Processing Encrypted and Compressed Time-Series Data

Matúš Harvan*, Samuel Kimoto†, Thomas Locher‡, Yvonne Anne Pignolet‡, Johannes Schneider§
* Enovos Luxembourg S.A., matus.harvan@enovos.eu
† Open Systems, ski@open.ch
‡ ABB Corporate Research, {thomas.locher,yvonne-anne.pignolet}@ch.abb.com
§ University of Liechtenstein, johannes.schneider@uni.li

***Abstract*—Numerous applications, e.g., in the industrial sector, produce large amounts of time-series data, which must be stored and made available for distributed processing. While outsourcing data storage and processing to third-party service providers offers many benefits, it raises data privacy issues. In light of this problem, techniques have been proposed to share only encrypted data with the remote service provider, yet the capability to run meaningful queries over the data is preserved. However, time-series data is typically compressed at the server to save space, which is not easily possible when dealing with encrypted data. Moreover, data must be compressed in such a way that queries can still be executed efficiently.**

As a first step in this direction, we present an approach that preserves data privacy, enables compression at the server, and supports querying of the stored data. Our evaluation using real-world time-series data shows that our compression mechanism can reduce the required space drastically. Moreover, the median running time of all considered queries increases marginally, implying that compression can be introduced without sacrificing performance of query execution.

I. Introduction

Time-series data, i.e., periodic time-stamped measurement data, plays an important role in various applications for data analysis and forecasting. As data is produced steadily, with a periodicity in the sub-second range for many industrial applications, providing the necessary facilities to store and process it can become a burden. A cost-effective and convenient approach is to outsource data storage and processing to a third-party service provider. Thus, the devices producing data are integrated into a distributed system comprising storage and processing nodes, which are offered as a service. However, as the service provider has full access to all data, this approach is encumbered by privacy concerns: companies from various industries are reluctant to provide access to their data since it can contain confidential information, e.g., about the state of operation, or it may reveal sensitive information concerning the devices producing the data, e.g., parameters of control algorithms.

Naturally, simply encrypting all data using standard encryption tools before the data leaves the data owner's premises is not a satisfactory solution as it prevents any meaningful computation at the service provider. Several techniques have been proposed to carry out certain computations without revealing the data, such as *homomorphic encryption* or *multi-party computation*, and systems have been built for this purpose. Noteworthy examples are CryptDB [15] and MONOMI [18], which enable data processing in a privacy-preserving manner, at the cost of a computational overhead that is shown to be small in benchmark tests. These systems offer features that are highly useful for time-series analysis, in particular deterministic and order-preserving encryption (OPE), which enable *range queries*.

However, privacy-preserving systems proposed so far fall short of providing an important feature for time-series data management: *compression* at the server side. As mentioned before, data volumes are usually high, and the potential for compression is large as well because the generated time-series data often contains many repetitions for a wide range of (industrial) applications. Therefore, a practical system must support compression, otherwise the space overhead compared to a system without privacy protection becomes exceedingly large. Moreover, the addition of compression must not result in a large increase of query execution times. Thus, cryptographic techniques must be combined carefully to preserve computational efficiency and data privacy while offering compression.

The **contributions** of our work are the following. We present an approach to store encrypted and compressed time-series data and execute queries over this data efficiently. To this end, we employ a novel combination of a *partially homomorphic K-probabilistic encryption scheme* with an *order preserving module*. Partially homomorphic encryption and order preservation are required to run queries over the encrypted data. We introduce the notion of *K-probabilistic encryption*, which has the desired property of protecting even low-entropy data while enabling data compression. A noteworthy feature of our approach is that clients can encrypt and prepare data and queries in a manner that enables the server to use standard techniques for compression and query processing. We describe an implementation of a practical system reconciling the somewhat conflicting requirements concerning confidentiality and space and computational complexity. Furthermore, we provide a thorough evaluation, focusing both on security and on the compression rate and query execution time. In order to obtain practically relevant results, we used real time-series data from a solar plant and a wind farm for our experiments.

The paper is structured as follows. §II describes the model of the system and the attacker model. Our approach is presented in §III. The security and performance evaluation are given in §IV and §V, respectively. Related work is summarized in §VI, and §VII concludes the paper.

Operator	Application
$<, >$	BETWEEN, ORDER BY, MIN, MAX, SORT
$=$	IN, DISTINCT, equality JOINS, GROUP BY
`count`	Count operator
`sum, mean, var`	Summation, mean and variance operators for numerical data

TABLE I: Table of supported query operators.

II. Model

For the sake of simplicity, we consider a simple distributed setup consisting of a single client communicating with a remote server. Extending the scenario to multiple clients and servers is straightforward. The client produces time-series data to be inserted in the database at the server and initiates queries to be answered by the server.

Time-series data are modeled as a stream d of data items $d_1, d_2, \ldots$, where each d_i is a feature vector. The value of the j^{th} feature of the i^{th} data item is denoted by $d_{i,j}$. Time-series data often has a relatively small entropy: it contains many repetitions, including multiple repetitions in a row, so-called *runs*, implying that the data is amenable to compression. What is more, only a small fraction of all possible values—e.g., all 32-bit or 64-bit numbers—occurs in the data stream. Our approach targets exactly such data sets, i.e., the data stream must have these properties for our approach to be effective. Real-world examples of such data sets are used in our practical evaluation (see §V).

The database DB supports queries of the following basic form:

$$q := \texttt{SELECT}\ \ F(c)\ \ \texttt{WHERE}\ \ \{d_{i,c} \in DB \mid P(d_{i,c})\},$$

where c is a column (feature) and P is a predicate that returns either true or false. The function F can be the identity operator, returning all values for which the predicate condition is satisfied, or an aggregation operation such as `sum` or `count`. The predicates are built using, e.g., comparisons or range checks. See Table I for a list of the operators that can be used to build queries. Note that mathematical operations are only permitted in the function F and not in the predicates.[1] This is a common limitation (see, e.g., [15]) when processing encrypted data due to the non-determinism in homomorphic encryption schemes: when adding up ciphertexts corresponding to plaintexts v and v' homomorphically, the resulting ciphertext is not always identical to a ciphertext obtained when encrypting $v + v'$. We find that many practically relevant queries can still be carried out despite this restriction. In particular, when using order-preserving encryption, it is possible to determine if values are in a certain range, e.g., using a clause of the form `WHERE` $const_1 < v < const_2$ or comparisons of different values of the form `WHERE` $v < v'$.

Requirement 1 (Query processing): The server must support range queries with aggregation operations.

The server is not fully trusted: We assume a passive attacker in the honest-but-curious model where the server follows the protocols faithfully, i.e., it stores all data and does not tamper with it, and executes queries correctly, but it tries to learn as much as possible about the data. Consequently, data needs to be encrypted before being transmitted to the server. Thus, an attacker may only learn something about the data by observing the queries and the returned (encrypted) results. Such a passive attack model is realistic because a more malicious attacker is more likely to be detected when changing the data or query results.

In order to enable the server to process such queries efficiently and securely, the encryption scheme $S(KG, E. D)$, where KG is the key generation function, E encrypts plaintexts, and D decrypts ciphertexts with the appropriate keys, must have certain properties. Given the fact that the entropy of the data is small, deterministic encryption cannot be used, otherwise an attacker is likely to find plaintext-ciphertext pairs by simply encrypting candidate plaintexts [4]. On the other hand, if probabilistic encryption is used, potential for compression and efficient (range) queries is lost. As a customizable trade-off, we require the encryption scheme to be K-probabilistic, which upper bounds the number of (probabilistically computed) ciphertexts for each plaintext to K. Formally, let C_v denote the set of all ciphertexts that E generates for fixed parameters K, s, and v, then it holds that $|C| = K$ for all v in the plaintext space. Modifying K affects the trade-off between security and compressibility: the larger K, the less the server can learn about the data and the more ciphertexts need to be stored. Note that K can be chosen independently for each user or for each use case. For numerical values, additional properties need to be satisfied. S must be additively homomorphic for the computation of `sum, var, mean`, i.e., $D(C(v) \oplus C(v')) = v + v'$ for plaintexts v, v' and a homomorphic addition operator $\oplus$.

Requirement 2 (Encryption scheme): A suitable K-probabilistic encryption scheme must be additively homomorphic and withstand chosen plaintext attacks.

Furthermore, the server must be able to compare encrypted values, i.e., there must be a total order $\preceq$ on the ciphertexts: For any pair of plaintexts v, v' it must hold that $D(v) \preceq D(v') \Leftrightarrow v \leq v'$. Moreover, merely the order between ciphertexts that a particular client has created can be determined by the server as otherwise an attacker could identify plaintexts by determining the order between the client's and the attacker's encrypted data. For example for numerical data, the attacker could compare the order of its own generated ciphertexts corresponding to the numbers $0, 1, 2, \ldots$ to a ciphertext c encrypted by the client in order to determine the plaintext v that corresponds to c.

Requirement 3 (Order preservation): The server learns the order of ciphertexts if and only if the ciphertexts were created by the same client.

Given this model, the goal is to minimize the space required at the server to store the data, yet the capability to execute queries efficiently must be preserved. Note that simply compressing all data minimizes the required space but query running times would increase tremendously as the entire database would have to be decompressed before queries can be answered. Moreover, even though the server has full access to the database, it must learn as little as possible about the data.

[1] For example, it is not allowed to define a clause of the form `WHERE` $col_1 + col_2 = col_3$.

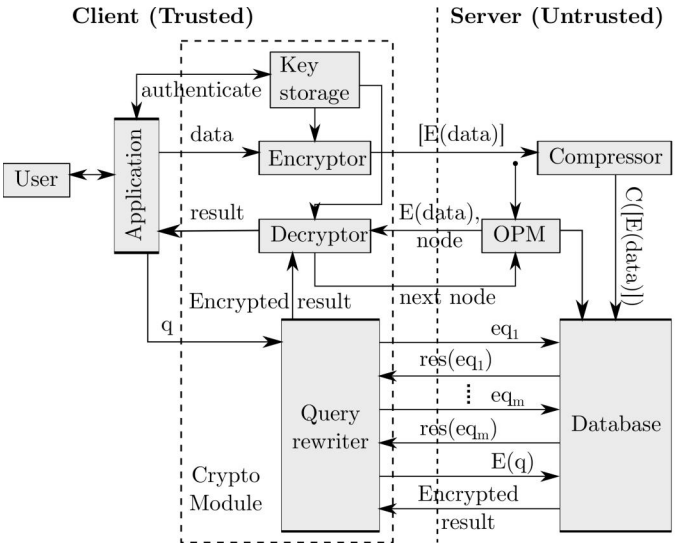

Fig. 1: Overview of components and their interactions.

III. System

In this section, we present our proposed approach to process time-series data efficiently in a privacy-preserving manner. After giving an overview in §III-A, the client and server parts are discussed in §III-B and §III-C, respectively.

A. Overview

The client produces data and initiates queries to be processed by the (untrusted) server. The client part includes the user, the application module, and the crypto module. The application module is used to produce data and initiates queries before sending them to the crypto module. The crypto module operates at the interface between the client and the server, residing at the client side. Its main purpose is to uphold *data confidentiality*. To this end, it encrypts each feature vector before sending it to the server. Moreover, since the server only stores encrypted data, it rewrites the client's queries into queries that the server can process. The user must authenticate to the crypto module to use the system. The server compresses the encrypted data, stores it, and processes the rewritten queries. To this end, it uses an order-preserving module (OPM) that maintains a total order of all ciphertexts known to the server. Figure 1 depicts the components and how they interact with each other. The crypto module constitutes our main contribution. In the remainder of this section, the most important components and mechanisms are described in more detail.

B. Client Crypto Module

The crypto module resides on the client side and consists of four components: key storage, encryptor, decryptor, and query rewriter. We will now discuss their functionality.

Authorization. The crypto module ensures that only authorized users can use the system. To this end, the key storage component authenticates the user and generates and stores the encryption and decryption keys for that particular user. User authentication works as follows: The user (at the client) sends its credentials to the client crypto module, and the crypto module verifies them. If the user is authorized, it loads the keys associated with the user's credentials.

Encryption Scheme. We assume that the client crypto module has the means to create cryptographically strong key material and random bit strings. The crypto module makes use of a pseudo-random number generator G and an IND-CPA-secure[2] probabilistic encryption scheme $S(KG, E, D)$. A probabilistic encryption scheme is probabilistic in the sense that the encryption function E takes a random component r as a parameter, which entails that E generates different ciphertexts even for the same plaintext input (with high probability).

The encryption scheme is transformed into a K-probabilistic encryption scheme as follows. The key generation function KG remains the same, except for the fact that a random bit string s is generated in addition. A plaintext v is encrypted by using the pseudo-random number generator G with the parameters K, s, and v to generate K random components $r_1, \ldots, r_K$ and then choosing any r_i, $i \in \{1, \ldots, K\}$, uniformly at random as input for E together with v. The decryption function D remains unchanged. S is a K-probabilistic encryption scheme as at most K different ciphertexts are output for any plaintext by design. Since any (valid) random component r can be used so that $D(E(v, r)) = v$, it follows immediately that the decryption function is also correct for any of the K random components that are computed deterministically for given K, s, and v. If an additively homomorphic probabilistic encryption scheme is turned into a K-probabilistic encryption scheme, it is easy to see that the resulting scheme is additively homomorphic as well.

Encryption of Runs. Typical time-series data are repetitive, i.e, measurements within a certain interval of time tend to have the same value, creating so-called *runs* of the same value in the data stream. The encryptor encrypts equal values occurring within a run with the same ciphertext by using the same random component in the encryption function during the run. This technique allows the server to leverage runs during compression. Note that this enables the server to learn how many times the same value occurred in a row. This leakage of information can be reduced by breaking runs up into shorter sub-runs and using different ciphertexts for different sub-runs. This straightforward mechanism enables a customized trade-off between confidentiality and compressibility.

Caching. Encryption and decryption are computationally expensive, yet a simple caching mechanism can speed them up. The encryption algorithm caches newly encountered pairs of plaintext and ciphertext index $i \in \{1, \ldots, K\}$, together with the corresponding ciphertext. The cached value is used whenever an entry is found in the cache; otherwise, the encryption is performed as described above. The decryption algorithm also uses the cache, which improves performance especially for queries that require a lot of decryption.

Query Rewriting. The query rewriter receives a query generated by the client and outputs a transformed query ready for processing by the server. The transformation consists in

[2]Informally, an encryption scheme is indistinguishable under a chosen plaintext attack, where every probabilistic polynomial time adversary cannot guess well for a given ciphertext which of two plaintexts it encrypts.

```
SELECT sum(x), mean(y)
WHERE x = 10 AND
y BETWEEN 20 AND 25
GROUP BY x
ORDER BY y
```

```
SELECT paillier_sum(E(x)),
paillier_sum(E(y)), count(E(y))
WHERE E(x) IN [E_min(10),E_max(10)]
AND E(y) IN [E_min(20),E_max(25)]
GROUP BY E(x)
ORDER BY E(y)
```

Fig. 2: An example query (left) and the corresponding encrypted query (right).

encrypting the column names and the constants in the query, mapping the plaintext operations to their ciphertext counterparts, and transforming all the predicates in the WHERE clause of the query into range predicates by exploiting the order relation defined on the ciphertexts. This transformation prevents the server from learning the formulation of the predicates in the original query. As depicted in Figure 1, a transformed query can consist of multiple sub-queries $eq_1, \ldots, eq_m$ that must be executed first and whose results must be returned before the main query $E(q)$ is executed. An example of a plaintext query and its encrypted counterpart is given in Figure 2.

C. Server

The server contains three modules: the compressor, the order-preserving module (OPM) enabling it to carry out range queries, and the database itself. Each module is based on standard techniques, i.e., no special hardware or software is necessary to benefit from our approach, and other compression and query processing methods than the ones described in the evaluation section could be used as well. The database on the server comprises a single table where each column c represents a type of measurements. A column c contains all feature vectors in compressed form and a bitmap index (BI). Upon reception of a feature vector, the database updates the BIs associated with the columns. The BIs help in the query processing to minimize the data that needs to be decompressed. In order to save space, the BIs are compressed as well. We now describe the functionality of the main components in more detail.

Compressor. The server compresses the data it receives before storing it to reduce the growth of the database size. The compressor receives a stream of data items and compresses it in *batches* of several feature vectors, which are then stored in the database. Finding the "right" batch size is a difficult problem because a larger batch size generally increases the potential of compression and a smaller batch size can be advantageous for partial decompression and fast insertion. However, more I/O operations are needed when processing queries and the available memory is not used efficiently if the batch size is too small [7]. Thus there is a trade-off between the disk-size used and data processing operation speed.

Aliasing. In order to reduce disk-space utilization, an aliasing scheme is used to cope with the fact that ciphertexts are large. An ordered dictionary with the ciphertexts as keys is constructed by the server to limit the ciphertext occurrences to the strict minimum. For each ciphertext $E(d_{i,c})$ obtained, the server checks whether $E(d_{i,c})$ is in the dictionary. If it is not, the ciphertext is given an alias $A_{E(d_{i,c})}$ and the key $E(d_{i,c})$ and its value $A_{E(d_{i,c})}$ are inserted into the dictionary. Since the dictionary is ordered, both the key and the value can be retrieved easily from one another. In our system, $A_{E(d_{i,c})}$ is simply the value of a counter that is incremented for each new ciphertext. The aliases $A_{E(d_{i,c})}$ of the E($d_{i,c}$) of column c are grouped together in batches, which are compressed and then stored in the database.

Order-Preserving Module (OPM). We define the following total order relation $\preceq$ on ciphertexts: Let c and c' be any ciphertexts corresponding to plaintexts $D(c) = v$ and $D(c') = v'$ and let r_i and r_j be the random numbers used in E to generate these ciphertexts. We define that $c \preceq c'$ if either $(v < v')$ or $(v = v'$ and $i \leq j)$. Thus, the order *does not* correspond to the numerical interpretation of the ciphertexts. Since $c_1 \preceq c_2 \preceq \ldots \preceq c_K$ for any v, we further define $E_{min}(v) := c_1$ and $E_{max}(v) := c_K$. The server cannot determine the total order on the ciphertexts the client generated on its own, yet it must know the order to process queries, which contain `WHERE` clauses in the form of range predicates. The OPM maintains the total order relation among the database entries by means of a simple interactive OPE scheme that uses a binary search tree to encode order [14]. We introduced a straightforward modification to this scheme: Rather than using the ciphertexts in both the OPE tree and look-up table, their aliases are used.

Requirement 3 states that the server (or an attacker) may only learn the order relation between two ciphertexts c and c' if they are both generated by the same client. This requirement can be satisfied in multiple ways. The client can keep track of recent data items that were sent to the server for insertion and the data items involved in recently issued queries for which order must be determined. The client will only provide help to insert them into the OPE structures for these data items. A stateless alternative is to verify that the ciphertext that is to be inserted originally came from the client for each request, e.g., by having the client sign ciphertexts [2]. While this approach does not require the client to maintain state, the computational cost is higher.

Data Insertion. The data insertion mechanism works as follows. Feature vectors are sent from the application to the encryptor at a fixed rate. At the encryptor, each vector is encrypted and the encrypted vector is sent to the server. At the server, the BIs are updated and the encrypted vectors are kept in memory until the next batch is complete. At this point, the batch is compressed and stored. In addition, the OPM makes sure that the encrypted vector is inserted into the OPE tree.

Query Processing. Once data is inserted in the database, users must be able to run queries to retrieve information out of the data. Before a user query is sent to the server, the client crypto module uses the query rewriter to rewrite the query as described in §III-B. When receiving a (transformed) query, the database uses the bitmaps to determine which

rows meet the range constraints. Subsequently, the database determines in which batches the matching rows are located. These batches are then decompressed and the corresponding rows are extracted. Finally, the result of the aggregate function in the `SELECT` statement is computed homomorphically over all returned (encrypted) data. The result is then transmitted to the client's crypto module, where it is decrypted, and the plaintext result is made available to the client.

IV. Security Analysis

As described in the model section, the threat model we use for our design assumes that an attacker is in control of the server side. Therefore, as depicted in Figure 1, only encrypted feature vectors and queries are sent from the client to the server and thus sensitive data is never available in plaintext at the server.

IND-CPA Security. We start by showing that the encryption protects the data in the sense that the K-probabilistic encryption scheme is IND-CPA secure, as required by Req. 2, if the underlying probabilistic encryption scheme is IND-CPA secure. We assume that the pseudo-random number generator G that is used to generate the random input to E generates bit strings that cannot be distinguished from truly random bit strings. Formally, for any K, s, and v, we assume that $G(K, S, v) = \{r_1, \ldots, r_K\}$ is indistinguishable from any set R, $|R| = K$, chosen uniformly at random from the set $\mathcal{R}$ of all valid random inputs to E. For the sake of contradiction, assume that there is a chosen-plaintext attack where an attacker first encrypts a set of chosen plaintexts and then receives the ciphertext challenge $c = E(r_i, v)$, where $v \in \{0, 1\}$. Since r_i is essentially random, an attacker that is able to identify v with probability substantially larger than 0.5 can use the same attack to identify v for the underlying probabilistic encryption scheme, which contradicts the assumption that the underlying scheme is IND-CPA secure. Note that the fact that the same ciphertext can re-occur for the K-probabilistic encryption scheme does not help the attacker.

Plaintext Runs. How much is revealed about the plaintext by the encryption of runs depends on the properties of the data. If runs occur only for a few plaintext values, then it is necessary to break these runs into short sub-runs, use a large enough K, and encrypt the sub-runs separately. Otherwise an attacker can guess which values are encrypted in runs. However, for data where many values occur in runs (Req. **??**) and the expected lengths of the runs of different values do not vary much, the proposed scheme does not enable the attacker to learn the values of encrypted runs.

Plaintext Distribution. Due to the K-probabilistic scheme, neither the plaintext frequency distribution nor the exact number of distinct plaintext values can be learned. The larger K, the less the ciphertexts reveal about the plaintexts. Figure 3 illustrates the frequency of ciphertexts for $K = 1, 2, 4, 8$ for normally distributed data of a plaintext domain with 1000 elements. For $K = 1$ the most frequent ciphertext occurs 2.8 more often than the median element which is picked with probability $1/1000$ as expected. For $K > 1$, at most $K \cdot 1000$ different ciphertexts are possible and the most frequent one among them appears roughly $1/K$ as often as the most frequent ciphertext for $K = 1$. In other words, the curves are flatter for higher values of K, i.e., the distribution becomes closer to the uniform distribution. This can also be seen by considering the min-entropy of the ciphertext distributions which grows logarithmically with K. If there are ϕ distinct ciphertexts in the database on the server, the number of distinct plaintexts lies in the range $[\lceil \phi/K \rceil, \phi]$. Therefore, it is not possible for the attacker to figure out exactly which range of ciphertexts corresponds to the same plaintext.

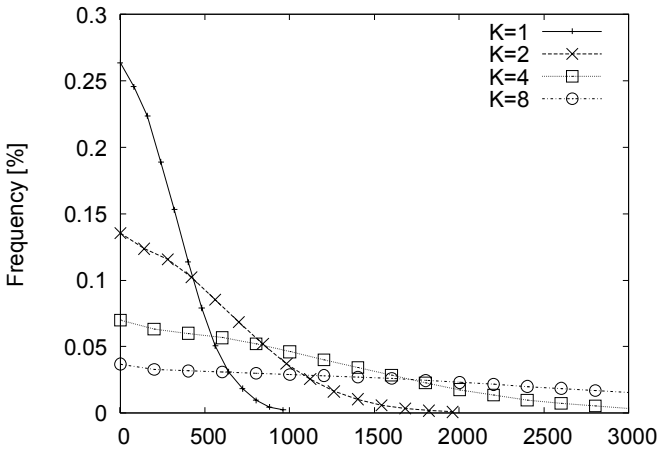

Fig. 3: Frequency distribution plot of K-probabilistically encrypted normally distributed data. Given a set of plaintexts drawn according to a discretized normal distribution of 1000 elements, the corresponding ciphertexts were produced with varying values for K. The frequency of the ciphertext occuring most often is depicted on the left of the plot. Observe that the gradient of the plotted lines approximates zero as K is increased.

Range queries with OPM. Since range queries are supported (Req. 1), the server can learn the order of all ciphertexts encrypted by the client from the OPM. However, the server cannot infer order information for additional ciphertexts it generated itself for chosen plaintexts because the client only reveals order information to the OPM with a mechanism checking the authenticity of the ciphertexts (either maintaining some state at the client or with a signature mechanism). Thus, Req. 3 is satisfied and the attacker cannot learn how the client's ciphertexts are ordered with respect to other values. Moreover, as only a small subset of all possible values in the domain are ever stored in the OPM (Req. **??**), an attacker cannot reliably match the sorted ciphertexts to plaintext values. Queries reveal what columns or specific ciphertexts are requested, with all `WHERE` clauses in the form of range queries.

V. Performance Evaluation

The performance of our system is analyzed in this section, focusing on the effectiveness of our compression mechanism and the impact on the time required to insert data and run queries. After discussing the setup of our evaluation environment, we present and discuss our experiments with respect to compression and running times.

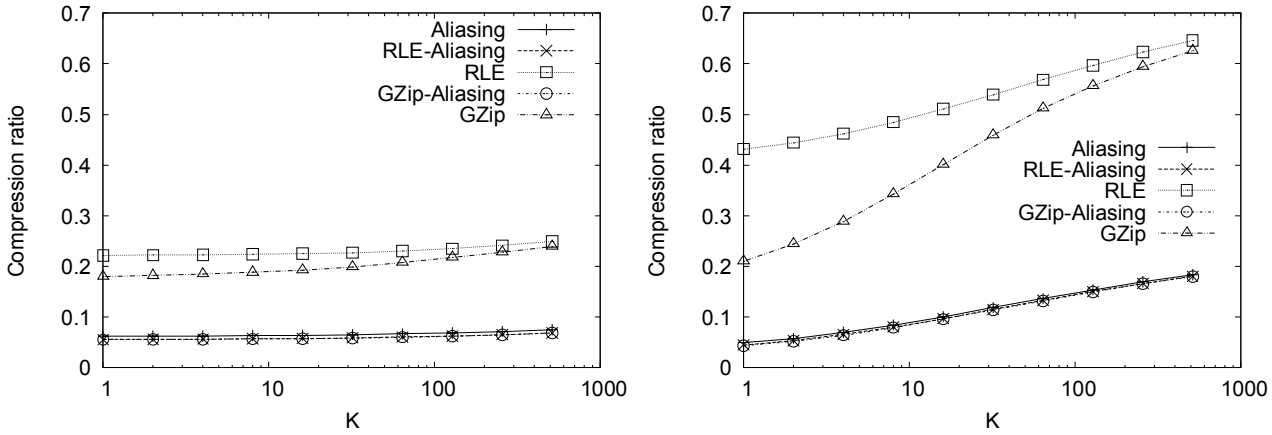

Fig. 4: Compression ratio as a function of K for SPP (left) and WPP (right) data.

A. Setup

Since the crypto module and the server are the two key system components, we evaluate their performance exclusively. Both components were executed on the same machine, i.e., overhead due to communication over a network is not considered. This omission is not crucial whenever the amount of data that is transferred at a time is relatively small, e.g., when transferring a feature vector or running a query returning a few 1000 entries or less as a result. In this case, the running time for computation on encrypted data typically exceeds the data transfer times by far. On the other hand, if there is little bandwidth and the query returns a large portion of the database, the query execution time is large in any case, which entails that the end-to-end overhead of our system is actually smaller. The experiments were conducted on computers with an i5-4570 quadcore CPU at 3.2 GHz with 8GB of RAM. As the basis for the K-probabilistic encryption scheme, we use the probabilistic and additively homomorphic Paillier encryption scheme [13]. The ciphertexts are generated with a 1024-bit key, which yields 2048-bit ciphertexts in the Paillier cryptosystem.

Data from real-world wind and solar power plants (WPP and SPP) were used to evaluate the system. The feature vectors contain 22 and 25 features for SPP and WPP data, respectively. These features are numerical values for the voltage, current, and power produced by the power plants as well as numerical parameter settings, sensor values and error codes, encoded as integers and floating-point numbers. Since Paillier expects integer values, we encoded floating-point values as integers using fixed-point arithmetic. Typically a low number of bits suffices to achieve highly accurate results (see, e.g., [12]), and the overhead in terms of additional bits required for the fixed-point representation is easily offset by the fact that ciphertexts are larger than plaintext values. We started with an empty database and inserted 100,000 feature vectors in total. This number of vectors is large enough to get a representative picture of the entire data set. The SPP data is an ideal time-series test case because it contains many repetitions for various features, which lends itself nicely to our encryption and compression mechanism. The WPP data has substantially fewer repetitions and more distinct values, i.e., the entropy is considerably higher. Thus, analyzing the behavior of our system for both data sets allows us to assess the negative effect on compression rate and running times when dealing with more volatile time-series data.

We constructed a set of queries inspired by the TPC-H benchmark[3]. The original queries in the benchmark needed to be adapted for our data sets. In particular, the name and number of columns and also the operations executed on the data needed to be adjusted. In total, we built two groups of queries, the second group being equivalent to the first one except for the fact that aggregation operations are performed on the result set(s), i.e., one or more aggregate operators listed in Table I are applied to the data satisfying the `WHERE` clauses.

B. Compression

We evaluate our compression mechanisms by measuring the achieved *compression ratio*, which is simply the ratio between the space required for the compressed database and the database without compression.

Apart from the data itself, the compression ratio depends on the parameter K, which defines how many ciphertexts can exist at most for a given plaintext. We consider the influence of K when studying the compression ratio for the following five compression schemes:

- RLE: based on the run-length encoding scheme RasterZip [7].
- GZip: standard compression utility based on the Lempel-Ziv algorithm [22].
- Aliasing: described in §III-C.
- RLE-Aliasing: combination of RLE and Aliasing.
- GZip-Aliasing: combination of GZip and Aliasing.

Note that the bitmap indices are always compressed using the *Word-Aligned Hybrid* (WAH) compression scheme [20], which offers efficient compression of bitmap indices even in case of high-cardinality features. Moreover, it supports bitwise operations for querying without prior decompression.

[3] See http://www.tpc.org/tpch/.

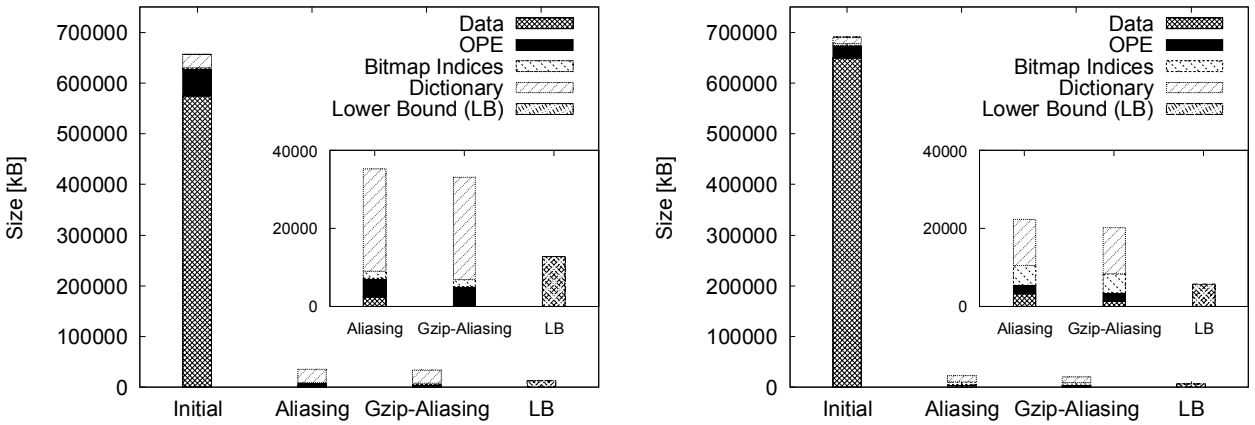

Fig. 5: Breakdown of the space requirements of individual database components for SPP (left) and WPP (right) data and $K = 2$.

While optimizing the batch size is typically not a trivial task, we found that its impact is relatively small in our case. Therefore, we used a fixed size of 4000 vectors per batch, which yields good results in our experiments. In the evaluation we compare the performance of five compression schemes.

Intuitively, one expects the compression ratio to degrade as K increases because a larger K entails a larger cardinality of the data set. Figure 4 depicts the effect of increasing K for SPP and WPP.

The key observation is that aliasing has a strong effect on the compression ratio due to the fact that ciphertexts are much larger than their aliases. By storing each ciphertext only once, the biggest reduction in the required space is achieved. When using aliasing alone, an improvement of roughly 72% (85%) over GZip can be achieved for SPP (WPP) data regardless of K. GZip or RLE alone is not sufficient, particularly for WPP data where the compression ratio is worse than 0.6 for large K. The graphs also reveal that GZip generally outperforms RLE. The compression ratio can further be improved by combining aliasing with RLE or GZip but the additional gain is relatively small at up to 10% because the ciphertext themselves cannot be compressed. This gain shrinks when increasing K, in particular for the WPP data where the gain drops to less than 5% for $K = 512$. The graphs confirm the intuition that increasing K has a negative effect on the compression ratio, in particular for high-entropy data (such as the WPP data).

Four components contribute to the total database size: the data, the OPE data structures (OPE tree and lookup table [14]), the bitmap indices, and the dictionary mapping bitmap indices to ciphertexts. Figure 5 depicts the breakdown for $K = 2$. Without aliasing, the data component requires by far the most space, followed by the OPE data structures, because these structures store ciphertexts. The third largest component is the aforementioned dictionary. When aliasing is used, the ciphertexts in the data component and the OPE data structures are replaced with pointers into the dictionary. As a result, both of these components shrink substantially: For SPP and WPP data, the compression ratio of the data and OPE component is approximately 0.0043 and 0.045, respectively. At this point, the aliasing dictionary requires most space. The only component that can be compressed further is the data component because the dictionary contains high-entropy ciphertexts and the bitmap indices and the OPE are already compressed using WAH and GZip, respectively. A final compression ratio of 0.0006 is achieved for the data component when combining Gzip compression and aliasing.

An important question is if we could potentially do better. The last "bin", denoted by *LB*, addresses this question in that it shows a *lower bound* on the space requirement, which is simply the number of distinct values in the database times the size of a ciphertext. When applying aliasing and GZip, the space requirement is merely a factor of 1.22 (1.71) larger than this lower bound for SPP (WPP) data. The same experiment has been repeated for $K = 512$, resulting in a factor of 1.80 and 29.56 for SPP and WPP data, respectively.

Since the dictionary takes up most of the space and its size grows linearly with the number of distinct ciphertexts, the database also grows linearly with K. The slow increase of the compression ratio in Figure 4 is due to the fact that almost all runs are encoded with different ciphertexts for a small K already, and further increasing K does not have a significant impact for the same data set. In general, K must be set to a small value to keep the compression ratio low.

C. Running Time

Next, we examine the running time of data insertion and query processing, again depending on the parameter K.

Cryptographic Operations. Before adding a new value to the database, it needs to be encrypted and inserted into the OPE data structures. After a query from the client has been processed on the server, one or several values of the result set need to be decrypted. We analyze the duration of the cryptographic operation time for five hundred feature vectors taken from the SPP data set for $K = 2$ and $K = 128$. The results of this evaluation are presented in Table II. We observe that the duration of cryptographic operations is largely unaffected by the choice of K. On the other hand, caching substantially reduces the mean duration since a proper encryption operation is only needed for newly encountered values and many of the decryption operations are replaced by lookups. However, increasing K counteracts the positive effect of caching because

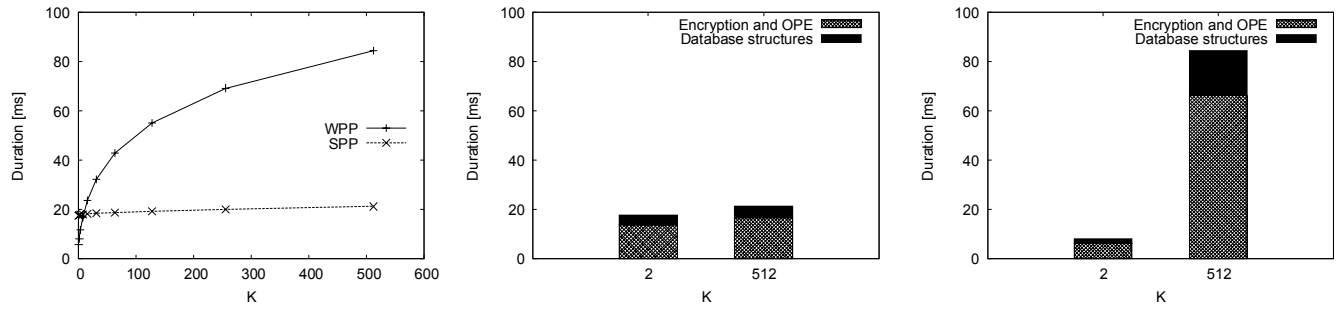

Fig. 6: Average duration of insertion per row as a function of K (left). Breakdown of running time for SPP (center) and WPP (right) of insertion per row into *encryption and OPE* and *DB maintenance*, for $K = 2$ and $K = 512$.

K	Before insertion [ms]		After query [ms]	
	without cache	with cache	without cache	with cache
2	9.59	4.19 (43.7%)	15.9	4.9 (30.8%)
128	9.52	5.73 (60.2%)	16.1	6.3 (39.1%)

TABLE II: Duration of cryptographic operations for data preparation (encryption and OPE insertion) before inserting feature vectors into the database and after query processing (decryption), per feature vector in milliseconds, with and without caching.

a larger K implies that there are more distinct ciphertexts, resulting in more cache misses.

Data Insertion. The more distinct ciphertexts have been inserted into the database, the higher the cost of *database maintenance*, which encompasses all operations related to the used data structures (bitmap indices and the dictionary). In other words, all operations *not* related to encryption or the OPM are considered database maintenance operations. Having studied the cost of encryption and OPE insertion, we now examine the cost of the entire data insertion process.

Figure 6 shows a slow increase in data insertion duration with respect to K for the SPP data. In this data set, most of the features contain sequences of long runs while the rest contain distinct values at each row. This composition means that the second group of features is responsible for most of the duration of data insertion. Thus, K has no impact on their encryption and their contribution to database maintenance. The increase in duration is due to the long runs. Caching minimizes the impact of K on encryption and therefore on data insertion duration. Since the data contains long runs, the number of different ciphertexts used per plaintext is small, thereby also mitigating the impact of K on database maintenance and therefore on the total data insertion duration.

On the other hand, for WPP data, a sharp increase can be observed in data insertion duration when varying K. There are no columns that contain distinct values at each row in the WPP data set. Many columns contain many short runs, which leads to a high probability of using multiple ciphertexts for a plaintext, resulting in more time spent on database maintenance. Therefore, the increase of K (from 2 to 512) in the presence of short runs causes a substantial (more than tenfold) increase in both the encryption and database maintenance duration.

Query Processing. In this part, we study the total time needed to rewrite queries at the crypto module and execute the queries at the server.[4] The first experiment examines query processing duration as a function of K. For this evaluation, two queries (0 and 1) of the adapted TPC-H benchmark queries and their counterparts making use of aggregate functions (10 and 11) were selected.

The intuition that query processing is slowed down when increasing K is confirmed in Figure 7. The processing time increase naturally depends on the query: The increase for queries 0 and 10 is lower than for queries 1 and 11 due to the fact that the `WHERE` clause is applied to fewer columns. A larger K implies a higher ciphertext cardinality, which leads to an increase in the number of BI operations. This in turn affects the performance of query processing negatively. A closer look the performance numbers reveals that for $K \leq 4$ query processing takes less than twice as long as for $K = 1$. Assuming that a slow-down factor of 2 for query processing is acceptable, an acceptable choice for K given our data would thus be $K = 4$.

The duration of all queries for $K = 4$ is depicted in Figure 8, presenting the breakdown of the duration into different processing steps for each query. Additionally, this figure shows the mean duration of query processing for $K = 1$ and without compression (*baseline execution*). We observe that query 10 has the longest execution time for both SPP and WPP data. The main contributors to this long duration are the aggregate operations and the formatting operations required by this query, which combines a large number of rows returned and a `GROUP BY` clause. The execution of the `SELECT` clause is the second most expensive operation, especially for queries that return a large number of rows because reading from disk takes time. Queries with aggregations perform worse than their counterparts, since their performance is impeded by the formatting of the data.

Note that the server returns larger groups for `GROUP BY` operations if $K > 1$ than for $K = 1$. This is due to the fact that for $K > 1$, a plaintext has more than one ciphertext and the server sees them as different values. Hence, some extra refinement steps are necessary on the client side to consolidate the results.

[4]The duration of the decryption process after the server has returned the result has been analyzed earlier in this section.

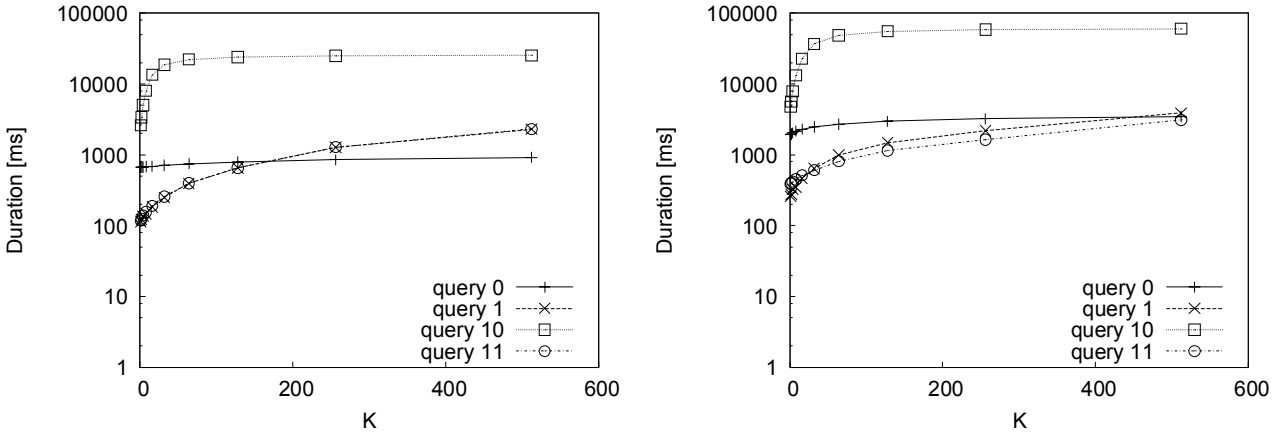

Fig. 7: Duration of four queries as a function of K for SPP (left) and WPP (right) data. Query 10 and 10 are the counterparts with aggregation of query 0 and 1, respectively.

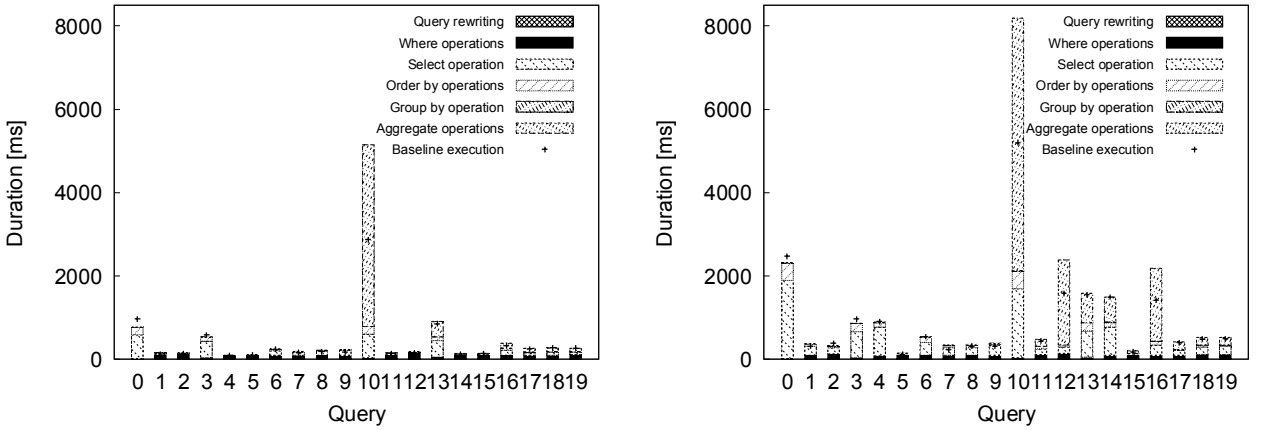

Fig. 8: Histogram of query processing duration for solar (left) and wind (right) data for each of the twenty queries. The breakdown includes the duration of query rewriting and execution of `WHERE`, `SELECT`, `ORDER BY`, `GROUP BY`, and *aggregate* operations. The graph also shows the duration of each query in the *baseline execution* ($K = 1$, not compressed).

For the first ten queries, setting $K = 4$ allows the system to process queries slightly faster than the baseline execution because more data must be retrieved due to the lack of compression. However, for the last ten queries, the baseline execution performs better since it returns fewer groups and needs less time for formatting than an execution with $K > 1$. For ≈50% (≈35%) of the queries, the system with compression and $K = 4$ was faster than the baseline execution for SPP (WPP) data. For the other queries, the measured median slowdown was quite small at ≈2.24% (≈4.66%) for SPP (WPP) data. The observed trends also hold for other moderate values of K.

VI. Related Work

Encrypted data processing has been widely studied in recent years [4], [5], [10], [17], [21]. Since fully homomorphic encryption [8], which enables the evaluation of arbitrary Boolean circuits, is still impractical due to its large computational overhead, the focus has mainly been on executing specific operations on encrypted data.

Order-preserving encryption (OPE) is a well-studied concept [1], [5], [6], [9]. However, all OPE schemes were found to leak more than just the order because they use weaker security guarantees. Recently, an *interactive scheme* [14], where multiple communication rounds between client and server are needed, was proposed that achieves ideal security, meaning that no more than the order of the ciphertexts is revealed. Our modifications to this encryption scheme are described in §III-B and discussed in §IV.

Probabilistic encryption offers the best security but at the same time renders searching quite difficult. In order to improve searching, secure indexing mechanisms [16] have been proposed. While the proposed secure index maintains a high level of security, they are not suitable for our purposes because the suggested use of probabilistic encryption is costly and the order among ciphertexts and the number of repetitions are revealed anyway in our approach.

Several complete database systems, using partial homomorphic encryption, have been proposed that achieve an overhead

small enough for practical applications [3], [15], [18], [19]. CryptDB [15] is a database system that enables the execution of queries over encrypted data for standard relational databases achieves strong performance numbers for certain benchmarks. In order to support different operations, CryptDB uses multiple encryption schemes, each scheme enabling certain operations. These encryption schemes are applied to the data in an onion approach, going from the less secure to the most secure encryption scheme (as the outermost layer). CryptDB adjusts the encryption layer based on the queries processed. MONOMI [18], which builds on CryptDB, is a system that supports more general (analytical) workloads over encrypted data. It uses a *split client/server query execution* approach where some parts of the query are executed at the client to improves efficiency and enable more complex queries and data processing. While CryptDB and MONOMI offer "reasonable" security (depending on the application), they do not offer server-side compression, an essential feature for time-series based applications. Our approach focuses on disk space utilization at the expense of security. Hence, instead of having multiple encryption schemes, it uses a single scheme that covers all required operations. Naveed et al. have described inference attacks that can be mounted against CryptDB storing medical data [11]. Thus, security for such systems also depends on the data that it stores (and how the system is set up). This observation is also true for our system. However, as mentioned earlier, the needed auxiliary information may not be available and the entropy of the data may be too high to launch such attacks for industrial-type time-series data.

VII. Conclusion

Processing time-series data efficiently in a confidentiality-preserving manner is a challenging problem, which is even harder when taking the requirement to bound the space complexity into account. In this paper we described an approach based on partially homomorphic K-deterministic encryption and query rewriting on the client to build a system that exploits standard methods for compression and query processing on the server. We have shown this can lower the space requirements substantially: Our experiments with real-world data from solar and wind farm power plants reveal that the size of the database can potentially be reduced by approximately 90-95% compared to uncompressed and encrypted data. The total space requirement is merely a small factor larger than the absolute minimum that is needed to store ciphertexts for all distinct data values. Another positive result is that the processing time of roughly one third to one half of all tested queries was *reduced* when applying our compression techniques. The median change in running time was close to zero, at a slowdown of merely 2-5% for solar plant and wind farm data. Thus, tremendous space savings for encrypted time-series databases are possible at a minor additional cost in terms of data processing time.

References

[1] D. Agrawal, A. El Abbadi, F. Emekci, and A. Metwally. Database Management as a Service: Challenges and Opportunities. In *Proc. 25th IEEE International Conference on Data Engineering (ICDE)*, pages 1709–1716, 2009.

[2] J. H. An, Y. Dodis, and T. Rabin. On the Security of Joint Signature and Encryption. In *Proc. 21st International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 83–107, 2002.

[3] S. Bajaj and R. Sion. TrustedDB: A Trusted Hardware Based Database with Privacy and Data Confidentiality. In *Proc. SIGMOD International Conference on Management of Data*, 2011.

[4] M. Bellare, A. Boldyreva, and A. O'Neil. Deterministic and Efficiently Searchable Encryption. In *Proc. 27th Annual Conference on Advances in Cryptology–CRYPTO*, pages 535–552, 2007.

[5] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-Preserving Symmetric Encryption. In *Proc. 28th Annual Conference on Advances in Cryptology–EUROCRYPT*, pages 224–241, 2009.

[6] A. Boldyreva, N. Chenette, and A. O'Neill. Order-preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions. In *Proc. 31st Annual Conference on Advances in Cryptology*, pages 578–595, 2011.

[7] F. Fusco, M. Vlachos, and X. Dimitropoulos. RasterZip: Compressing Network Monitoring Data with Support for Partial Decompression. In *Proc. 12th ACM Conference on Internet Measurement Conference (IMC)*, pages 51–64, 2012.

[8] C. Gentry, S. Halevi, and N. Smart. Fully Homomorphic Encryption with Polylog Overhead. In *Proc. 31st Annual Conference on Advances in Cryptology–EUROCRYPT*, pages 465–482, 2012.

[9] S. Lee, T.-J. Park, D. Lee, T. Nam, and S. Kim. Chaotic Order Preserving Encryption for Efficient and Secure Queries on Databases. *IEICE Transactions on Information and Systems*, E92.D(11), 2009.

[10] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Authenticated Index Structures for Aggregation Queries in Outsourced Databases. *ACM Transactions on Information and System Security (TISSEC)*, 13(4):32:1–32:35, 2010.

[11] M. Naveed, S. Kamara, and C. V. Wright. Inference Attacks on Property-Preserving Encrypted Databases. In *Proc. 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 644–655, 2015.

[12] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 334–348. IEEE, 2013.

[13] P. Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Proc. 18th Annual Conference on Advances in Cryptology–EUROCRYPT*, pages 223–238, 1999.

[14] R. A. Popa, F. H. Li, and N. Zeldovich. An Ideal-Security Protocol for Order-Preserving Encoding. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, pages 463–477, 2013.

[15] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proc. 23rd Symposium on Operating Systems Principles (SOSP)*, 2011.

[16] E. Shmueli, R. Waisenberg, Y. Elovici, and E. Gudes. Designing Secure Indexes for Encrypted Databases. In *Proc. 19th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (DBSec)*, pages 54–68, 2005.

[17] B. Thompson, S. Haber, W. G. Horne, T. Sander, and D. Yao. Privacy-Preserving Computation and Verification of Aggregate Queries on Outsourced Databases. In *Proc. 9th Privacy Enhancing Technologies Symposium (PETS)*, pages 185–201, 2009.

[18] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing Analytical Queries over Encrypted Data. In *Proceedings of the VLDB Endowment*, volume 6, pages 289–300, 2013.

[19] W. K. Wong, B. Kao, D. W.-L. Cheung, R. Li, and S.-M. Yiu. Secure Query Processing with Data Interoperability in a Cloud Database Environment. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 1395–1406, 2014.

[20] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing Bitmap Indices With Efficient Compression. *ACM Transactions on Database Systems (TODS)*, 31(1):1–38, 2006.

[21] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis. Authenticated Join Processing in Outsourced Databases. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 5–18, 2009.

[22] J. Ziv and A. Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.