2017 IEEE 36th Symposium on Reliable Distributed Systems

A Practical Framework for Privacy-Preserving NoSQL Databases

Ricardo Macedo*, João Paulo*, Rogério Pontes*, Bernardo Portela†, Tiago Oliveira†, Miguel Matos‡, Rui Oliveira*
*HASLab - High-Assurance Software Lab, INESC TEC & U. Minho, Portugal.
†HASLab - High-Assurance Software Lab, INESC TEC & FCUP, Portugal.
‡INESC ID/IST, U. Lisboa, Portugal.

***Abstract*—Cloud infrastructures provide database services as cost-efficient and scalable solutions for storing and processing large amounts of data. To maximize performance, these services require users to trust sensitive information to the cloud provider, which raises privacy and legal concerns. This represents a major obstacle to the adoption of the cloud computing paradigm.**

Recent work addressed this issue by extending databases to compute over encrypted data. However, these approaches usually support a single and strict combination of cryptographic techniques invariably making them application specific. To assess and broaden the applicability of cryptographic techniques in secure cloud storage and processing, these techniques need to be thoroughly evaluated in a modular and configurable database environment. This is even more noticeable for NoSQL data stores where data privacy is still mostly overlooked.

In this paper, we present a generic NoSQL framework and a set of libraries supporting data processing cryptographic techniques that can be used with existing NoSQL engines and composed to meet the privacy and performance requirements of different applications. This is achieved through a modular and extensible design that enables data processing over multiple cryptographic techniques applied on the same database. For each technique, we provide an overview of its security model, along with an extensive set of experiments. The framework is evaluated with the YCSB benchmark, where we assess the practicality and performance tradeoffs for different combinations of cryptographic techniques. The results for a set of macro experiments show that the average overhead in NoSQL operations performance is below 15%, when comparing our system with a baseline database without privacy guarantees.

I. Introduction

Nowadays, cloud computing is an ubiquitous technology capable of satisfying the most demanding storage and processing workloads [1]. Its benefits are well known: virtual infinite resources, fine-grained resource allocation, no up-front infrastructure costs, and constant access from anywhere at any time. Outsourcing databases to cloud providers became a logical step for many IT organizations to reduce costs and provide a good quality of service.

The cloud computing model assumes a level of trust on the provider that is not realistic for many applications. Several security concerns are raised when users want to offload sensitive data to a cloud provider. Once data is outsourced to the cloud, the users' control over that data is lost. This is an issue even if one could trust the cloud provider not break confidentiality, as recent reports have shown that cloud services often have security flaws that can result in the leakage of sensitive information [2]. These privacy issues justify why many enterprises holding sensitive data are reluctant to adopt the cloud paradigm. Information may be sensitive for a number of reasons, *e.g.*, if it is personal data, part of a business' competitive advantage or even due to regulations designed to ensure privacy or confidentiality, such as the novel European General Data Protection Regulation (GDPR)[1].

The protection of data is generally achieved using efficient encryption standards [3]. However, these approaches inherently prevent any sort of computation to be performed over encrypted data. This has motivated research and development of cryptographic techniques that provide a restricted set of computations to be performed over encrypted data, such as equality and range queries [4, 5]. The development of secure databases can thus be achieved in a variety of different ways, with varying levels of performance and security trade-offs.

Finding the cryptographic mechanisms best suited for the functionality, performance and security requirements of specific applications is a non-trivial task. On one hand, strong security can lead to a system that is neither available nor scalable. On the other hand, disregarding privacy towards performance can lead to unmet security requirements preventing real-world deployment. The careful selection of cryptographic techniques is, therefore, highly dependent on the expected application workload, and on the limitations imposed to sensitive data. This task gets increasingly complex once one considers that real-world databases store different types of information with (potentially widely) varying levels of privacy requirements.[2]

A great amount of research is centered around privacy-aware SQL databases as these have for many years been the standard systems for applications to store and query data [6, 7]. More recently, NoSQL databases have emerged to address the high-scalability and availability needs of some applications, which can afford relaxed data consistency guarantees [8, 9]. NoSQL databases have a simplified API which has been left unprotected and is subpar to the existing security guarantees of SQL queries. As a matter of fact, research on secure NoSQL databases is still poorly addressed in the literature [10, 11]. This paper aims to

[1] http://www.eugdpr.org.

[2] According to GDPR, users' personal information (*e.g.*, name, date of birth, citizen id) must be encrypted while information generated by the application that does not identify the users, does not need the same level of protection.

978-1-5386-1679-6/17 $31.00 © 2017 IEEE
DOI 10.1109/SRDS.2017.10

IEEE computer society

make NoSQL databases up-to-date to current security standards and guarantees of SQL databases.

Moreover, current secure SQL and NoSQL database solutions usually support a single and strict combination of cryptographic techniques invariably making them application specific. Our contributions in this paper stem from the idea that the design of privacy-preserving databases should be supported by a modular and extensible architecture, enabling a granular specification of functional and security requirements. This would allow for cloud developers to devise highly scalable NoSQL databases with security mechanisms tailored to the application at hand, which in turn maximizes system performance and throughput, while ensuring an adequate level of privacy for the system to be securely deployed on the cloud. In summary, our main contributions are:

- SAFENOSQL, a generic framework supporting existing NoSQL engines able to meet the privacy and performance requirements of different applications. This framework has a modular and extensible design that enables data processing over multiple cryptographic techniques applied on the same database schema.
- A SAFENOSQL prototype based on Apache HBase, along with a set of libraries implementing different data processing cryptographic techniques.
- An extensive evaluation of the prototype with micro and macro experiments under different representative application scenarios. The results of macro experiments show that the average overhead in NoSQL operations performance is less than 15% with respect to an HBase deployment without privacy-preserving guarantees.

The paper is structured as follows: Section II presents the relevant related work of privacy-aware databases. Section III describes the cryptographic schemes to be applied on the prototype and the security models that must be considered. The modular and extensible architecture for privacy-aware NoSQL databases SAFENOSQL is presented in Section IV. The implementation of a prototype that follows the architecture is then presented in Section V. Section VI presents an extensive experimental evaluation using realistic workloads. The paper concludes in Section VII with relevant observations and future work.

II. RELATED WORK

Several approaches have been proposed to address the gap in NoSQL privacy-preserving databases. In the BigSecret system, stored data is protected with standard encryption, while the indexes are encoded using techniques that allow comparisons (pseudo-random functions) and range queries (order-preserving partitioning) [10]. *Yuan et al.* employ algorithms of searchable encryption to build a privacy-preserving key-value store on top of the Redis database [11]. The values are protected with symmetric encryption and the keys are secured with pseudo-random functions. However, this approach provides a restricted set of features and a low modularity, since to provide more computation capabilities, the key-value pair must be rewritten in order to append to the key more information about the corresponding value. As a distinct solution, SafeRegions combines secret sharing and multi-party computation to perform secure NoSQL queries on three independent and untrusted HBase clusters [12]. Furthermore, this solution provides simultaneously secure computation over the stored values and security guarantees similar to standard encryption. Finally, in Arx a variant of order-preserving encryption with stronger security guarantees is proposed [13]. NoSQL queries are rewritten by a proxy at the trusted premises and a backend component deployed at the untrusted premises is used to perform computation over encrypted data. Messages exchanged between the proxy and backend component are done with a SQL dialect which requires translating queries to NoSQL language when the system interacts with NoSQL applications or NoSQL backend components. All previous solutions have been designed considering a specific, and hence restricted, set of data protection techniques. The main advantage of our work is that it provides a modular and flexible design where these and other techniques, with varying performance and security guarantees, can easily be supported.

In a different context, yet relevant for this paper, the design of secure databases has also been explored in the SQL paradigm. In CryptDB, a client rewrites SQL queries so that the database engine can execute them over protected data at untrusted environments [6]. It leverages different layered security schemes, allowing for the execution of equality checking and range queries, and generic sums and multiplications over encrypted data. A different approach is presented in Monomi, a CryptDB-based framework, able to perform secure data analytics by splitting the queries execution between the server and the client, allowing complex SQL queries to be performed [7]. An alternative solution proposed by the L-EncDB system ensures sensitive data protection while preserving the same length, format and primitive type through a set of format-preserving encryption techniques with deterministic properties, alongside an order-preserving encryption scheme to support most of the SQL queries [14]. In *Xiang et al.*, a secret sharing scheme is combined with order-preserving encryption to protect sensitive information stored and processed at the multiple untrusted cloud providers [15, 16]. It was also shown that it is possible to provide a SQL database, with a restricted range of supported queries, by relying solely on secret sharing and multi-party computation over three untrusted remote backends [17, 18]. Despite being a fundamentally different paradigm, these monolithic secure SQL database designs show the need of having a generic, flexible and modular framework solution able to combine different encryption mechanisms.

Another approach towards data privacy in the context of relational databases is to perform "at rest" data encryption. In this setting the cryptographic keys used to encrypt data are stored on trusted infrastructures with security measures that prevent attackers from corrupting the system and obtaining the keys [19, 20]. However, since the security mechanisms disallow most computations, some queries require the key to be given to the untrusted server, for decrypting information and responding to queries. Contrary to our deployment, where

the key remains within trusted premises at all times, this model has the disadvantage that if the cloud is corrupted while one of these queries is being processed at the untrusted server's memory, the attacker can retrieve the associated sensitive data.

III. Security

When considering various cryptographic techniques, the differences in privacy guarantees are observable in their respective security models. A more functionally restrictive encryption algorithm might ensure indistinguishability against a powerful adversary, while a different scheme allowing for some operations to be performed over encrypted data has to consider more limited adversaries. Throughout this work we explore, implement and analyse three different techniques for ensuring privacy-preserving computation that are widely used in different contexts, and that can be applicable towards enabling NoSQL processing over encrypted data. In this section, we describe each cryptographic technique, as well as the privacy guarantees and performance impact of the respective instantiations.

For clarity in presentation throughout the paper, we briefly establish a clear distinction between what we will refer to as *primitives* and *implementations*. A cryptographic primitive is a high-level description of a fixed set of algorithms and a security model defining the context where such a technique can be considered secure. A cryptographic implementation is an instantiation of a given primitive, typically a library that provides the required algorithms, and ensures the security properties enforced by it.

An encryption scheme generally consists in a triple of Probabilistic Polynomial Time (PPT) algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$. On input 1^λ, where λ is the security parameter, the key generation algorithm Gen returns a fresh key k. Upon input key k and message m, the encryption algorithm Enc returns a ciphertext m'. Upon input key k and ciphertext m', the decryption algorithm Dec returns the original message m. We require that $\mathsf{m} = \mathsf{Dec}(\mathsf{k}, \mathsf{Enc}(\mathsf{k}, \mathsf{m}))$ for all $\lambda \in \mathbb{N}$, all $\mathsf{k} \in \mathsf{Gen}(1^\lambda)$ and all m. We will now refine this definition to describe several primitives, and refer to our respective implementations.

A. Standard encryption

The encryption scheme considers a probabilistic encryption algorithm Enc. Our respective implementation follows the Advanced Encryption Standard [3], whose security guarantees adhere to the standard notions of semantic security definitions detailed in [21]. This entails a considerably robust level of security, meaning it is infeasible for a computationally bounded adversary to derive significant information about a message from the associated ciphertext.

Classical cryptographic algorithms ensuring semantic security are not designed to produce ciphertexts over which one can perform meaningful computations. The applicability of these techniques is, therefore, limited to the protection of data in scenarios where no operations are to be performed over it, such as data transmission or storage. For instance, retrieving a value encrypted in this fashion would require a full database retrieval and subsequent decryption, which is unfeasible for any realistic database deployment.

B. Deterministic encryption

The encryption scheme considers a deterministic encryption algorithm Enc, and must ensure that $\mathsf{m}_1 = \mathsf{m}_2 \Rightarrow \mathsf{Enc}(\mathsf{k}, \mathsf{m}_1) = \mathsf{Enc}(\mathsf{k}, \mathsf{m}_2)$ for all $\lambda \in \mathbb{N}$, all $\mathsf{k} \in \mathsf{Gen}(1^\lambda)$ and all $\mathsf{m}_1, \mathsf{m}_2$. The implementation of our scheme is achieved by adapting the Advanced Encryption Standard [3] to behave deterministically. The security guarantees of deterministic encryption schemes are formalized in [22], and somewhat follow the semantic security definitions in [21] with the caveat that the messages to be encrypted must have high min entropy conditioned on values of the other messages. One application example that fits this requirement would be the encryption of social security numbers, which likely share prefixes, but are otherwise uncorrelated.

From the properties of deterministic algorithms, ciphertexts can be compared without requiring the associated key, which also entails that the set of ciphertexts referring to the same data are known to the data holder as well. This approach of revealing duplicates to obtain functionality benefits is commonly used in secure storage systems relying on it for data deduplication [23]. In this setting, obtaining an encrypted value is trivial, but range querying would again require a full database retrieval and decryption, since equality does not suffice.

C. Order-preserving encryption

The encryption scheme considers a deterministic encryption algorithm Enc, and must ensure that $\mathsf{m}_1 > \mathsf{m}_2 \Rightarrow \mathsf{Enc}(\mathsf{k}, \mathsf{m}_1) > \mathsf{Enc}(\mathsf{k}, \mathsf{m}_2)$ for all $\lambda \in \mathbb{N}$, all $\mathsf{k} \in \mathsf{Gen}(1^\lambda)$ and all $\mathsf{m}_1, \mathsf{m}_2$. We follow the scheme proposed in [24]. Security guarantees of order-preserving encryption are formalized and discussed in [5]. The security of an OPE scheme is described via two main notions: *window one-wayness*, the expected margin of error for a server holding an OPE-encrypted database to extract original values, and *window distance one-wayness*, the expected margin of error for a server holding an OPE-encrypted database to extract distances between original values. Contrary to previous definitions, the security of OPE does not ensure strong security properties such as indistinguishability.

In addition to equality, the order-revealing nature of these schemes allows for comparisons to be performed over encrypted data. This is also the factor that enables inference attacks to be performed over data encrypted with OPE, where knowledge of data distribution can lead to high success rates for sensitive data extraction [25]. As such, the applicability of OPE should follow from these security limitations, for usage on high entropy attributes such as sequential identifiers, or for handling data with low sensitivity, where revealing partial information about the original values is not problematic. This considerable security downgrade has the significant advantage of allowing for equality and range queries to be executed efficiently on the server side.

IV. Architecture

The proposed architecture for SafeNoSQL framework aims to be generic, in order to be compatible with most of the

		Column	Column	Column
Row	Key 1	Value	Value	Value
Row	Key 2	Value	Value	Value
Row	Key 3	Value	Value	Value

Fig. 1: Logical view of a NoSQL table.

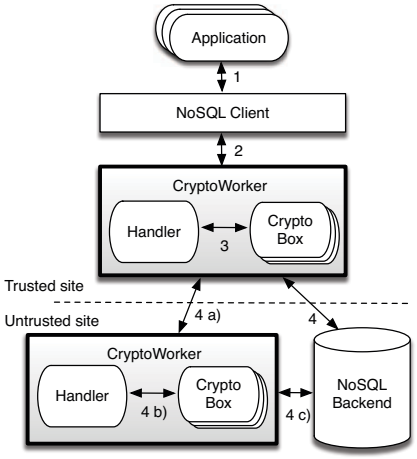


Fig. 2: Architecture of SAFENOSQL.

existing key-value based NoSQL databases [8, 9]. For this reason, and as depicted in Figure 1, we assume a typical logical table view where each key-value can be seen as a row indexed by a key. A key is associated with several values organized in different columns. In this context, the following set of operations is supported in our framework:

- **Put**. A new key-value pair is inserted or updated if the corresponding key already exists.
- **Get**. The key-value for a specific key is read.
- **Delete**. The key-value for a specific key is deleted.
- **Scan**. A range of key-value pairs is read.
- **Filter**. Search for one or more key-value pairs where a specific column matches a desired value.

Figure 2 shows the proposed architecture for SAFENOSQL. We consider two deployment environments for databases over which we design our framework: *trusted site* and *untrusted site*. The *trusted site* is the point of access of the database clients. This can either be independent personal computers or an on-premise trusted cluster controlled by the data owner.[3] The *untrusted site* is where the bulk of data processing is made. One or more cloud providers can play this role, where data is not controlled by the data owner at all times, and where the existence of security vulnerabilities must be considered.

A. CryptoWorkers and CryptoBoxes

Our framework extends NoSQL databases security mechanisms with CryptoWorkers, which abstract the integration of cryptographic schemes on the system. CryptoWorkers reside on the trusted and untrusted site, and provide a privacy-aware NoSQL API for simple integration with NoSQL database architectures. Each database request that is handled by a CryptoWorker is converted into an operation with the same semantics but with additional security guarantees. Depending on the defined security guarantee and on the cryptographic primitive implementation, requests will be translated differently. However, the translation process is abstracted in three operations:

- Encode - is executed on the trusted site, where a query is protected before transmission. For instance, encoding a *Put* with standard encryption will simply require for the data to be encrypted before being sent to the untrusted site. However, complex mechanisms for obfuscation can also be implemented, where an encoding of a *Put* operation translates into several encoded insertion requests.
- Process - is executed on the untrusted site, where an encoded query is processed, and some data is returned. This is trivial for some techniques, but this operation enables our framework to support additional cryptographic mechanisms that require computation over ciphertext.
- Decode - is executed on the trusted site, where a plaintext query response is generated from the ciphertext NoSQL database result.

These operations are supported by modular and easily interchangeable components, CryptoBoxes. These components resort to libraries containing standalone cryptographic implementations. More concretely, supporting a specific security technique in our architecture requires a developer to provide an implementation of a CryptoBox, where the most granular cryptographic operations will be performed, and an implementation of a CryptoWorker, which will have NoSQL context, and employ the CryptoBox to appropriately capture the NoSQL API. This gives us modularity on two different levels: i.) on the security primitive, by allowing extensible deployment configurations with different data sets protected with different techniques (CryptoBoxes), co-existing within the same system, and ii.) on the level of the actual implementation, where if one is interested in upgrading an existing technique with some state-of-the-art advancement, this can be achieved by simply exchanging the implementation on the CryptoBox level, while the original CryptoWorker component can be reused. Note that replacing the CryptoBox, for a specific database column, with another one will require the migration of data belonging to that column. Different encryption schemes are not compatible so it is necessary to generate new protected ciphertexts according to the new CryptoBox encryption algorithm.

B. The life cycle of a Put and Get operation

To exemplify the behavior of our framework, and how this set of components interact in an application scenario, lets assume a NoSQL schema where each key is protected with deterministic encryption, the first column value with standard encryption, and the second column value with deterministic encryption. In this step-by-step description, we will reference Figure 2 for specifying where each computation is taking place.

[3] A trusted infrastructure is achieved via strong access control and security policies, which is necessary for the reliability of cryptographic mechanisms mentioned in the paper.

Upon a *Put* request for a certain key-value pair (1), the CryptoWorker module intercepts the request (2) and executes its Encode operation. This will encrypt the key-value pair with the appropriate techniques, by resorting to the various CryptoBoxes (3). Afterwards, the secure *Put* request will be forwarded to the NoSQL backend, where the Process operation will simply store the data at the untrusted site (4). No operations are necessary to Decode a *Put* operation.

A *Get* request for a specific key (1) would go through the CryptoWorker (2) to execute the according Encode operation. This encrypts the key with the same technique, thus resorting to the associated deterministic encryption CryptoBox (3). This encoded *Get* operation is now sent to the NoSQL backend (4), that executes the query, which is a simple and unmodified operation to recover the value associated to the encrypted key. The data is returned to the trusted site CryptoWorker (4), for execution of Decode. This again resorts to the CryptoBox to decrypt the key-value elements to their plaintext values (3), and reply to the original client request with the result (2,1).

C. Remote processing

The encryption techniques implemented in the experimental part of this paper only require CryptoWorker processing at the trusted infrastructure (Encode and Decode). However, for our framework to be extensible towards optimizations or other techniques discussed in the literature, such as searchable encryption [11], one must be able to employ additional structures (such as protected indexes) and perform some additional calculations at the untrusted site. This is the motivation to use the CryptoWorker module deployed at the untrusted deployment for the aforementioned Process operation, which is sufficiently generic to capture complex mechanisms for computation over encrypted data. In fact, our design can be easily extended with other state-of-the art cryptographic techniques.

To exemplify, consider a single-word search query over data protected with searchable encryption (1). This primitive makes use of *tokens*, which are used exclusively to process some search query over encrypted data. The trusted site CryptoWorker intercepts the request (2) and executes the Encode operation to produce a token. This token is then sent to the untrusted site CryptoWorker (4 a) to execute Process. The remote process must now access a searchable encryption CryptoBox to compute the token over stored data and produce a query response (4b and 4c). This response is then returned to the trusted site CryptoWorker for Decode to be executed. Data is then decrypted according to the respective CryptoBox (3) and the reply is provided to the client (2,1).

In terms of scalability and availability, the techniques discussed in this paper do not affect the sharding and replication design of the NoSQL backend. However, in order to support several NoSQL clients accessing the same data, it is necessary to have a key management service so that all CryptoWorkers at the trusted site have access to the necessary keys for encoding and decoding the data in each query.

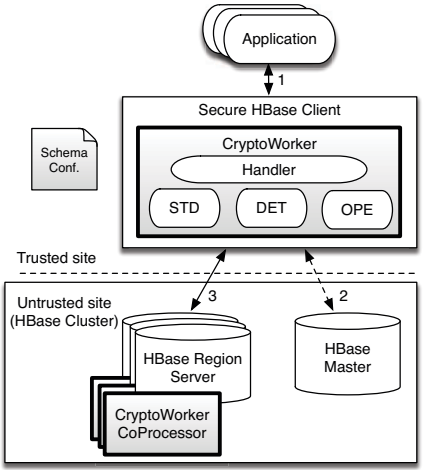


Fig. 3: Implementation of SAFENOSQL resorting to HBase.

V. IMPLEMENTATION

The SAFENOSQL prototype is implemented on top of Apache HBase, a distributed, scalable and open-source non-relational database [9]. Inspired by Google's BigTable, HBase tables are multi-dimensional sorted maps, similar to the NoSQL logical table representation discussed in the previous section. Row keys are associated with an unbounded and dynamic number of qualifiers (columns) that are grouped into column families (groups of columns). Each value on a table is uniquely identified by a column family's name and qualifier's name. A table can be horizontally partitioned into several Regions, each holding a subset of the rows for that table. This partitioning scheme is transparent for the database applications and is a fundamental characteristic that makes HBase highly scalable.

Figure 3 depicts how SAFENOSQL is instantiated over HBase. At the trusted site, applications resort to the *Secure HBase client* component, a modified HBase client that exports the original HBase API (1) with privacy guarantees. Secure requests are forward to the HBase cluster composed by an HBase Master and several Region Servers. The client contacts the Master component when it needs to locate the Region Server(s) holding the Region(s) that serves the rows for that request (2). Once the Region server is located, client requests are made directly to the Region Server (3) that handles the desired data to be processed and retrieved. The HBase Master may be deployed in a primary/secondary replication mode to ensure high availability, while the Regions are replicated using HDFS to ensure data availability.

The HBase client API is similar to the one described in Section IV (*i.e.*, Put/Update, Get, Delete, Scan). Additionally, Scan operations support Filters for qualifiers (SingleColumnValueFilter). Namely, it is possible to issue a Scan for the entire table, or for a range of rows, and filter at the HBase backend the desired qualifier values by equality or range.

In Figure 3 we also depict the components of SAFENOSQL prototype. The gray boxes present the novel components that were added to HBase in order to have a secure NoSQL

implementation. Our CryptoWorker implementation extends the original HBase client implementation as a middleware component and provides confidentiality guarantees on NoSQL operations. Through a configuration file it is possible to define the encryption technique that will be used for keys and for qualifiers. Since qualifiers can be grouped into column families, our configuration file allows defining the same encryption scheme for all the qualifiers belonging to a specific column family. As discussed in Section IV, the CryptoWorker component is composed by a module that intercepts NoSQL requests and resorts to the appropriate CryptoBoxes to encode sensitive data, process it (if necessary), and decode encrypted data according to the configuration file schema.

Our prototype currently contemplates three distinct CryptoBoxes. Standard encryption (STD) CryptoBox relies on OpenSSL [26] cryptographic library. Deterministic encryption (DET) is implemented in accordance to the construction described in [22]. Finally, the OPE CryptoBox is implemented following the design of [24] and it relies on OpenSSL and MPFR [27], a multiple-precision floating-point library.

Although the three supported cryptographic techniques do not require additional computation at the HBase backend, other techniques such as searchable encryption may require keeping a secure index and having additional processing over encrypted data at the backend. In HBase, supporting these novel techniques is attainable without changing the core implementation of HBase backend components, which would increase significantly the implementation effort of our prototype. For this, one can resort to HBase co-processors. These can be seen as plugins specifying additional computation that must be done at each Region Server when specific NoSQL queries are executed. For instance, if a Get operation requires consulting the secure index and doing some additional computation, it is possible to deploy a CryptoWorker as a co-processor that, for each Get operation, will do the necessary steps to provide the correct results for that query. Our current prototype is designed to avoid changing HBase's core implementation. In fact, supporting the three techniques discussed above did not require any line of code at the HBase backend's core to be changed, while for the HBase client's code we only added approximately 2100 lines to integrate our CryptoWorker implementation. This approach has the additional benefits of compatibility with evolving versions of HBase, as well as easier transition from HBase to other NoSQL databases.

Encrypted data retrieved with Get and Scan operations must be decoded to plaintext at the trusted site CryptoWorker before being forwarded to the application. As shown in our experimental evaluation discussed in Section VI, decoding information encrypted with OPE has a significant penalty in the latency and throughput of HBase operations. As such, we propose an optimization that trades additional storage space for a considerable performance improvement. In our system, every column qualifier encrypted with OPE will be accompanied by the same value protected with STD. Then, when a value protected with OPE must be retrieved to the client, instead of decoding the OPE encryption, the CryptoWorker module decodes the value protected with STD instead, which is considerably faster. For instance, decoding a 14 bytes length ciphertext with OPE takes $567.434\mu s$ and with STD takes $5.884\mu s$. Moreover, for a 256 bytes ciphertext, OPE takes $2.861s$ to decode while STD takes $8.028\mu s$. This optimization also contemplates the storage of data encrypted with OPE, so filtering operations such as equality or range queries are still supported.

VI. Experimental Evaluation

This section presents the experimental evaluation of the SafeNoSQL prototype. First, we describe the experimental setup and workloads, and then we discuss the results obtained.

A. Experimental Setup, Benchmark and Workloads

Experiments ran on a cluster composed of 6 servers equipped with an Intel i3 CPU with four cores at 3.7 GHz, 8GB of RAM and a 128 GB SSD disk interconnected with a gigabit switch. The HBase cluster was deployed on 5 nodes. The HBase master ran on an isolated server and four HBase Region Servers were deployed on the remaining servers. Each RegionServer was configured with 4GB of *heap size*, with 55% of this space assigned to the *memstore* and 10% to the *block cache*.

The remaining server was used as the database client. For this client, our experiments resort to the Yahoo! Cloud Serving Benchmark (YCSB), a well-known benchmark suited for NoSQL data stores, including HBase, that provides realistic cloud-based workloads [28]. Each workload can be customized by defining the operations that are going to be performed at the NoSQL data store *i.e.,* Insert (HBase's Put), Read (HBase's Get), Update (HBase's Put), Read-Modify-Write (HBase's Get-Update), Delete (HBase's Delete) and Scan (HBase's Scan) operations. Also, other parameters such as the operation's data access pattern, the ratio for each operation, and the benchmark's execution time can be defined by the user.

Although YCSB operations allow testing most of the HBase's API, these do not contemplate equality and range filter operations over column qualifiers (QualifierFilter or SingleColumnValueFilter as typically known in HBase). For this reason, we extended YCSB to include such operations on the benchmarking suite. Each experiment discussed along this section ran for 20 minutes with an extra 3 minute period of ramp-up time for a database pre-populated with 10 million rows. Also, each experiment was repeated 5 times to calculate the mean and standard deviation across distinct runs.

Two table schemas were designed specifically for the experiments in order to test our prototype in a more realistic setup. With this approach it is possible to extract meaningful results and conclusions, for this and other similar use-cases, about the performance overhead induced by the different cryptographic techniques. In more detail, as the healthcare sector deals with different types of sensitive information and it must comply with several legal regulations concerning data privacy, we chose to explore such use-case.

The first table schema, *Patients*, is shown in Table I and contemplates a subset of the data typically found on an Hospital

Key	Identification						Contacts		Obs	App
	MainID	*Surname*	*Name*	*Birth*	*Nationality*	*CivilID*	*Address*	*Contact*	*Observations*	*[1-*]*
8	64	64	64	14	4	9	256	13	1024	8
DET	DET	STD	STD	STD	STD	STD	STD	STD	STD	STD

TABLE I: NoSQL table schema for Hospital Patients.

Key	Physician	Patient	Appointment				Institution	
	PhysicianID	*PatientID*	*Date*	*Date-STD*	*Type*	*Obs*	*Name*	*Address*
8	16	16	14	14	64	1024	128	256
DET	DET	STD	OPE	STD	STD	STD	STD	STD

TABLE II: NoSQL table schema for Hospital Appointments.

database table that stores personal information from patients. The NoSQL table stores an application generated key (Patient ID) for each patient, while each row is composed by a set of column families (Identification, Contacts, Observations, Appointments) that group distinct column qualifiers holding patient's information (MainID, Surname, Name, Birth date, Nationality, CivilID. Address, Contact, Observations). The Appointments (App) column family can have a dynamic number of qualifiers, each indicating the ID of the patient's medical appointments (Appointment ID). Table I also shows the size in bytes of each column qualifier and a proposal for the encryption techniques to be applied on each field in order to ensure privacy of personal information while still allowing querying such data. Briefly, most information related to a given Patient is encrypted using standard encryption (STD). In order to be able to retrieve the information of a specific patient given her first name, last name and date of birth, the column qualifier *MainID* contains such information encrypted with deterministic encryption (DET). This specific set of characteristics is often used in medical systems to identify the patients. The storage space for a single row in plaintext is 1526 bytes and for the proposed secure schema is 1888 bytes. This is an important aspect to have in account as it represents another tradeoff on data storage and bandwidth for secure NoSQL databases.

The second table schema, *Appointments*, is shown in Table II and stores the Hospital appointments between a given physician and a patient. The key (Appointment ID) is an unique identifier generated by the application and each row is composed by a set of column families (Physician, Patient, Appointment, Institution) grouping distinct column qualifiers holding relevant information for the appointment (Physician ID, Patient ID, Appointment Date, Type of appointment, Observations, Institution Name and Address). In this table, we propose a possible schema where the column qualifiers *PhysicianID* is encrypted with DET, the *Date* of the appointment is encrypted with OPE, while the remaining column qualifiers are encrypted with STD. This design allows, for instance, retrieving all appointments of a given physician for a given time period. According to the optimization discussed in the previous section, a novel column qualifier *Date-STD* was created in order to reduce the overhead of decoding operations when OPE is being used. The storage space for a single row in plaintext is 1552 bytes and for the proposed secure schema is 1756 bytes.

B. Micro-experiments

Micro-experiments were performed for the *Appointments* schema to understand the isolated impact of the cryptographic techniques supported by SAFENOSQL prototype. Experiments ran isolated YCSB operations, while all table qualifiers were stored in plaintext and only the row keys were protected with different cryptographic techniques (STD, DET, OPE). This approach was used to provide a controlled testing scenario where the only factor changing is the way a single data value is encrypted. Such approach, allows a more precise comparison of the overhead of each technique and a comparison with a baseline HBase deployment without encryption. Benchmark operations write/read the entire row, while Scan and Filter operations were issued with a random starting row key. For the Filter operations a pre-defined value stored in the database was used as the value to be searched. We also varied the data access pattern of the benchmark by running all experiments with both the Zipfian (Hotspot) and Uniform distributions.

Figure 4 and Figure 5 and Table III show the throughput and latency values for the micro-experiments. For STD encryption we only show values for the write tests, while for DET encryption we do not show values for range queries. As explained in Section III, STD encryption applicability is limited to the protection of data in scenarios where no operations are to be performed over it, such as data transmission or storage. Similarly, for DET encryption, as the order of the plaintext is not enforced on the resulting cyphertext, this is not a valid technique for retrieving data whose value is between a certain range. Performing such queries over data encrypted in this fashion would require a full database retrieval to the client premises and subsequent decryption, which is unfeasible for any realistic database deployment.

Regarding the results, as expected, performing insertions and queries over data protected with STD and DET encryption has a small overhead when compared with the analogous operations done over plaintext. On the other hand, the overhead is significant for the OPE technique mainly due to the time spent encoding and decoding the plaintext and ciphertexts, respectively. Nevertheless, this overhead value is according to the expected value for our OPE CryptoBox implementation [24]. This means, that such value can be improved in the future with more efficient implementations of OPE or by relying on more efficient cryptographic techniques such as searchable encryption [13]. The only exception is shown in the Qualifier Equality Filter (QEF), where the performance of OPE is similar to the baseline HBase deployment. In this equality filter most of the overhead is due to the search of the correct value at the HBase backend and a single set of key-value pairs is returned and decrypted. On the other hand, for both Scan and Qualifier Range Filter (QRF) operations the set of returned rows to the client is significantly higher (at least one order of magnitude) and the overhead of decoding key encrypted with OPE keys is

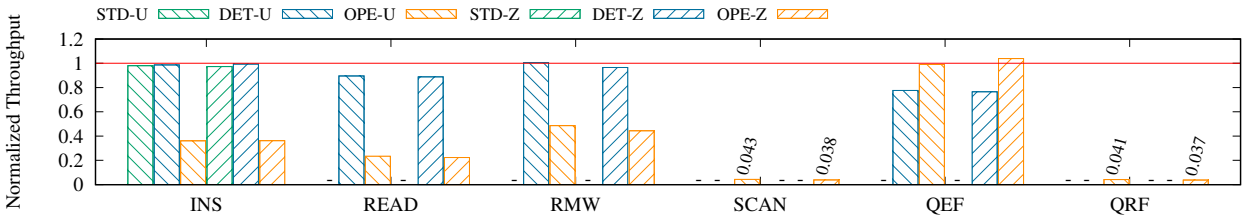


Fig. 4: Normalized throughput for the micro-experiments. HBase baseline corresponds to value 1. For each YCSB operation (Insert (INS), Read, Read-Modify-Write (RMW), SCAN, Qualifier Equality Filter (QEF), Qualifier Range Filter (QRF)) the throughput is shown for STD, DET, and OPE encryption techniques and both Uniform (U) and Zipfian (Z) access patterns.

		INS		READ		RMW		SCAN		QEF		QRF	
		U	Z	U	Z	U	Z	U	Z	U	Z	U	Z
BAS	Thr (ops/s)	333.835 ± 7.107	334.255 ±5.291	532.071 ±10.689	614.828 ±11.542	183.432 ±7.082	204.391 ±4.873	69.576 ±7.331	79.095 ±7.301	0.043 ±0.006	0.043 ±0.004	70.234 ±11.020	78.537 ±11.519
	Lat (ms)	2.033 ± 0.064	2.928 ±0.047	1.874 ±0.038	1.624 ±0.031	5.582 ±0.240	5.105 ±0.165	14.447 ±1.463	12.666 ±1.134	26675.757 ±3234.64	23404.915 ±2063.02	14.502 ±2.440	12.912 ± 2.019
STD	Thr (ops/s)	327.228 ± 5.783	325.061 ±5.746	-	-	-	-	-	-	-	-	-	-
	Lat (ms)	2.993 ± 0.054	3.015 ±0.056	-	-	-	-	-	-	-	-	-	-
DET	Thr (ops/s)	329.315 ± 4.954	331.907 ±7.559	476.395 ±5.631	546.04 ±22.883	184.304 ±6.208	197.209 ±8.222	-	-	0.033 ±0.002	0.032 ±0.002	-	-
	Lat (ms)	2.973 ± 0.046	2.951 ±0.068	2.093 ±0.025	1.828 ±0.079	5.630 ±0.137	5.313 ±0.156	-	-	29821.832 ±1850.85	30228.441 ±1951.79	-	-
OPE	Thr (ops/s)	120.438 ± 0.862	121.082 ±0.927	124.83 ±1.382	136.988 ±0.440	89.02 ±1.892	90.579 ±1.641	2.994 ±0.026	2.998 ±0.039	0.042 ±0.006	0.045 ±0.002	2.893 ±0.049	2.906 ±0.092
	Lat (ms)	8.234 ± 0.061	8.191 ±0.064	8.001 ±0.089	7.288 ±0.023	11.998 ±0.798	12.056 ±0.449	333.716 ±2.952	333.176 ±4.441	23816.373 ±3787.09	22355.588 ±1134.39	345.251 ±5.888	343.964 ±11.362

TABLE III: Throughput (Thr) and latency (Lat) results for the micro-experiments. For YCSB operation (Insert (INS), Read, Read-Modify-Write (RMW), Scan, Qualifier Equality Filter (QEF), Qualifier Range Filter (QRF)) values are shown for baseline (BAS) HBase solution and for STD, DET and OPE schemes, including both Uniform (U) and Zipfian (Z) access patterns.

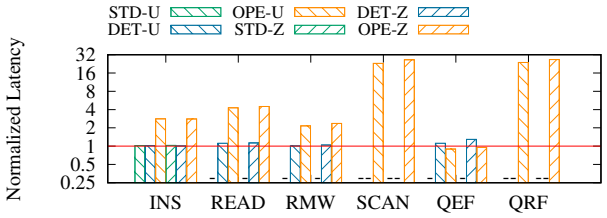


Fig. 5: Normalized latency for the micro-experiments. HBase baseline results correspond to value 1. For each YCSB operation (Insert (INS), Read, Read-Modify-Write (RMW), Scan, Qualifier Equality Filter (QEF), Qualifier Range Filter (QRF)) the latency is shown for STD, DET, and OPE encryption schemes and both Uniform (U) and Zipfian (Z) access patterns.

Workload	*Insert*	*Update*	*RMW*	*Read*	*Scan*	*QEF*	*QRF*
A	-	50%	-	50%	-	-	-
B	-	5%	-	95%	-	-	-
E-1	5%	-	-	-	75%	10%	10%
E-2	5%	-	-	-	75%	20%	-
F	-	-	50%	50%	-	-	-
G	50%	-	15%	10%	-	10%	10%
H	10%	-	45%	30%	-	15%	-

TABLE IV: Operations percentage per YCSB test.

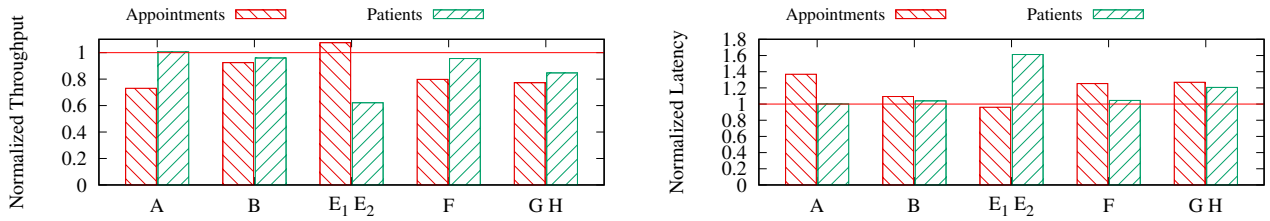
highly noticeable. As expected, the Zipfian distribution provides higher throughput for most tests due to the hotspot distribution that leverages HBase caching mechanisms.

The previous results show that for each cryptographic technique one can expect different tradeoffs in terms of database performance and supported functionality. This way, on a real deployment scenario, a single technique is not the best approach for protecting all the information stored in the same database. To understand the impact of combining different techniques we next focus on macro-experiments where multiple techniques are combined to provide a fully-functional database in a more realistic experimental setup.

C. Macro-experiments

The macro-experiments were devised to assess the impact of combining different cryptographic techniques for the *Patients* and *Appointments* schemas. As discussed in Subsection VI-A, we protected the key and column qualifiers with the cryptographic techniques described in Table I and Table II in order to ensure that useful queries can still run over encrypted data while protecting sensitive information.

To evaluate the performance of our solution, we compared it against a baseline HBase deployment storing everything in plaintext, and ran a set of YCSB tests. Table IV shows the ratio of operations performed for each test. Tests (*A* to *F*) are standard configurations already provided in YCSB and typically used in previous work [29]. We modified test *E* and created two variants, E_1 and E_2. In the first we leverage Qualifier Equality Filters (QEF) and Range Filters (QRF) for *Date* values of the *Appointments* schema. In the latter, we leverage Equality Filters (QEF) for the *MainID* of the *Patients* schema. Additionally, the tests *G* and *H* were defined specifically for our tests, in order to reproduce a typical query environment for our healthcare use-case. The *Appointments* schema (test *G*) has a significant



(a) (b)

Fig. 6: Normalized throughput (a) and latency (b) results of macro-experiments (HBase baseline results correspond to value 1).

SC.	System	Metric	A	B	E_1	E_2	F	G	H
AP	BAS	Thr (ops/s)	400.36 ± 16.191	421.183 ± 11.920	6.370 ± 0.430	-	277.306 ± 9.689	8.567 ± 1.897	-
		Lat (ms)	2.502 ± 0.097	2.377 ± 0.081	157.703 ± 10.578	-	3.10 ± 0.123	125.278 ± 39.02	
	SAFENOSQL	Thr (ops/s)	292.396 ± 10.986	389.056 ± 41.218	6.843 ± 1.352	-	221.253 ± 3.725	6.617 ± 1.595	-
		Lat (ms)	3.425 ± 0.127	2.597 ± 0.255	151.533 ± 27.342	-	4.521 ± 0.078	159.074 ± 33.503	-
PA	BAS	Thr (ops/s)	331.014 ± 10.075	326.441 ± 23.309	-	0.089 ± 0.003	221.154 ± 11.286	-	0.066 ± 0.002
		Lat (ms)	3.024 ± 0.092	3.078 ± 0.210	-	11,284.588±395.137	4.534 ± 0.232	-	15,219.327±533.208
	SAFENOSQL	Thr (ops/s)	332.467 ± 23.298	313.042 ± 11.189	-	0.055 ± 0.003	211.308 ± 8.300	-	0.056 ± 0.008
		Lat (ms)	3.022 ± 0.195	3.199 ± 0.114	-	18,196.153±925.209	4.740 ± 0.186	-	18,354.071±2 801.68

TABLE V: *Appointments* (AP) and *Patients* (PA) schemas macro-experiments latency (Lat) and throughput (Thr) results for the baseline (BAS) HBase and SAFENOSQL prototype.

number of insertions and a lower number of search queries. The *Patients* schema (test *H*) has a smaller percentage of insertions and a higher percentage of search queries. All experiments ran with the *zipfian* access distribution.

In test *G*, appointments *Date* was populated with a random value between 2015-2020. Filter operations (QEF and QRF) ran for values comprehended between this range and performed a full table Scan to search for such values. In test *H*, Patient's *MainID* were also filtered with a full table Scan to find the desired values. For this experiment we ensured that only values stored at the database are searched.

Figure 6a, Figure 6b and Table V show the throughput and latency values for the macro-experiments. As expected, the *Patients* schema provides less overhead in most tests since it does not resort to OPE for protecting sensitive information. In average, when compared to the baseline HBase system, the *Patients* tests present an overhead of 12.29% across the different YCSB tests. As the worst-case scenario an overhead of 37.9% is visible in test E_2. On the other hand, the Appointments schema presents an average performance loss of 14.03%. In workload *A* the overhead reaches approximately 27%.

For the workloads tests *G* and *H*, when compared to the baseline HBase systems, overheads of 22.76% and 15.42% are observable, respectively. A considerable part of the overhead in workload *G* is due to OPE cipher's performance, since the insert proportion corresponds to 50% of the total operations. In workloads *H* and E_2, most of the performance overhead is due to performing a range scan over keys protected with DET encryption. As this encryption technique does not preserve the order of the plaintext, a full table scan must be done to search for the keys between the requests range of values.

Interesting results are shown for workload E_1 where the throughput and latency values are similar to the baseline. This happens due to the extra qualifier *Date-STD* protected with STD encryption that is stored along with the OPE *Date* qualifier. This optimization was described in Section V and allows reducing the overhead of decoding operations with the OPE CryptoBox.

The previous results show that combining different cryptographic techniques is key for supporting a wide range of applications workloads, with different NoSQL operations, while providing acceptable performance. Also, there is still some space for improvement, which justifies the importance of a flexible framework such as SAFENOSQL that will allow easily incorporating novel cryptographic techniques, with different tradeoffs in terms of performance, security and functionality.

VII. CONCLUSION

This paper presents SAFENOSQL, a secure framework for NoSQL databases aiming at a modular and flexible design that can easily be extended with state-of-the-art privacy-preserving computation techniques. In more detail, we propose an architecture that resorts to novel CryptoWorker and CryptoBox components in order to be generic for most NoSQL databases and to easily accommodate different widely used cryptographic techniques. We have implemented a prototype of our framework based on Apache HBase and conducted an extensive set of micro- and macro-experiments with various realistic workloads to assess the practicality of our solution.

The results show that by combining different cryptographic techniques, it is possible to have a practical solution that balances the desired functionality, performance and security for different applications. In average, when compared with a baseline HBase deployment without any data privacy guarantees, SAFENOSQL prototype introduces less than 15% of performance overhead across the different realistic macro-workloads tested. Also, the results show that there is still space for improvement, for instance, by introducing other cryptographic techniques in SAFENOSQL that can overcome the current performance penalty of OPE. This is a key conclusion that shows the importance of designing a flexible

architecture where the addition of novel encryption techniques can be done in a straightforward fashion.

VIII. Acknowledgements

This research was supported by the European Union's Horizon 2020 - The EU Framework Programme for Research and Innovation 2014-2020, under grant agreement No. 653884. The authors João Paulo, Ricardo Macedo, Rogério Pontes and Rui Oliveira were financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme within project «POCI-01-0145-FEDER-006961», and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia as part of project «UID/EEA/50014/2013». Bernardo Portela was funded by project "NanoSTIMA: Macro-to-Nano Human Sensing: Towards Integrated Multi-modal Health Monitoring and Analytics/NORTE-01-0145-FEDER-000016" and Tiago Oliveira was funded by project "TEC4Growth - Pervasive Intelligence, Enhancers and Proofs of Concept with Industrial Impact/NORTE-01-0145-FEDER-000020", both financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF).

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010.

[2] H. Takabi, J. B. D. Joshi, and G. J. Ahn, "Security and privacy challenges in cloud computing environments," *IEEE Security Privacy*, 2010.

[3] F. P. Miller, A. F. Vandome, and J. McBrewster, "Advanced encryption standard," 2009.

[4] A. Boldyreva, S. Fehr, and A. O'Neill, "On notions of security for deterministic encryption, and efficient constructions without random oracles," in *Annual International Cryptology Conference*, 2008, pp. 335–359.

[5] A. Boldyreva, N. Chenette, and A. O'Neill, "Order-preserving encryption revisited: Improved security analysis and alternative solutions," in *Annual Cryptology Conference*, 2011, pp. 578–595.

[6] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: Protecting confidentiality with encrypted query processing," in *Symposium on Operating Systems Principles*, 2011, pp. 85–100.

[7] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich, "Processing analytical queries over encrypted data," *Proc. VLDB Endow.*, 2013.

[8] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[9] "HBase," Available: https://hbase.apache.org/.

[10] E. Pattuk, M. Kantarcioglu, V. Khadilkar, H. Ulusoy, and S. Mehrotra, "Bigsecret: A secure data management framework for key-value stores," in *International Conference on Cloud Computing*, 2013.

[11] X. Yuan, X. Wang, C. Wang, C. Qian, and J. Lin, "Building an encrypted, distributed, and searchable key-value store," in *Asia Conference on Computer and Communications Security*, 2016, pp. 547–558.

[12] J. P. Rogerio Pontes, Francisco Maia and R. Vilaca, "SafeRegions: Performance Evaluation of Multi-party Protocols on HBase," in *Symposium on Reliable Distributed Systems Workshops*, 2016, pp. 31–36.

[13] R. Poddar, T. Boelter, and R. A. Popa, "Arx: A strongly encrypted database system," *Cryptology ePrint Archive*, vol. 2016, p. 591, 2016.

[14] J. Li, Z. Liu, X. Chen, F. Xhafa, X. Tan, and D. S. Wong, "L-encdb: A lightweight framework for privacy-preserving data queries in cloud computing," *Knowl.-Based Syst.*, vol. 79, pp. 18–26, 2015.

[15] A. Shamir, "How to share a secret," *Commun. ACM*, pp. 612–613, 1979.

[16] T. Xiang, X. Li, F. Chen, S. Guo, and Y. Yang, ""processing secure, verifiable and efficient SQL over outsourced database"," *Inf. Sciences*, vol. 348, pp. 163 – 178, 2016.

[17] D. Bogdanov, M. Niitsoo, T. Toft, and J. Willemson, "High-performance secure multi-party computation for data mining applications," *International Journal of Information Security*, pp. 403–418, 2012.

[18] R. Pontes, M. Pinto, M. Barbosa, R. Vilaca, M. Matos, and R. Oliveira, "Performance trade-offs on a secure multi-party relational database," 2017.

[19] "MySQL enterprise transparent data encryption," https://www.mysql.com/products/enterprise/tde.html.

[20] "ORACLE transparent data encryption," http://www.oracle.com/technetwork/database/options/advanced-security/index-099011.html.

[21] J. Katz and Y. Lindell, *Introduction to modern cryptography*. CRC press, 2014.

[22] P. Rogaway and T. Shrimpton, "Deterministic authenticated-encryption," in *Advances in Cryptology–EUROCRYPT*, vol. 6, 2007.

[23] J. R. Douceur, A. Adya, W. J. Bolosky, P. Simon, and M. Theimer, "Reclaiming space from duplicate files in a serverless distributed file system," in *Int. Conference on Distributed Computing Systems*, 2002, pp. 617–624.

[24] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill, "Order-preserving symmetric encryption," in *Int. Conference on Advances in Cryptology: The Theory and Applications of Cryptographic Techniques*, 2009, pp. 224–241.

[25] M. Naveed, S. Kamara, and C. V. Wright, "Inference attacks on property-preserving encrypted databases," in *Conf. on Computer and Communications Security*, 2015.

[26] "OpenSSL." [Online]. Available: http://www.openssl.org/

[27] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "Mpfr: A multiple-precision binary floating-point library with correct rounding," *ACM Transactions on Mathematical Software*, vol. 33, no. 2, p. 13, 2007.

[28] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Symp. on Cloud Computing*, 2010, pp. 143–154.

[29] R. Escriva, B. Wong, and E. G. Sirer, "Hyperdex: A distributed, searchable key-value store," in *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2012, pp. 25–36.