movies, and how close the ratings are. To query the movies rated both by Alice and Bob, let

  $$\alpha_{\text{AID}} = \pi_{\text{MovieID,RatingA}}(\rho_{Rating,RatingA}(\mathcal{U}(\text{AID})))$$

and

$$\beta_{\mathrm{BID}} = \pi_{\mathrm{MovieID,RatingB}}(\rho_{Rating,RatingB}(\mathcal{U}(\mathrm{BID}))).$$

Then

$$\textsc{SimilaritySet}\ (\mathrm{AID}, \mathrm{BID}) = \alpha_{AID} \bowtie \beta_{BID} \quad (8)$$

returns a sequence of tuples of ratings made by Alice and Bob. A simple approach for evaluating proximity is to compute the average of the difference of ratings for each movie returned by Eq. 8, as shown in Eq. 9.

$$\frac{\sum_{\textsc{SimilaritySet}(\mathrm{AID,BID})} |\mathrm{RatingA} - \mathrm{RatingB}|}{\mathcal{C}\ (\textsc{SimilaritySet}\ (\mathrm{AID}, \mathrm{BID}))} \quad (9)$$

7 Implementation

A proof-of-concept implementation of the proposed framework was developed and made available to the community under a *GNU GPLv3* license [29]. It runs upon the popular document-based database MongoDB and was designed as a wrapper over its Python driver [56]. Hence, we are able to evaluate its competence as a search framework as well as the compatibility with a state-of-the-art DBMS. Moreover, running as a wrapper makes it database-agnostic and restricts the server to dealing with encrypted data. We choose to implement our wrapper over a NoSQL database so we could avoid dealing with the SQL interpreter and thus reduce the implementation complexity. However, our solution should be easily portable to any SQL database because of its strong roots in relational algebra. Table 1 provides the schemes used for each attribute class, the parameter size and its security level.

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 [34]. Finally, Paillier and ElGamal are well-known public-key schemes. Both achieve IND-CPA and are based on the hardness of solving integer factorization and discrete logarithm problems, respectively. Paillier supports homomorphic addition, while ElGamal provides homomorphic multiplication. Both are classified as PHE schemes [38, 39]. The implementation of AES was provided by the *pycrypto* toolkit [57]; we wrote a Python binding over the implementation of Lewi-Wu provided by the authors [58]; and we implemented Paillier and ElGamal schemes. An AVL tree was used as the index structure. It is important to notice that performance was not the main focus in this proof-of-concept implementation.

The machines used to run our experiments are described in Tables 2 and 3. The former specifies the machine used to host the MongoDB server, and latter describes the one used to run the client. Both machines were connected by a Gigabit local network connection.

While it was trivial to index the plaintext dataset 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. 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. Such limitation brings a major performance overhead that especially affects range queries.

7.1 Netflix's prize dataset

We used the Netflix's dataset to measure the computational costs of managing an encrypted database.

We consider the two threat scenarios discussed in Section 6.1, a recommendation system running in production, and the disclosure of a real ratings dataset. Both require the ability of running all queries presented in Section 6.3, differing only in the content that must be inserted in the encrypted dataset (for instance, how much of the *index* ciphertexts may be stored). Hence, to demonstrate the suitability of our framework as a strategy to fulfill the development and execution of a good predictor in such contexts, and being capable of mitigating damages to user privacy, we implemented those queries in an encrypted instance of the dataset.

As shown in Table 4, the four attributes chosen are classified as *static*, which use the faster encryption and decryption available. *Rating* is tagged *computable* for addition and multiplication, thus being compatible with Eqs. 3 and 4. We use *CustomerID*, *MovieID*, and *Date* for indexing. Encrypting the document structure takes 540μs per record.

There is no way to implement integer division over Paillier ciphertexts. Thus, the predictor may be adapted to

Table 1 Chosen cryptosystems for each attribute presented in Section 5

Attribute	Cryptosystem	Parameters	Sec. level
static	AES	128 bits	128 bits
index	Lewi-Wu	128 bits	128 bits
computable (+)	Paillier	3072 bits	128 bits
computable (×)	ElGamal	3072 bits	128 bits

Table 2 Specifications of the machine used for running the MongoDB instance

CPU	2 x Intel Xeon E5-2670 v1 @ 2.60GH
OS	CentOS 7.3
Memory	16 x DDR3 DIMM 8192MB @ 1600MHz
Disk	7200RPM Western Digital HDD (SATA)

Table 3 Specifications of the machine used for running the queries described in this document

CPU	2 x Intel Xeon E5-2640 v2 @ 2.60GH
OS	Ubuntu 16.04.2
Memory	4 x DDR3 DIMM 8192MB @ 1600MHz
Disk	7200RPM Western Digital HDD (SATA)

use the non-divided result on Eqs. 3 and 4. Otherwise, a division oracle must be provided, to which one could submit their homomorphically added values and ask for a ciphertext equivalent to its division by an arbitrary integer. This approach does not reduce security for an IND-CPA homomorphic scheme.

Handling such a large dataset was not an easy task. The ciphertext expansion factor caused by AES, Paillier and ElGamal cryptosystems was relatively small, but the Lewi-Wu implementation is very inefficient in this regard, having an expansion of about 400×. This directly affects the index building and motivated us to explore different strategies to encrypt and load the dataset to a MongoDB instance in reasonable time.

Again, MongoDB is not friendly for custom indexing. A contribution by Grim, Wiersma and Turkmen to our code enables us to manage the AVL tree inside the database through JavaScript code stored inside MongoDB's engine (the only way to execute arbitrary code in MongoDB) [59]. Thus, our primary approach to feed the DBMS with the dataset was quite simple: encrypt each record in our wrapper, insert in the database, and update the index and balance the tree inside the DBMS. The two first operations suffered from an extremely high memory consumption and by far surpassed our available RAM capacity. However, an even worse problem we faced was to build the AVL tree. For the first thousand records we could do the node insertion and tree balancing with a transfer rate of about 600 documents per second, but it dropped quickly as the tree height increases, reaching less than 1 document per second before insertion of the 10,000th record.

We found out that the initial insertions required a novel approach. We completely decoupled the *index* from the *static* data encryption and chose to first feed the database with the *static* ciphertexts, constructing the entire AVL tree using the plaintext on client-sided memory, and then inserting it in the database. Moreover, to speed up the index construction we followed Algorithms 1 and 2 to construct the AVL tree. It takes a sorted list of inputs and builds the tree with time complexity of $O(n)$ on the list size. As a result of this approach we were able to build the encrypted database and the index by 3000 documents per second during the entire procedure.

Table 4 Attribute structure of elements in the Netflix's prize dataset

Name	Value type	Class
CustomerID	integer	*index, static*
MovieID	integer	*index, static*
Rating	integer	*static, computable*
Date	integer	*index, static*

Algorithm 1 Build an AVL tree using an array of documents

```
1: procedure BUILD_INDEX(docs)
2:     $docs_{sort} \leftarrow sort(docs)$;
3:     $docs_{group} \leftarrow group(docs_{sort})$;    ▷ Combine equal elements
4:     return $build\_aux(docs_{group}, 0, lenght(docs_{group}) - 1)$;
5: end procedure
```

Algorithm 2 Recursively builds an AVL tree with a sorted array of documents without repeated elements. Receives the array itself, and the indexes for the leftmost and rightmost elements to be handled in each recursive call

```
1: procedure BUILD_AUX(docs, L, R)
2:     if $L = R$ then
3:         return $new\_node(docs[L])$;
4:     else if $L + 1 = R$ then
5:         $left\_node \leftarrow new\_node(docs[L])$;
6:         $right\_node \leftarrow new\_node(docs[R])$;
7:         $left\_node.right = right\_node$;
8:         $left\_node.height = 1$;
9:         return $left\_node$;
10:    else
11:        $M \leftarrow L + \lfloor (R - L)/2 \rfloor$;
12:        $middle\_node \leftarrow new\_node(docs[M])$;
13:        $middle\_node.left \leftarrow build\_aux(docs, L, M-1)$;
14:        $middle\_node.right \leftarrow build\_aux(docs, M+1, R)$;
15:        $lh \leftarrow middle\_node.left.height$;
16:        $rh \leftarrow middle\_node.right.height$;
17:        $middle\_node.height = 1 + \max(lh, rh)$;
18:        return $middle\_node$;
19:    end if
20: end procedure
```

Table 5 shows the latency of each step we observed during the construction of the AVL tree-based indexes. The total time to build those 3 indexes was 40 min.

The queries we derived in Section 6.3 were ported to our encrypted database, and the latency for each one can be seen in Table 6. The parameters used for each Equation were arbitrarily selected. The *CustomerIDs* for Alice and

Table 5 Latency for each step in the construction of an AVL tree following Algorithm 1 for each *index* attribute specified in 4

Attribute	Sort (s)	Group (s)	Build_index (s)
CustomerID	329	459	129
MovieID	270	161	2
Date	187	197	5

Bob (AID and BID) were 1061110 and 2486445 respectively, while MID was fixed as 6287. The time interval used was 01/01/2003 to 01/01/2004. Lastly, we defined a "loved" rating as those greater than 3, and "hated" rating as those lower than 3. We applied some efforts in optimizing the execution, however these results can still be improved.

As it can be seen, complex queries composed by range selections, as well as those with numerous outcomes, suffered from the slow communication between server and the client. The latter influenced even the plaintext results. The outcome of Eq. 1 is quite small, requiring much less time to return than the outcome of Eq. 2 (the number of movies rated by a user is much smaller than the number of users that rated a movie).

The time interval selection in Eqs. 3 and 4 required our implementation to visit many nodes in the index tree for *Date*. Because each iteration requires a back and forth between the server and the client, this dramatically impacted the performance. The latencies for Eqs. 1 and 5 were only 1.4 times higher in the encrypted database, however it reached 710 times for Eq. 3. Lastly, Eqs. 6 and 7 depend on Paillier's homomorphic additions. This implied in a factor-12 slowdown.

The implementation of queries based on Eqs. 3 and 4 took the previous suggestion and skipped the final division. We believe this does not undermine any procedure that eventually consumes this outcome.

Table 6 Execution times for implementations of the equations presented in Section 6.3 on an encrypted MongoDB collection and an equivalent plaintext version

Equation	Encrypted	Plaintext
1	16.6 ms	11.9 ms
2	2 s	850 ms
3	2.7 s	3.8 ms
4	2.7 s	1.0 s
5	16.8 ms	11.8 ms
6 and 7	12 ms	1.0 ms
9	603 ms	200 ms

Each row contains the latency for the entire circuit required by the respective Equation and returning the outcome to the client. Times are computed as the average for 100 independent executions. The machine and parameters used in each cryptosystem follow those defined in Section 7

The optimal implementation of Eqs. 6 and 7 requires indexing of *MovieID* and *Rating* attributes. However, due to limitations in our implementation, rather than indexing the latter we use linear search over the outcome of the movie selection on client-side. Our approach for building indexes use the set data structure of MongoDB documents. Yet, in the most recent release such structure holds up to 16MB of data, much smaller than the required for indexing the entire dataset for *Rating* with our strategy.

Lastly, Eq. 8 was implemented aiming at the joining of data regarding two users, Alice and Bob. We let the evaluation of such information by a similarity-evaluation function as future work.

8 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, a relational algebra was built to support encrypted data composed by: projection, selection, Cartesian product, difference, union, rename, and join-like operators.

An overview of the security provided is discussed, as well as a performance analysis about the impact in a realistic database. As a case study we explored the Netflix prize, which published an anonymized dataset with real-world information about user behavior which was later deanonymized through correlation attacks involving public databases.

We offered a proof-of-concept implementation in Python over the document-based database MongoDB. To demonstrate its functionality, we selected and ran some of the main predicates required by the winning solution of the Netflix Grand Prize and measured the performance impact of the execution in a encrypted version of the dataset. We conclude that our proposal offers robustness against a compromised server and we discuss how it would help to avoid the deanonymization of the Netflix dataset. 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 compromise.

As future research objectives we can mention:

- *Extend the scope to associative arrays*: Despite being powerful on SQL, Codd's relational algebra is not completely applicable for non-relational databases. For instance, NoSQL and NewSQL databases lack the concept of joining. A more convenient foundation for such context is algebra of associative arrays [60].

Hence, the formalization of our primitives in such algebra would be an interesting work.

- *Reduce the leakage of index construction in the database*: Our proposal leaks both sides of *index* ciphertexts to enable the index construction. At this moment, an eavesdropper monitoring queries would learn all information required to freely compare the exposed ciphertexts. As discussed in this document, such capability must be restricted, under risk of enabling an inference attack.
- *Key refreshment algorithm*: A persistent passive attacker is capable of learning the required information to perform comparisons through the entire database, just by observing issued queries and its outcome. Thus, the framework primitives must be improved to support an algorithm capable of avoid any damage caused by the knowledge of such information.
- *Hide repeated queries*: Even with encrypted queries and outcomes, the access pattern in a database may indicate repeated queries and the associated records. A technique such as ORAM could be useful to protect such information [61].
- *Explore different databases*: As stated, MongoDB is a very popular NoSQL database. However, it is not friendly to custom indexing or third party code running in its engine. Thus, to replace it by a more appropriate database could provide a more productive system.
- *Improve performance of our implementation*: Our implementation had as objective to be a proof-of-concept and demonstrate how the proposal works. The development of a space and speed-optimized versions is an important next step.

Endnotes

[1] SHE stands for "Somewhat homomorphic encryption".

[2] LHE stands for "Leveled fully homomorphic encryption".

[3] Relative to a set of desirable properties for a database. Acronym to "Atomicity, Consistency, Isolation, Durability".

[4] The Advanced Encryption Standard (AES) is a well-established symmetric block cipher enabling high performance implementation in hardware and software [62].

[5] When the security of a system relies only in the lack of knowledge by adversaries about its implementation details and flaws.

Acknowledgements
A prior version of this paper was presented at the XVI Brazilian Symposium on Information and Computational Systems Security (SBSeg16) [63].
The authors thank Proof. André Santanchè for the initial opportunity to develop this work, the Multidisciplinary High Performance Computing Lab (LMCAD) for providing the required infrastructure, and the anonymous reviewers that helped improving this work.

Funding
This research was partially founded by CNPq and the Google Research Awards Latin America.

Availability of data and materials
Our proof-of-concept implementation is publicly available on GitHub, as well as a generator for a synthetic dataset used for testing [29]. The Netflix dataset is available in academic repositories [64].

Authors' contributions
The first author developed the study design, carried out the implementation efforts and wrote most the paper. The second author contributed with discussions about the proposal and its validation with the case study. Both authors read and approved the final manuscript.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Received: 5 September 2017 Accepted: 6 December 2017
Published online: 03 January 2018

References
1. Vecchiola C, Pandey S, Buyya R. High-Performance Cloud Computing: A View of Scientific Applications. In: Pervasive Systems, Algorithms, and Networks (ISPAN), 2009 10th International Symposium On. Kaohsiung: IEEE. 2009. p. 4–16.
2. Buyya R. Market-Oriented Cloud Computing: Vision, Hype, and Reality of Delivering Computing As the 5th Utility. In: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGRID '09. Washington: IEEE Computer Society. 2009.
3. Hoffa C, Mehta G, Freeman T, Deelman E, Keahey K, Berriman B, Good J. On the Use of Cloud Computing for Scientific Workflows. In: eScience 08. IEEE Fourth International Conference On. Indianapolis: IEEE. 2008. p. 640–5.
4. Dinh HT, Lee C, Niyato D, Wang P. A survey of mobile cloud computing: architecture, applications, and approaches. Wirel Commun Mob Comput. 2013;13(18):1587–611.
5. Xiao Z, Xiao Y. Security and Privacy in Cloud Computing. IEEE Commun Surv Tutor. 2013;15(2):843–59.
6. Pascual A. 2017 Data Breach Fraud Impact Report: Going Undercover and Recovering Data. Technical report: Javelin Advisory Services; 2017.
7. Popa RA, Redfield CMS, Zeldovich N, Balakrishnan H. Cryptdb: Protecting confidentiality with encrypted query processing. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. SOSP '11. New York: ACM. 2011. p. 85–100.
8. Poddar R, Boelter T, Popa RA. Arx: A Strongly Encrypted Database System: Cryptology ePrint Archive, Report 2016/591; 2016.
9. Arasu A, Eguro K, Kaushik R, Kossmann D, Ramamurthy R, Venkatesan R. A secure coprocessor for database applications. In: 2013 23rd International Conference on Field programmable Logic and Applications. 2013. p. 1–8. doi:10.1109/FPL.2013.6645524.
10. Tu S, Kaashoek MF, Madden S, Zeldovich N. Processing analytical queries over encrypted data. Proc VLDB Endow. 2013;6(5):289–300.
11. Bennett J, Lanning S, et al. The netflix prize. In: Proceedings of KDD Cup and Workshop. New York. 2007. p. 35.
12. Arrington M. AOL Proudly Releases Massive Amounts of Private Data. 2006. https://techcrunch.com/2006/08/06/aol-proudly-releases-massive-amounts-of-user-search-data/ Accessed 24 July 2017.
13. Said A, Bellogín A. Comparative recommender system evaluation: benchmarking recommendation frameworks. In: Proceedings of the 8th ACM Conference on Recommender Systems. New York: ACM. 2014. p. 129–36.
14. Wang Z, Liao J, Cao Q, Qi H, Wang Z. Friendbook: a semantic-based friend recommendation system for social networks. IEEE Trans Mob Comput. 2015;14(3):538–51.
15. Pazzani MJ, Billsus D. Content-based recommendation systems. In: Brusilovsky P, Kobsa A, Nejdl W, editors. The Adaptive Web: Methods and Strategies of Web Personalization. Berlin: Springer. 2007. p. 325–41.

16. Narayanan A, Shmatikov V. Robust de-anonymization of large sparse datasets. In: Proceedings of the 2008 IEEE Symposium on Security and Privacy. SP '08. Washington: IEEE Computer Society. 2008. p. 111–25.
17. Barbaro M, Zeller T. A Face Is Exposed for AOL Searcher No. 4417749: The New York Times; 2006. Accessed 05 Apr 2017.
18. Narayanan A, Felten EW. No silver bullet: De-identification still doesn't work: White Paper; 2014. Manuscript. http://randomwalker.info/publications/no-silver-bullet-de-identification.pdf.
19. Greenwald G, MacAskill E. NSA Prism program taps in to user data of Apple, Google and others: The Guardian; 2013. https://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data.
20. Weber H. How the NSA & FBI made Facebook the perfect mass surveillance tool: Venture Beat; 2014. https://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data.
21. Thomsen S. Extramarital affair website Ashley Madison has been hacked and attackers are threatening to leak data online: Business Insider; 2015. Accessed 25 May 2016.
22. Magnusson N, Rolander N. Sweden Tries to Stem Fallout of Security Breach in IBM Contract: Bloomberg; 2017.
23. BBC News. Yahoo 'state' hackers stole data from 500 million users. 2016. Accessed 23 Sept 2016.
24. Sweeney L. Simple Demographics Often Identify People Uniquely. 2000. http://dataprivacylab.org/projects/identifiability/.
25. Golle P. Revisiting the uniqueness of simple demographics in the us population. In: Proceedings of the 5th ACM Workshop on Privacy in Electronic Society. WPES '06. New York: ACM. 2006. p. 77–80.
26. DuckDuckGo. Privacy Mythbusting #3: Anonymized data is safe, right? (Er, no.) https://spreadprivacy.com/dataanonymization-e1e2b3105f3c. Accessed 24 July 2017.
27. Schneier B. Data is a toxic asset. 2016. http://edition.cnn.com/2016/03/01/opinions/data-is-a-toxic-asset-opinion-schneier/index.html.
28. Bösch C, Hartel P, Jonker W, Peter A. A survey of provably secure searchable encryption. ACM Comput Surv. 2014;47(2):18–11851.
29. Alves P. A proof-of-concept searchable encryption backend for mongodb. 2016. https://github.com/pdroalves/encrypted-mongodb. Accessed July 2017.
30. Bellare M, Desai A, Pointcheval D, Rogaway P. Relations among notions of security for public-key encryption schemes. In: Advances in Cryptology — CRYPTO '98: 18th Annual International Cryptology Conference Santa Barbara, California, USA August 23–27, 1998 Proceedings. Berlin: Springer. 1998. p. 26–45.
31. Curtmola R, Garay J, Kamara S, Ostrovsky R. Searchable symmetric encryption: Improved definitions and efficient constructions. J Comput Secur. 2011;19(5):895–934.
32. Boldyreva A, Chenette N, Lee Y, O'Neill A. Order-preserving symmetric encryption. Lect Notes Comput Sci. 2009;5479:224–41.
33. Boneh D, Lewi K, Raykova M, Sahai A, Zhandry M, Zimmerman J. Semantically Secure Order-Revealing Encryption: Multi-input Functional Encryption Without Obfuscation. Berlin: Springer. 2015. p. 563–94.
34. Lewi K, Wu DJ. Order-Revealing Encryption: New Constructions, Applications, and Lower Bounds: Cryptology ePrint Archive; 2016. Report 2016/612.
35. Kolesnikov V, Shikfa A. On the limits of privacy provided by Order-Preserving Encryption. Bell Labs Tech J. 2012.
36. Chenette N, Lewi K, Weis SA, Wu DJ. Practical Order-Revealing Encryption with Limited Leakage. In: FSE. Bochum: Springer. 2016.
37. Naveed M, Kamara S, Wright CV. Inference attacks on property-preserving encrypted databases. In: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security. CCS '15. New York: ACM. 2015. p. 644–55.
38. Paillier P. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In: Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques, EUROCRYPT'99. Berlin: Springer. 1999. p. 223–38. http://dl.acm.org/citation.cfm?id=1756123.1756146.
39. El Gamal T. A public key cryptosystem and a signature scheme based on discrete logarithms. In: Proceedings of CRYPTO 84 on Advances in Cryptology. New York: Springer. 1985. p. 10–18. http://dl.acm.org/citation.cfm?id=19478.19480.
40. Gentry C. Computing Arbitrary Functions of Encrypted Data. Commun ACM. 2010;53(3):97–105.
41. Brakerski Z, Gentry C, Vaikuntanathan V. (leveled) fully homomorphic encryption without bootstrapping. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference. ITCS '12. New York: ACM. 2012. p. 309–25.
42. Loftus J, May A, Smart NP, Vercauteren F. On CCA-Secure Somewhat Homomorphic Encryption. In: Proceedings of the 18th International Conference on Selected Areas in Cryptography. SAC'11. Berlin: Springer. 2012. p. 55–72.
43. Song DX, Wagner D, Perrig A, Perrig A. Practical techniques for searches on encrypted data. In: Proc 2000 IEEE Symp Secur Privacy. S&P 2000. Berkeley: IEEE. 2000. p. 44–55.
44. Grubbs P, Ristenpart T, Shmatikov V. Why Your Encrypted Database Is Not Secure: Cryptology ePrint Archive; 2017. Report 2017/468. http://eprint.iacr.org/2017/468.
45. Rogaway P. The moral character of cryptographic work.: Cryptology ePrint Archive, IACR; 2015, p. 1162.
46. Naveed M, Kamara S, Wright CV. Inference attacks on property-preserving encrypted databases. In: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security. CCS '15. New York: ACM. 2015. p. 644–55.
47. Grubbs P, McPherson R, Naveed M, Ristenpart T, Shmatikov V. Breaking web applications built on top of encrypted data. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. CCS '16. New York: ACM. 2016. p. 1353–64.
48. Papadimitriou A, Bhagwan R, Chandran N, Ramjee R, Haeberlen A, Singh H, Modi A, Badrinarayanan S. Big data analytics over encrypted datasets with seabed. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. OSDI'16. Berkeley: USENIX Association. 2016. p. 587–602. http://dl.acm.org/citation.cfm?id=3026877.3026922.
49. Shoro AG, Soomro TR. Big data analysis: Apache spark perspective. Global J Comput Sci Technol. 2015;15(1).
50. Codd EF. A relational model of data for large shared data banks. Commun. ACM 26. 1983;6:64–69.
51. Ramakrishnan R, Gehrke J. Database Management Systems, 3rd edn. New York: McGraw-Hill, Inc.; 2003.
52. Sedgewick R, Wayne K. Algorithms, 4th edn: Addison-Wesley Professional; 2011.
53. Doröz Y, Hoffstein J, Pipher J, Silverman JH, Sunar B, Whyte W, Zhang Z. Fully Homomorphic Encryption from the Finite Field Isomorphism Problem: Cryptology ePrint Archive; 2017. Report 2017/548.
54. Boneh D, Gentry C, Halevi S, Wang F, Wu DJ. Private database queries using somewhat homomorphic encryption. In: Proceedings of the 11th International Conference on Applied Cryptography and Network Security. ACNS'13. Berlin: Springer. 2013. p. 102–18.
55. Töscher A, Jahrer M, Bell RM. The BigChaos Solution to the Netflix Grand Prize. 2009. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.178.1356&rep=rep1&type=pdf.
56. Chodorow K, Dirolf M. MongoDB: The Definitive Guide, 1st edn. USA: O'Reilly Media, Inc.; 2010.
57. Litzenberger D. Python Cryptography Toolkit. 2016. http://www.pycrypto.org/. Accessed 3 July 2016.
58. Wu DJ, Lewi K. FastORE. 2016. https://github.com/kevinlewi/fastore. Accessed July 2017.
59. Grim MW, Wiersma AT, Turkmen F. Security and Performance Analysis of Encrypted NoSQL Databases. Technical report: University of Amsterdam; 2017. http://www.delaat.net/rp/2016-2017/p37/report.pdf.
60. Kepner J, Gadepally V, Hutchison D, Jananthan H, Mattson TG, Samsi S, Reuther A. Associative array model of sql, nosql, and newsql databases. Waltham: IEEE. 2016.
61. Stefanov E, Van Dijk M, Shi E, Fletcher C, Ren L, Yu X, Devadas S. Path oram: an extremely simple oblivious ram protocol. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. Berlin: ACM. 2013. p. 299–310.
62. Daemen J, Rijmen V. The design of Rijndael: AES-the advanced encryption standard. Springer Science & Business Media. 2013.
63. Alves PGMR, Aranha DF. A framework for searching encrypted databases. In: Proceedings of the XVI Brazilian Symposium on Information and Computational Systems Security. Niterói: SBC. 2016.
64. Netflix Prize Data Set; 2009. http://academictorrents.com/details/9b13183dc4d60676b773c9e2cd6de5e5542cee9a. Accessed July 2017.