CloudDBGuard: A Framework for encrypted data storage in NoSQL Wide Column Stores

Lena Wiese[a,*], Tim Waage[b], Michael Brenner[c]

[a] *University of Hannover, Knowledge-Based Systems Group/L3S Research Center, Appelstr. 4, 30167 Hannover, Germany*
[b] *University of Goettingen, Institute of Computer Science, Goldschmidtstr. 7, 37077 Goettingen, Germany*
[c] *University of Hannover, Computational Health Informatics, Schloßwender Str. 5, 30159 Hannover, Germany*

Abstract

Nowadays, cloud storage providers are widely used for outsourcing data. These remote cloud servers are not trustworthy when storing sensitive data. In this article we focus on the use case of storing data in a cloud database using a particular sub-category of NoSQL databases – so-called wide column stores. Unfortunately security was not a primary concern of the NoSQL systems designers. Using encryption before outsourcing the data can provide security. Conventional encryption however limits the options for interaction because the encrypted data lacks properties of the plaintext data that the database systems rely on. Various schemes have been proposed for property-preserving encryption in order to overcome these issues, allowing a database to process queries over encrypted data. In this article we comprehensively present details of our framework CloudDBGuard that allows using property-preserving encryption in unmodified wide column stores. It hides the complexity of the encryption and decryption process and allows various adjustments on specific use cases in order to achieve a maximum of security, functionality and performance.

Keywords: Property-preserving encryption, NoSQL databases, Wide Column Stores
2010 MSC: 68P25

1. Introduction

In times of "Big Data" and the "Web 2.0" [1] conventional (often SQL-based) databases may run into difficulties due to changed usage requirements. On the one hand there are new performance demands. While SQL environments usually focus on being ACID [2] compliant, it is much more important for modern web services to to address the trade-off between high availability, consistency and (since they naturally run in distributed environments) to be tolerant regarding network partitions in the underlying infrastructure [3]. On the other hand there are different demands regarding the data structures themselves. SQL tables are not well suited to represent loosely structured or sparse data sets like they are typical for today's webservices.

NoSQL (Not only SQL) databases (see [4] for a comprehensive survey) were designed for meeting those new requirements, and they attracted more and more attention over the last years. A special sub-category of NoSQL databases are so-called wide column stores (WCS). Many of the global players on the market developed their own solutions: examples like Google's Bigtable [5](used for instance in the Google search engine, Google Maps, Google Earth, Youtube) or Facebook's Cassandra [6] (used for instance in Twitter, Reddit and Facebook itself until 2011) show that almost everybody uses several services in their daily life that heavily utilize NoSQL WCSs. Because of their flexible and efficient behavior, WCSs are also offered by cloud storage providers, where users can book storage space on demand. It is hence also common to use

*Corresponding author
Email addresses: wiese@l3s.de (Lena Wiese), tim.waage@cs.uni-goettingen.de (Tim Waage), brenner@luis.uni-hannover.de (Michael Brenner)

WCSs as a cloud storage customer. In particular, several cloud database providers offer flexible on-demand services for running Apache Cassandra or Apache HBase databases remotely in their clusters.

Most WCSs have not been designed with security aspects in mind. The fact that WCSs usually lack security features like authentication or user (rights) management exacerbates this problem. The only exception is Apache Accumulo that provides cell-level access control but no encryption. This raises concerns how confidential data can be protected from a curious cloud storage provider or other attacks by unknown third parties. The common WCSs do not offer any means to ensure the protection of confidential data from unauthorized access.

Data confidentiality and security can be provided by encryption. But besides the protection of data there is also the requirement for the ability to process the data efficiently in cloud services. Thus, in general there is a tension between these two needs. Ideally direct processing of encrypted data should be possible in order to avoid downloading data to a trusted site, decrypting, processing and then re-encrypting and uploading the data again. We target the use case of managing and querying data in WCSs. For the sake of not compromising the working principles of the database systems, the data needs to be kept searchable (e.g. text) and sortable (e.g. text, numeric values). That means certain information (e.g. order relations) is supposed to leak intentionally. This concept is commonly referred to as property-preserving encryption (PPE). While several existing approaches made use of PPE schemes, none of them targeted the application of PPE for WCS. We extend prior work [7, 8, 9, 10] as follows:

- we present the architecture of our framework called "CloudDBGuard"[1] for utilizing property-preserving encryption in NoSQL WCSs and discuss several practical considerations that have to be taken into account when running encryption frameworks with these database systems;
- we give an in-depth account of implementation details that are specific to managing and querying encrypted data in WCSs – in particular, when providing a unified API for HBase and Cassandra;
- we investigate combinations of property-preserving encryption schemes that exploit their advantages and minimize their weaknesses for certain use cases; we organize them in so-called "table profiles";
- we show practical feasibility of our approch by conducting performance comparisons assessing the impact of the encryption overhead using the currently most popular NoSQL WCSs[2] Apache Cassandra [6] and Apache HBase [11]; this allows making statements in the context of real-world technologies.

We also note that our framework offers support for data distribution over multiple databases (even cross-platform); that is, columns of the same table can be further distributed over multiple databases at different cloud providers to enhance security (a concept known as "separation of duties"[12, 13] or "partitioned data security" [14]). We also put much effort into the extensibility of our platform; other WCS database technologies as well as more PPE schemes can easily be added.

The article is organized as follows. Section 2 provides the background on wide column stores, property-preserving encryption and onion layer encryption. Section 3 describes the concepts and interaction workflows of our framework. Section 4 gives an in-depth description of the components of our framework. Section 5 reports on the runtime experiments with 10 benchmark queries that combine several property-preserving encryption types. Section 6 concludes this article with a discussion and suggestions for future work.

2. Background and Related Work

2.1. Motivating Example

An application scenario for the CloudDBGuard framework is a mail server that stores confidential emails of several users. While the users are relieved from storing all emails locally, they still require management features from the mail server. PPE enables the mail server to manage the emails efficiently while preserving

[1] https://github.com/dbsec/FamilyGuard/

[2] Solit-IT: DB-engines ranking - `http://db-engines.com/en/ranking`

their confidentiality. For example, a user wants to search for emails containing a certain search word – which is enabled by searchable encryption. Furthermore, the user requires the server to sort the emails by date in order to retrieve only the most recent ones or the emails from a specified time interval – which is enabled by order-preserving encryption. In order to benchmark this scenario (in Section 5), the widely-used Enron corpus [15] served as our data set. The Enron dataset comprises e-mails of 150 employees of a company. Our test queries combine equality tests (to find a sender exactly), range queries (to find emails in a certain timespan) and word search (on the email body).

2.2. The Adversary Scenario

CloudDBGuard provides data confidentiality in case an attacker gets (hacker) or has (administrator) full read access to the database server. This may also include its hardware (even the physical RAM) as well as its communication to and from the clients. The attacker behaves passively and follows the designated protocol specifications. He wants to analyze and infer information from the data, but he does not manipulate it in any way. He further does not modify queries from the clients or the results being returned. This threat model is commonly referred to as "honest-but-curious" [16]. The assumption of an honest-but-curious attacker is common in many scenarios where it is assumed that the cloud database offers its service truthfully and follows the protocol as otherwise customers would move on to another provider.

Note that while the attacker in the honest-but-curious model remains passive, there is also the possibility of a malicious attacker, that manipulates result sets or protocols. The case of a malicious/active attacker includes the case that responses are withheld by the cloud database or that fake records are returned. These cases address the problem of integrity of database answers. There are several approaches (like authenticated data structures [17, 18]) that handle integrity checking of database responses, however they do not come for free in terms of storage space and response time. Database integrity is not the main focus of the presented system and hence out of scope of this article.

2.3. The Data Model of Wide Column Stores

WCSs (sometimes also called extensible record stores) are inspired by Google's BigTable architecture [5]. Publicly available open source databases that rely on the same or a very similar data model are for example Hypertable [19], Apache Cassandra [6], Apache HBase [11] and Apache Accumulo [20].

The operating principles of WCSs can be roughly described as follows. A so-called keyspace contains a set of tables and is hence analogous to a database in a conventional SQL database system. WCSs furthermore use tables, rows and columns like traditional relational (SQL-based) databases; however, every row has to have an identifier that is unique for the table. The fundamental difference is that columns are created for each row instead of being predefined by the table structure. Thus, except for a row identifier, two rows can have a completely different set of columns, even though they belong to the same table. In a way rows are comparable to sets of key-value pairs: They can consist of an arbitrary number of fields, that are required to have unique names (we call them "column identifiers") and can be of any data type. Another important difference to SQL databases is that rows are maintained in lexicographic order of their row identifiers. As WCSs are distributed systems, ranges of such row identifiers serve as units of distribution. Hence similar row identifiers (and thus data items that are likely to be semantically related to each other) are always kept physically close together, in the best case on neighboring sectors on disk, but at least on the same node of a cluster for the purpose that reads of ranges are restricted to communication with a minimum number of servers. Because row identifiers are used for coordinating distribution, changing them would result in changing the data's physical position within the database (cluster). That is why most WCSs do not even support changing row identifiers.

The smallest units of information are key-value-pairs with the key itself having multiple components. One of these components is a timestamp, enabling the database to maintain an automatic version control, which can be operated in two ways: either by setting a maximum number of versions to keep, or by specifying a "time-to-live" (TTL) after which data items are to be deleted. Other components of the full key needed to access a value are the keyspace, table and column identifier. Thus, more formally WCSs can be considered

sparse, distributed, multidimensional maps (see [5]) of the form

$$(keyspace, table, column, row\ identifier, timestamp) \rightarrow value.$$

WCSs in general heavily profit from table layouts that are tailored to the queries appearing later on. They are designed for queries with filter conditions involving (if possible) only the row identifier column or columns that maintain a secondary index. Thus it is not unusual to have one table per query, even if that causes data redundancy. The fact that storage space is usually not an issue in cloud scenarios, is also helpful in this regard.

Even though all WCSs share the general data model described above, they may differ fundamentally in the underlying architectural concepts. This also applies to Apache Cassandra and Apache HBase. Hence our ambition was to not interfere with the general principles of each database system; the CloudDBGuard framework runs without modifying the database installations.

2.4. Property-preserving Encryption

Property-preserving encryption (PPE) retains certain properties of the plaintext (like order of numerical values) on the ciphertext – or it relies on additional index structures on encrypted values (to support efficient search on encrypted data). The types of PPE relevant for this work are deterministic encryption (DET), order-preserving encryption (OPE) and searchable encryption (SE):

- **DET.** The purpose of DET is enabling the database server to check for equality by mapping identical plaintexts to identical ciphertexts.
- **OPE.** The purpose of OPE is enabling a server to learn the relative order of data elements without revealing their exact values. OPE encrypts two elements p_1, p_2 of a domain D in a such way that $p_1 \leq p_2 \Rightarrow Enc(p_1) \leq Enc(p_2)$ for all $p \in D$. Thus, its use cases are sorting and range queries over encrypted data. A lot of OPE schemes have been proposed with different strategies to map a plaintext to a ciphertext domain (see [21, 22, 23, 24]).
- **SE.** The purpose of SE is enabling a server to search over encrypted data without revealing plaintext data. Most SE schemes use indexes (see [25, 26, 27]), which are encrypted in such a way, that only a token (a so-called trapdoor) sent by the querying user allows for comparing the searchword with the ciphertext. There are also schemes, that avoid having an index by embedding the trapdoor in a special format into the ciphertext itself (see [27]).

2.5. Related Work

This section surveys related work, limited to approaches that are also designed for the honest-but-curious adversary model, compute over encrypted data and rely on encryption to provide data confidentiality.

The most popular work on performing queries over encrypted data is "CryptDB" [28] for mySQL and PostgreSQL. It was the first system that could be considered practical, introducing a variety of innovative features, most importantly: the onion layer model (OLM). However, it uses only PPE schemes that are slow for querying, because the authors avoid (client or server side) indexes. Thus, CryptDB does not scale well when datasets reach a certain size. However, it still receives a lot of scientific attention, in favorable [29, 30] as well as critic ways [31]. "Monomi" [32] can be considered being an extension of CryptDB, trying to support arbitrary SQL queries with the cost of higher requirements for the client machine. "BlindSeer" [33] addresses efficient sub-linear searches for SQL-queries that can be represented as a monotone boolean formula, consisting of the search conditions: keyword match, range and negation. "DBMask" [34] enforces access control cryptographically, based on attribute based access control and combining broadcast and hierarchical key management. [35] propose a distributed architecture for encrypted query execution on PostgreSQL, MySql, and SQL Server relational databases; they include several cryptographic methods and test them with the common TPC-C benchmark. More recently, [36] propose a new encryption scheme that is able to handle conjunctive queries (as a subset of SQL queries).

Some approaches focus managing encrypted data in non-relational database systems. One of them is "Arx" [37] on top of MongoDB, which introduces two proxy servers and needs to know in advance what operations are to be performed on what fields in order to maintain the required indexes; these indexes are built on top of data that are encrypted with conventional symmetric encryption. The Minicrypt [38] system uses Cassandra as the underlying storage system but focuses on compression of data encrypted with conventional symmetric encryption.

Existing benchmarks of querying encrypted data often assess only one encryption method per category in isolation. In contrast, our approach is a comparative one: we aim to integrate different encryption methods into one platform and compare their behaviour for different use cases in order to identify the best settings for each use case.

3. CloudDBGuard Concepts and Workflows

3.1. Concepts and Overview

CloudDBGuard aims for executing queries over encrypted data in WCSs. Several different encryption methods are applied to the plain data to support different kinds of queries. It uses the basic concept of onion layers [39] to surround data with a strong encryption that is only removed when data are actually accessed. CloudDBGuard provides an application programming interface (API) for key management and encryption tasks. Because we support several different database systems, the API of CloudDBGuard hides the complexity of the databases' native APIs; CloudDBGuard hence unifies the data access by mapping the API queries to the query languages and data models of the two underlying database systems (Cassandra and HBase) as detailed in Section 3.4.

The client application using the API of CloudDBGuard runs in a trusted environment. For the API to be able to manage its tasks, it has to maintain auxiliary data, namely keys, metadata and (if necessary) indexes on client side. Such data are stored persistently in the client's file system. The API manages the database connections, data transfer, encrypting and decrypting. Furthermore it keeps track of metadata and key management. CloudDBGuard utilizes advanced (index-based) encryption schemes, which allow the system to scale better when datasets become large. The database server never sees any decryption keys, hence it is never able to decrypt private data. Thus, adversaries (e.g. administrators) are not able to gain sensitive information only from read access. Moreover, the database server software remains unaffected: there is no need to change the database server software in order to work with CloudDBGuard.

As CloudDBGuard uses encryption schemes that potentially use a high number of cryptographic keys, the manual management of these keys is impractical for the user, but since the database server is not allowed to possess them either, they have to be managed and stored on the client side. This is why CloudDBGuard uses a Java Cryptography Extension KeyStore (JCEKS) provided by the Java Cryptography Extension (JCE) for that task. A JCEKS allows storing an arbitrary number of keys, each of which can be accessed using a custom label. The user has to provide only one single password for the client to gain access to all keys, which massively improves the usability.

3.2. Onion layers in WCS

The concept of onion layers was introduced in CryptDB by [39], also calling it *adjustable query-based encryption*, for SQL databases. However WCSs have some fundamental differences, that affect the designs of the onion layer model. Our framework uses four types of onion layers. The following paragraphs describe them from the database perspective.

RND - Random Encryption. The Random Encryption layer provides the maximum security possible, which is indistinguishability under an adaptive chosen plaintext attack (IND-CPA). Two equal plaintext values are mapped to different ciphertexts with a very high probability. This is achieved using AES in cipher block chaining (CBC) mode or Blowfish with randomly generated encryption keys and initialization vectors (IVs). Every row of a table has a column for storing its own individual IV ("RND Row-IVs" in Figure 1) and for every column an own individual encryption key is stored in the column's metadata on client side

("RND column encryption key" in Figure 1). Thus, the server cannot learn any information, which is why this layer is used as outermost layer for all onions, except for the SE onion, which already provides strong security guarantees by itself (depending on the used scheme either IND-CPA or IND-CKA2 as described below). The RND layer cannot be used for any computations over the encrypted data, because it leaks no information relevant for database operations. Thus it protects data that is never required to be processed by query conditions.

DET - Deterministic Encryption. The layer for deterministic encryption needs to store non-probabilistic ciphertexts, meaning the same plaintexts have to be mapped to the same ciphertexts, e.g. in order to check for equality. This is achieved using AES (CBC) with the same randomly generated encryption key and IV throughout the entire table. Both are stored in the table's metadata on client side (see Figure 1). Using the same key for deterministic encryption throughout the entire table enables the system to evaluate equality conditions between different columns inside the same table. As long as the outermost random layer has not been removed, the database cannot infer such equality conditions from the encrypted columns.

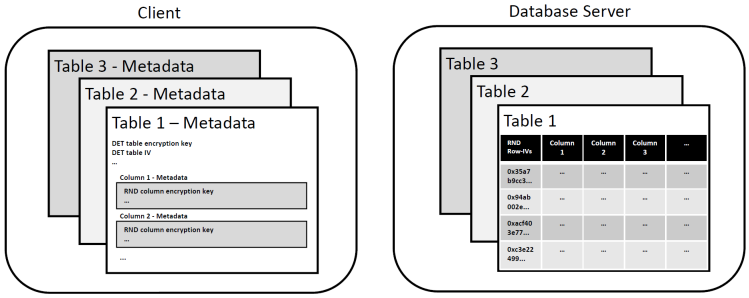

Figure 1: Encryption key and IV management of the RND and DET layer

OPE - Order-Preserving Encryption. We evaluated the performance as well as strengths and weaknesses of various OPE schemes in our framework; the tests were performed using the OPE schemes RSS [22], OACIS [23], mOPE [21]. Their different strengths and weaknesses qualify them for different tasks for the encryption of a table. RSS or mOPE can be used for row identifiers because their ciphertexts are immutable – which maintains database-internal sorting properties discussed in detail in Section 2.3. All other columns that are involved in range queries, can be encrypted with either of the three schemes. However, they differ in their encryption and decryption efficiency and the storage space consumption. RSS is a good allrounder, it is fast to compute, but it requires a client side index. Furthermore, mOPE can be used if client-side indexes are to be avoided; it is thus the most storage-efficient solution but it is also the computationally most expensive OPE scheme. As described in Section 3.3, OACIS cannot be used for row identifier columns because of its mutable ciphertexts; it has the best runtime properties of the available schemes, which makes it especially attractive for large amounts of data; a client-side index is needed as well with OACIS.

SE - Searchable encryption. We realized the SE layer using the schemes SWP [27] and SUISE [26] and evaluated them with our framework. Similar to the OPE schemes their strengths and weaknesses determine their individual use cases when it comes to encrypting a table. SWP does not require maintaining a state or index and thus can avoid using client side storage; the downside of this is that it has a search time linear to the dataset size, since it requires reading the data entirely when searching. Alternatively, using SUISE requires a very small client side index and a rather large additional server side index. SUISE is beneficial when certain queries are supposed to be executed frequently, as it can return results cached in the client-side index beginning from a second search for a keyword in constant time. While SWP satisfies the notion of IND-CPA (as described above), SUISE satisfies the notion of indistinguishability against adaptive chosen keyword attacks (IND-CKA2, see [25]). IND-CKA2 denotes security against an adversary capable to adapt his queries to previously obtained trapdoors and results.

3.3. Mapping Plaintext Columns to Ciphertext Columns

After having defined the onions, one might be tempted to just encrypt all columns using all onions and strip off the necessary layers when querying requires it. However, the WCS data model as well as the PPE schemes themselves come with some aspects that impede this approach.

The first aspect to consider can be inferred from the WCS datamodel itself. As explained in Section 2.3 a fundamental working principle of WCSs is keeping all rows of a table sorted by the content of the row identifier column. That means the database has to perform a sorting by this column already at data insertion time. Thus, it has to be able to compare the row identifier of a new row to be inserted with already existing ones in the database. That means the standard OPE onion cannot be used, since its outermost RND layer does not allow for order comparisons. Using it would break the WCS data model. Therefore row identifier columns must be treated differently from all other columns regarding the onion layer design. They must leak the order of values, before it comes to querying the database and thus they are not allowed to have a RND layer as outermost layer neither can OPE schemes with mutable ciphertexts be used.

A second limiting aspect is caused by the query operators. Depending on the type of data it makes no sense to maintain all onions. For the sake of simplicity we distinguish between three types of data in this work: strings, numerical values and byte blobs. All other data types can be inferred from these three basic types. An example for an onion that makes no sense is the SE onion for numerical values. There is no query mechanism provided by a database, that allows searching for substrings within numerical values.

OPE comes with a third challenge. OPE schemes work by mapping values from a plaintext space ("domain") to a ciphertext space ("range"); it is crucial to define the exact size of both spaces in advance. This is a hard thing to do for strings and byte blobs as they can become extremely large. A straight-forward solution to this problem was introduced by [40], proposing to pad all strings to the same length using 0x00s and then simply use the numerical values given per character by the ASCII table[3]. Assuming a small maximum allowed string length of only 4 bytes "abc"[4] would result in $0x61626300$ (which is equivalent to $97 \cdot 256^3 + 98 \cdot 256^2 + 99 \cdot 256^1 + 0 \cdot 256^0 = 1627389952 + 6422528 + 25344 = 1633837824$ in decimal notation). As can be seen from this example, even for very short strings the numbers soon become very large. Storing the above number would require 31 bits[5]. Limiting the message space to the 96 actually allowed and printable characters of the ASCII table barely improves the situation: $65 \cdot 96^3 + 66 \cdot 96^2 + 67 \cdot 96^1 + 0 \cdot 96^0 = 58122528$ which appears to be a much smaller number in decimal notation, but still takes 26 bits when written binary[6]. In order to still deliver a good compromise between string length flexibility and performance, we use the following approach. We chose to use native Java data types of fixed length which is highly desirable for performance reasons; hence we operate with Java's native data types Integer (32 bits) and Long (64 bits) – in contrast for example to using Java's BigInteger type, that can grow arbitrarily depending on the value it represents which makes it computationally expensive. Our API uses a 32 bit Integer as input for the OPE schemes, while a 64 bit Long serves as output; the ciphertext space is hence at least twice as big in terms of bit length compared to the plaintext space. Because we chose Java's Integer (32 bits) as input for OPE, Strings longer than 4 characters (and thus 32 bit) are split up into chunks, each with a length of 4 bytes, while the last (= least significant) part is padded, if necessary. Those chunks are then used as input being 4 bytes = 32 bits long. After they are OPE-encrypted separately, they are (now having a size of 8 bytes = 64 bits) concatenated again and stored in the database in byte array representation. Since having a pre-defined maximum string length is still mandatory for producing results that are comparable to each other later on, this is done for a exactly eight chunks (= 32 characters). If the plaintext string is not long enough to produce eight chunks, the "least significant" chunks are generated randomly. This will not have an impact on the order after encryption, since the actually existing characters of the original plaintext strings are always completely considered (up to the 32nd byte). Padding with zeros as in [40] or using "0000"-chunks would leak the plaintext string length.

[3] see for instance `http://www.asciitable.com/`

[4] numerical values from the ASCII table: $a = 97 = 0x61$, $b = 98 = 0x62$, $c = 99 = 0x63$

[5] 110 0001 0110 0010 0110 0011 0000 0000

[6] 101 0010 1011 0110 1101 0010 0000

Mapping the plaintext to ciphertext hence depends on the data type stored in a column. String columns are the most complex cases, mainly due to the costly transformation procedure necessary for realizing OPE column as described above. All three onion types are applied for strings. The DET layer can be used for equality checks. The OPE layer is able to sort strings lexicographically or for example do range queries from "A" to "Z". Furthermore the SE onion enables the user to query for single words within larger texts.

Integer columns are easier to handle. They already contain numerical values, such that the costly conversion process explained above is not needed. Furthermore the SE layer is not needed for integers, because searching for substrings within numbers is not supported (if this functionality is desired, string columns should be used instead). Finally, byte blob columns are the simplest cases. Byte blobs usually represent raw binary data (e.g. images), that are not supposed to be searched, ordered by or compared to something. Thus, they are only deterministically encrypted in order to be able to decrypt them efficiently or perform equality checks.

To sum up, the SE layer encryption is only performed, if the plaintext data is a string. OPE layer encryption is only performed, if the plaintext data is a string or numerical value. Every column gets its own instances of the PPE schemes they use. That means in particular, indexes are maintained per column, not per table; answering a query involves only the index data that is actually required.

3.4. Unifying the Data Models of Cassandra and HBase

Cassandra as well as HBase follow the data model of WCSs but they differ in the way of achieving that. Involving both databases at the same time for storing data therefore requires analyzing their differences, which have to be compensated for by the API. The following section discusses key differences of the way both databases realize the WCS data model and how this affected the design of our API; a summary is given in Table 1.

How to Address the Row Identifier Column. Cassandra and HBase have different ways of addressing the row identifier. Cassandra requires assigning a specific name and data type for it in the process of creating a table. In contrast, row identifier columns in HBase do not get an explicit name and are always of type byte blob. Thus, Cassandra has to be given the more precise definitions regarding the row identifier. That is why defining a name and data type is mandatory for creating tables in CloudDBGuard.

Composite Keys. The row identifier can be considered equivalent to SQL's primary key, since both have the task to uniquely identify each row in a table. Thus, fields containing row identifiers must contain unique values and cannot have NULL values. In Cassandra it is possible to combine multiple fields to create a row identifier, which is then called a composite key. A composite key always consists of two parts. The first part is the partition key, that is responsible for data distribution across the nodes like a "regular" row identifier consisting of a single field. The second part is the clustering key, that is responsible for storing data within a partition defined by a certain partition key. Both parts can again consist of multiple fields.

In contrast, HBase does not know the concept of multiple fields defining a row identifier. It always uses one single-field row identifier per row. If the combination of multiple fields is desired for generating unique row identifying key values, that has to be created "manually" (by concatenating values, e.g. append one string to another) and stored as single row identifier. HBase's native Java API provides options for defining column prefix filters, that can be used to "simulate" composite keys. HBase is more restrictive concerning the row identifier design – thus, CloudDBGuard only allows row identifiers consisting of a single field.

Collection Types. Apache Cassandra supports collection types. That means, a single field in a row cannot only contain a single value, but also a list, set or map. The single elements of these collections can be addressed by traversal (set), specifying an index (list) or a key (map). In contrast, HBase does not support collection types, but has an additional column qualifier that can be used as an indexing mechanism within a "single" row element and thus, realize collection types. While using the API proposed in this article, the user does not need to care about the difference. When a column is for example specified to contain a set, the underlying differentiation between using Cassandra's collection type *set* and using HBase's additional column qualifiers for indexing the different values of a set is taken care of by the transformation layer (see

CloudDBGuard API	Cassandra	HBase
Row identifier column: explicit name and type	∼ (API uses explicit name/type)	+ (API enables explicit name/type)
Row identifier value: single field	− (API restricted to single field)	∼ (API uses single field)
Data types for columns	∼ (API uses data types)	+ (API enables data types)
Keyspace	Keyspace	Namespace
Table	Column family	Table
Column identifier	Column name	Column family
Collection types	Collection types	Column qualifier

Table 1: Functionalities of the CloudDBGuard API compared to Cassandra and HBase; ∼=functionality maintained, +=functionality increased, −=functionality restricted

section 4). Thus, even though this design aspect of the databases is fundamentally different, there are no restrictions or compromises regarding the use of collections, when using CloudDBGuard.

Data types. As already briefly mentioned in Section 2.3 Cassandra differentiates between a variety of data types, whereas HBase stores everything as byte array. This work will follow the HBase approach and store only byte arrays in encrypted columns in Cassandra as well. Except for OPE schemes, outcomes of all used encryption schemes are byte arrays anyway. Storing them as such avoids conversions back to their original data type and saves runtime. Only OPE ciphertexts have to be converted to byte arrays, which can be done fast. Columns that are not encrypted ("selective encryption") will use appropriate data types in Cassandra.

3.5. Interacting with the Databases

Using encryption requires additional efforts, when reading from and writing to the databases. The ciphertext name of a keyspace, a table or a column identifier is a randomly generated string with a length of 8 printable characters that is used to identify the corresponding element in queries. Due to the random generation no information about the plaintext name can leak. The mapping from plaintext to ciphertext name is only stored on the trusted client side. This section describes the steps that our API executes in detail.

3.5.1. Writing

Creating a keyspace involves the following steps:

1. For hiding the plaintext name of the keyspace, 8 characters are chosen randomly.
2. A metadata object representing the keyspace is created, containing the the mapping between the plaintext name and the ciphertext name and all other relevant metadata.
3. Appropriate queries are built and executed to actually create the keyspace on server side.

Creating a table and creating the individual columns involves the following steps:

1. For hiding the plaintext name of the table, 8 characters are chosen randomly. The same is done for every column. Note that depending on its datatype one plaintext column might result in multiple columns on server side, each of which represents a required onion. Thus, multiple ciphertext names per column might be necessary.
2. Metadata objects representing all columns of the table and the table itself are created.
3. Column keys for RND layer encryption as well as a table key and initialization vector IV for DET layer encryption are created and associated to the table's metadata.
4. A distribution profile can be set for this table (see [41]) that specifies how the individual columns are spread across the available database instances (at different cloud providers); this implements the separation of duties approach to increase security. The only exception is the row identifier column, which is written to every database in order to be able to join the data items again in the query process.
5. Appropriate queries are built and executed to actually create the (physical) table on server side.

PPE schemes are involved when actual data is written to the database, that is supposed to be queried later on. Inserting data requires two types of information – what is supposed to be inserted (the actual row data) and where is it supposed to be inserted (keyspace, table and column identifiers) – and then involves the following steps:

1. Given the plaintext names of keyspace and table for the new data items, the first step is to look up the corresponding ciphertext names in the metadata, as well as the database instances (type and IP address) involved in storing the table. Furthermore the keystore associated to the keyspace is loaded.
2. An IV for the RND layer encryption of the row is created and stored in the row.
3. Using this IV and (depending on the used PPE scheme) crypographic keys from the keystore all data items are encrypted in a property-preserving way according to the onion layer model (see Section 3.2). Thus, one plaintext data item may result in multiple ciphertext data items (each of which maybe the result of multiple encryptions).
4. Finally the write queries can be constructed (one per row and involved database instance) involving all previously collected information.

The impact on the runtime required by all these pre-processing steps is investigated further in Section 5.

3.5.2. Querying

When a client sends a query, the API executes the following steps to retrieve and decrypt results:

1. The query may contain a number of conditions that have to be met for a row to be included in the result set. These conditions are parsed to identify the columns that are involved in those conditions.
2. To check, if a condition is met by a column that was identified in the first step, it might be necessary to remove the RND layer from a particular onion to get to its underlying DET or OPE encrypted values. The metadata is asked if the RND layer on that column still exists; if so, it is removed.
3. Afterwards the set of all columns is identified that have to be read from the database(s). This set consists of two subsets: The first subset consists of all columns that are involved in query conditions; the second subset consists of all columns, that were selected by the user (and thus, are not necessarily involved in any query conditions). For the second subset, the DET onion is chosen, because it is the fastest one to be decrypted later on. For the first subset – depending on the type of condition – the appropriate ciphertext column is selected (e.g. the OPE onion, if the condition it is involved in is an order comparison).
4. Furthermore, there are two columns, that are always read from the database, independent of the query: the row identifier column (needed to address rows in the result set later and to join result sets from multiple database instances) and the IV column (needed for further RND layer decryption on columns, that were never involved in query conditions).
5. After all necessary columns are identified, the metadata is used to look up the database instances responsible for storing them. One query is constructed for each database instance. In every query the plaintext keyspace, table and column identifiers are replaced by their ciphertext counterparts. In addition, all terms in conditions are replaced by their PPE encrypted equivalents. In the end no query contains any plaintext information anymore and can be executed.
6. If multiple database instances were involved in the initial query, the rows of their individual result sets are now joined using the row identifier column.
7. As a last step the final result set is decrypted using the DET onion, as discussed in Step 3.

4. CloudDBGuard Architecture and Implementation

Implementation was done in Java, including all PPE schemes, API methods and database communication. Since disk access and memory management in WCSs are performed at column level, the indexes of all PPE schemes are designed at column level, too. That means every column that uses a PPE scheme relying on

an index, gets its own index. If a column is part of a filter condition specified in the query, only the index information of that column has to be taken into account when processing that query.

The Cassandra Java Driver is used to interact with Apache Cassandra, which allows utilizing the current version 3 series of the Cassandra Query Language (CQL); CQLv3 is the first version of CQL, that explicitly supports collections. Internally it creates a key-value-pair for every item of a set with the item as key and null as value. Querying Cassandra can be done either by directly passing query strings to the driver or using the integrated query builder.

In contrast, HBase does not provide a high level query language. The fastest way to interact with it is its native Java API. All operations are performed by creating put-, get-, or delete-objects first, equipping them with appropriate row filters that correspond to query conditions (if desired) and handing them over in form of "scanners" to objects, that represent tables and keyspaces.

Persistent storage on client side is mainly needed for two things. On the one hand, OPE schemes need to store their indexes here. For that purpose it is sufficient to just save the serialized representation of the corresponding data structures in files. On the other hand, there is the metadata of the encrypted columns, which is stored in XML representation.

This section discusses the data flow and involved components of CloudDBGuard's architecture in detail. As can be seen in Figure 2 the data passes three layers on its way from the application to the database: the application layer, the encryption layer and the transformation layer. Each layer serves individual tasks.

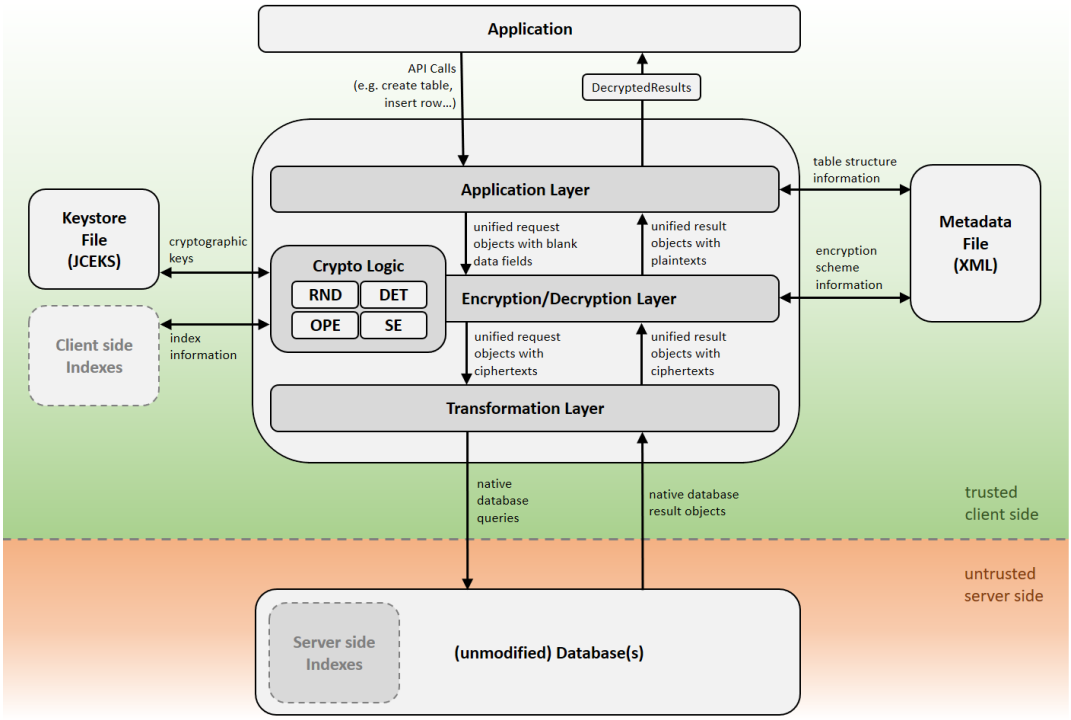

Figure 2: Architecture of CloudDBGuard

4.1. Application Layer and Unified Request Objects

Every interaction with the database is initiated by calling one of the available methods provided by the application layer. Thus, this layer fulfills the following tasks:

- It provides a unified way of querying the supported databases. The details of the database's native query languages are no longer of concern for the user. Therefore the application layer creates "uni-

fied request objects" (UROs) for every interaction with the database. These UROs describe exactly, what is supposed to happen (using their `Type` field, e.g. `CREATE_TABLE, INSERT_ROW` etc.), where it is supposed to happen (using their `DBLoc` field, short for DBLocation) and what data is involved (`String/Int/Byte[_Set]_Args`). A URO has one more field, that stores a reference to the row identifier column, which is important for some query types. As the name suggests, `DBLoc` is a data structure, that precisely describes a location within the database, consisting of a keyspace, a table name and a pattern restricting rows or columns. For example, when creating keyspaces only the keyspace name is of interest and all other values can be set to `null`. In contrast, when querying the database, the keyspace and table names have to be provided, as well as the selected columns and row restricting conditions (for example in range queries). The use of the `String/Int/Byte[_Set]_Args` is very versatile. They can contain nothing or very few data – e.g. when creating a keyspace – or be filled with values – when actually inserting rows of data into tables.

- It takes care of the client side metadata, which consists of two subtasks. Firstly, it always synchronizes the metadata. If new keyspaces, tables or only columns are created or deleted, the metadata has to be updated. This is important for knowing the location of actual data and for translating plaintext UROs into their ciphertext counterparts later on. Secondly, it stores the metadata persistently on client side, and loads/saves it to an XML file whenever necessary.
- When data is returned from the database(s) it makes the decrypted results available to the requesting application in an easy and unified way.

4.2. Encryption Layer

The UROs returned by the application layer still contain plaintext data and plaintext metadata that must not leak to the database. Thus, when querying the database it is the purpose of the encryption layer to replace all sensitive information within a URO with the corresponding ciphertext data. That encompasses mainly two subtasks:

- All URO data that describes a location within the database is replaced by the appropriately mapped ciphertext names read from the client side metadata.
- All plaintext data items within the URO are replaced by their corresponding onion layer ciphertexts. Information about the state of the onions are read from the metadata as well.

When data is returned from the database, the encryption layer decrypts these ciphertexts.

4.3. Transformation Layer

After having received the UROs with ciphertexts from the encryption layer it is now the task of the transformation layer to translate the UROs into the databases' native query mechanisms and actually connect to and interact with them. In case of Apache Cassandra that means composing and issuing CQL queries; in case of HBase it means executing the corresponding calls of the native HBase Java API. The transformation layer also receives the database's individual result objects and makes them accessible in a unified way for the encryption layer. Note that the transformation layer is the only part of the entire architecture, that deals with individual database details. Thus, it is also the only part, that has to be changed, to support other database systems – only two Java classes have to be implemented in this case.

5. Benchmark

All experiments in this section were run on an Intel Core i5 CPU@2.30GHz, 16GB RAM, an APPLE SSD AP1024M 1TB SSD using macOS 10.14.5. as a client platform and CentOS 7 on a Parallels Desktop virtual machine with 4 vCPUs and 4GB of vRAM. The PPE schemes were implemented in Java, using cryptographic primitives of the Java Cryptography Extension and The Legion of the Bouncy Castle Java Cryptography API. In order to avoid measuring network effects local installations of the databases were

Query	Types of schemes	Queried Columns	Query	Types of schemes	Queried Columns
Q0	DET	receiver	Q5	DET + SE	sender, body
Q1	OPE	timestamp	Q6	OPE + OPE	sender, timestamp
Q2	SE	sender	Q7	OPE + SE	timestamp,body
Q3	DET + DET	sender, receiver	Q8	SE + SE	body, subject
Q4	DET + OPE	sender, timestamp	Q9	DET + OPE + SE	sender, timestamp, subject

Table 2: Types of generated queries

used, as only the computation time of the schemes in combination with the speed of the databases was to be measured.

The widely-used Enron corpus [15] served as our data set. The Enron dataset contains the e-mails of 150 employees of the Enron company. We parsed a random subset of 10 000 mails to simulate an average sized mailbox as well as a random subset of 100 000 mail to simulate a larger mailbox and created one database row for each mail. This also includes that the result sizes where scaled due to more matching database records. Our benchmark queries select the primary key (the mail identifier) of these rows meeting certain conditions.

We aim to benchmark the performance of different categories of property-preserving encryption schemes (deterministic, order-preserving and searchable encryption) and their combination. In this way we go beyond prior work [7, 8, 9, 10] where we only benchmarked each category in isolation. We wrote queries for each type of schemes on its own (Q0-Q2), the six possible combinations of two types (Q3-Q8) and the combination of all the three types (Q9). Details about the columns queried can be found in Table 2. Each of the 10 queries was run ten times with randomly generated conditions on the queried columns. We tested all queries with three table profiles [41] for which we identified in previous bencharks the best performing encryption methods for the following three use cases: (1) `STORAGE-EFFICIENT` (OPE layer columns are encrypted using mOPE, SE layer columns are encrypted using SWP), (2) `OPTIMIZED WRITING` (the OPE scheme best suited for fast writing is OACIS, as long as pre-sorted inputs are avoided, for the SE layer the SWP scheme is used) (3) `OPTIMIZED READING` (the OPE scheme best suited for fast reading is RSS, for the SE layer the SUISE scheme is used).

Figures 3 and 4 show the results as the average over 10 runs for the dataset sizes of 10k and 100k emails. We measured the overhead on the client side which includes translating the query into its encrypted equivalent and obtaining the decrypted result (see Section 4). In particular, the steps measured on client side are:

- mapping each column involved in the query, the plaintext table names and keyspace names to the randomized names;
- mapping each query condition on some column into an encrypted variant based on the comparator contained in the condition;
- translating the API query into the query language of the underlying database system;
- setting up a thread pool for asynchronously querying the affected database systems;
- as well as decrypting the result set and returning it to the client application as a result object.

Moreover, we measured the DB communication as the roundtrip time between issuing the (already encrypted) query to the database and retrieving the (still encrypted) results.

As can be seen, the query times remain within acceptable time spans of only a few seconds. With respect to DB communication, HBase (less than 0.8 seconds maximum per query) is consistently faster than Cassandra (less than 1.6 seconds maximum). This reflects the fact that Cassandra is optimized for writing while HBase is optimized for reading. Furthermore, searchable encryption is by far the most expensive type of property preserving encryption, as all queries involving it take the most time. It has a strong impact,

especially when the data fields are large (like in Q5, Q7 and Q8, which involve performing SE on mail bodies). In particular, due to randomly chosen query conditions we see an increased runtime of the client-side result transformation for queries Q5 and Q7 when using HBase in those cases where no matching mail body could be found: queries without a matching body took roughly 5 seconds in total while queries with a match took less than 0.5 seconds. We aim to optimize this client-side API behavior in the future. In contrast, deterministic and order-preserving encryption have less impact on runtime.

Regarding the scalability of our system we observed only a slight overhead due to the increased result sizes when comparing the 10 000 emails dataset with the 100 000 emails dataset; the maximum time difference was observed for Q3 in Cassandra for the storage efficient profile where query execution took 2.3 times as long for the larger data set as for the smaller one.

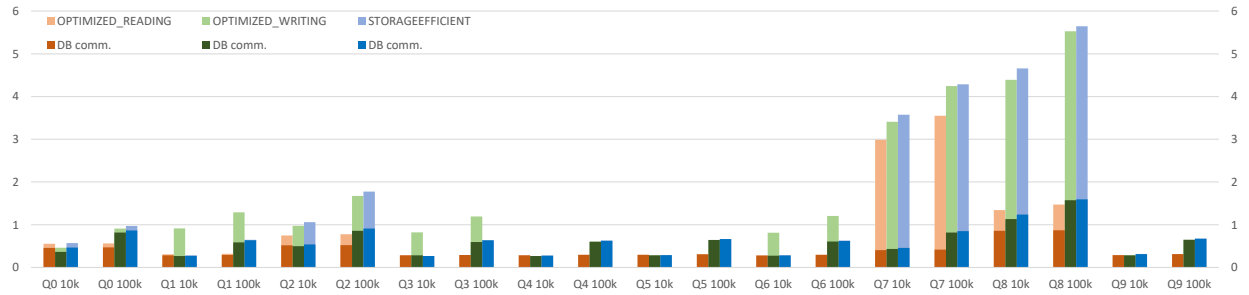

Figure 3: Query Performance with Cassandra including DB communication (lower part of each bar)

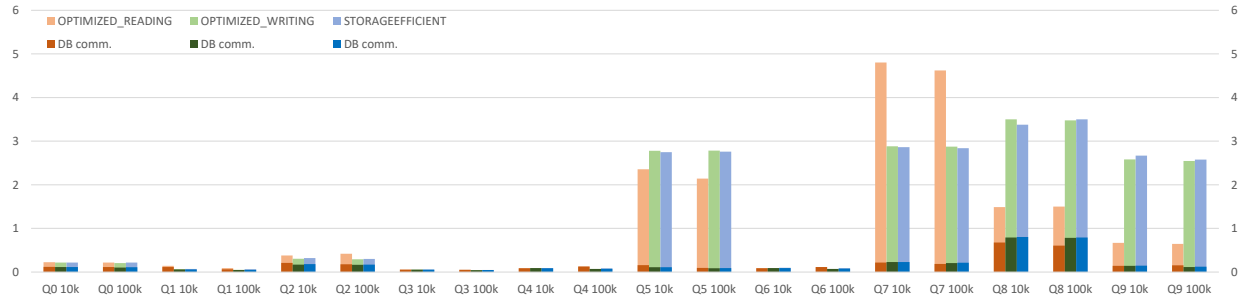

Figure 4: Query Performance with HBase including DB communication (lower part of each bar)

6. Conclusion and Future Work

We presented the CloudDBGuard framework that extends our prior work on real-world applications of PPE schemes. Its functionality was wrapped into an easy-to-use API that hides the complexity of property-preserving encryption. The user can simply insert a plaintext record into the database by calling the insertRow method and later on retrieve data by calling the query method that just requires the plaintext column names to be returned as well as some plaintext query conditions. The user is not bothered by the cryptographic settings and processes: key generation, key management, encryption and decryption of both data records and query strings are totally transparent for the user.

Most importantly the API also hides the native interfaces of the underlying database from the user. Very different native query mechanisms (CQL of Cassandra, the native Java API of HBase) are unified in an easy-to-use API. Access to all decrypted query results can be obtained using an iterator or even by directly addressing a value by its row identifier and column identifier no matter what underlying database was used – this is an additional functionality that is not offered in the native resultset objects of Cassandra

and HBase. Other database systems can be integrated as well by implementing a single abstract Java class, most likely resulting in no more than 300 lines of code. The database server(s) remain unmodified. That means the approach of this work can be easily extended to several other databases (e.g. those hosted by cloud database providers). Because a client application talks directly to the database server(s) via the API, this avoids extra network traffic and reduces latencies. Furthermore no specialized cryptographic hardware is necessary.

The user can optimize the performance by selecting PPE schemes based on profiles for certain use cases ("table profiles") or exclude non-sensitive data columns from encryption ("selective encryption"). In particular, by utilizing SUISE [26] for SE as well as OACIS [23] and RSS [22] for OPE the architecture proposed in this article takes advantage of index-based PPE schemes, that lead to a better performance for querying larger datasets. Simple interfaces can be implemented to support further databases and PPE schemes.

Database reconstruction attacks [42, 43] based on statistics (on the ciphertext or query behavior) are one major threat to property-preserving encryption; by construction any property-preserving encryption scheme is prone to such attacks based on the properties leaked in the ciphertext. Some more or less straight-forward workarounds, which have been proposed in the literature for several approaches, are to perturb statistics of the ciphertext by 1) distorting the underlying distribution with dummy entries or 2) choose a proper granularity of the data on which PPE is applied (e.g. only encrypting the year information with a PPE scheme instead of an entire timestamp down to the level of seconds). Those workarounds clearly come at the cost of extra post-filtering and/or post-processing on the client side. Another option to rule out statistical attacks can be based on a combination with other techniques. In our framework, we combined PPE with a distribution approach (separation of duties [12]) to increase security: tables can be spread across a set of independent database instances (consisting even of different database types) to increase security and minimize the threat of statistical attacks. Our prototype supports seamlessly such a distribution of the outsourced data on multiple providers when creating new keyspaces.

The area of processing queries over encrypted data in WCSs leaves room for further research. We aim to optimize query behavior of our system in future work ; we plan to include recent developments in the development of cryptographic algorithms by incorporating them into our framework as well. Moreover, the CloudDBGuard framework can be extended to support aggregations on server side in order to support the mathematical functions available in HBase and Cassandra on numerical data. In future work we aim to compare a choice of homomorphic encryption methods and identify the one with the best properties for that purpose.

Acknowledgements. This work was funded by the DFG under grant number Wi 4086/2-2.

References

References

[1] T. O'Reilly, What is Web 2.0: Design patterns and business models for the next generation of software, Communications and Strategies 65 (1) (2007) 17–37.
[2] T. Haerder, A. Reuter, Principles of transaction-oriented database recovery, ACM Computing Surveys (CSUR) 15 (4) (1983) 287–317.
[3] E. Brewer, A certain freedom: thoughts on the CAP theorem, in: Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, ACM, 2010, pp. 335–335.
[4] L. Wiese, Advanced Data Management for SQL, NoSQL, Cloud and Distributed Databases, DeGruyter, 2015.
[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber, Bigtable: A distributed storage system for structured data, ACM Transactions on Computer Systems (TOCS) 26 (2) (2008) 4.
[6] A. Lakshman, P. Malik, Cassandra: a decentralized structured storage system, ACM SIGOPS Operating Systems Review 44 (2) (2010) 35–40.
[7] T. Waage, L. Wiese, Benchmarking encrypted data storage in hbase and cassandra with ycsb, in: International Symposium on Foundations and Practice of Security, Springer, 2014, pp. 311–325.
[8] T. Waage, R. S. Jhajj, L. Wiese, Searchable encryption in Apache Cassandra, in: International Symposium on Foundations and Practice of Security, Springer, 2015, pp. 286–293.

[9] T. Waage, D. Homann, L. Wiese, Practical Application of Order-Preserving Encryption in Wide Column Stores, in: Proceedings of the 13th International Joint Conference on e-Business and Telecommunications - SECRYPT, 2016, pp. 352–359.
[10] T. Waage, L. Wiese, Property preserving encryption in nosql wide column stores, in: OTM Confederated International Conferences, Springer, 2017, pp. 3–21.
[11] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, et al., Apache Hadoop goes realtime at Facebook, in: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, ACM, 2011, pp. 1071–1080.
[12] F. Bollwein, L. Wiese, Keeping secrets by separation of duties while minimizing the amount of cloud servers, in: Transactions on Large-Scale Data-and Knowledge-Centered Systems XXXVII, Springer, 2018, pp. 1–40.
[13] F. Bollwein, L. Wiese, Separation of duties for multiple relations in cloud databases as an optimization problem, in: Proceedings of the 21st International Database Engineering & Applications Symposium, ACM, 2017, pp. 98–107.
[14] S. Mehrotra, S. Sharma, J. D. Ullman, A. Mishra, Partitioned data security on outsourced sensitive and non-sensitive data, arXiv preprint arXiv:1812.09233.
[15] B. Klimt, Y. Yang, The enron corpus: A new dataset for email classification research, in: Machine learning: ECML 2004, Springer, 2004, pp. 217–226.
[16] O. Goldreich, Foundations of Cryptography: Basic Applications, Vol. 2, Cambridge university press, 2004.
[17] C. Papamanthou, R. Tamassia, N. Triandopoulos, Authenticated hash tables based on cryptographic accumulators, Algorithmica 74 (2) (2016) 664–712.
[18] D. Pennino, M. Pizzonia, F. Griscioli, Pipeline-integrity: Scaling the use of authenticated data structures up to the cloud, Future Generation Computer Systems 100 (2019) 618–647.
[19] A. Khetrapal, V. Ganesh, HBase and Hypertable for large scale distributed storage systems, Dept. of Computer Science, Purdue University (2006) 22–28.
[20] S. M. Sawyer, B. D. O'Gwynn, A. Tran, T. Yu, Understanding query performance in Accumulo, in: High Performance Extreme Computing Conference (HPEC), 2013 IEEE, IEEE, 2013, pp. 1–6.
[21] A. Boldyreva, N. Chenette, A. ONeill, Order-preserving encryption revisited: Improved security analysis and alternative solutions, in: Advances in Cryptology–CRYPTO 2011, Springer, 2011, pp. 578–595.
[22] S. Wozniak, M. Rossberg, S. Grau, A. Alshawish, G. Schaefer, Beyond the ideal object: towards disclosure-resilient order-preserving encryption schemes, in: Proceedings of the 2013 ACM workshop on cloud computing, ACM, 2013, pp. 89–100.
[23] F. Kerschbaum, A. Schröpfer, Optimal average-complexity ideal-security order-preserving encryption, in: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2014, pp. 275–286.
[24] N. Chenette, K. Lewi, S. A. Weis, D. J. Wu, Practical order-revealing encryption with limited leakage, in: International Conference on Fast Software Encryption, Springer, 2016, pp. 474–493.
[25] R. Curtmola, J. Garay, S. Kamara, R. Ostrovsky, Searchable symmetric encryption: improved definitions and efficient constructions, in: Proceedings of the 13th ACM Conference on Computer and Communications Security, ACM, 2006, pp. 79–88.
[26] F. Hahn, F. Kerschbaum, Searchable encryption with secure and efficient updates, in: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2014, pp. 310–320.
[27] D. X. Song, D. Wagner, A. Perrig, Practical techniques for searches on encrypted data, in: Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on, IEEE, 2000, pp. 44–55.
[28] R. A. Popa, C. Redfield, N. Zeldovich, H. Balakrishnan, CryptDB: processing queries on an encrypted database, Communications of the ACM 55 (9) (2012) 103–111.
[29] F. Shahzad, W. Iqbal, F. S. Bokhari, On the use of CryptDB for securing electronic health data in the cloud: A performance study, in: 2015 17th International Conference on E-health Networking, Application & Services (HealthCom), IEEE, 2015, pp. 120–125.
[30] S. D. Tetali, M. Lesani, R. Majumdar, T. Millstein, Mrcrypt: Static analysis for secure cloud computations, ACM SIGPLAN Notices 48 (10) (2013) 271–286.
[31] I. H. Akin, B. Sunar, On the difficulty of securing web applications using CryptDB, in: Big Data and Cloud Computing (BdCloud), 2014 IEEE Fourth International Conference on, IEEE, 2014, pp. 745–752.
[32] S. Tu, M. F. Kaashoek, S. Madden, N. Zeldovich, Processing analytical queries over encrypted data, Proceedings of the VLDB Endowment 6 (5) (2013) 289–300.
[33] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, S. Bellovin, Blind seer: A scalable private DBMS, in: 2014 IEEE Symposium on Security and Privacy, IEEE, 2014, pp. 359–374.
[34] M. I. Sarfraz, M. Nabeel, J. Cao, E. Bertino, Dbmask: fine-grained access control on encrypted relational databases, in: Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, ACM, 2015, pp. 1–11.
[35] L. Ferretti, M. Colajanni, M. Marchetti, Distributed, concurrent, and independent access to encrypted cloud databases, IEEE transactions on parallel and distributed systems 25 (2) (2014) 437–446.
[36] S. Kamara, T. Moataz, Sql on structurally-encrypted databases, in: International Conference on the Theory and Application of Cryptology and Information Security, Springer, 2018, pp. 149–180.
[37] R. Poddar, T. Boelter, R. A. Popa, Arx: A strongly encrypted database system, IACR Cryptology ePrint Archive 2016 (2016) 591.
[38] W. Zheng, F. Li, R. A. Popa, I. Stoica, R. Agarwal, Minicrypt: Reconciling encryption and compression for big data stores, in: Proceedings of the Twelfth European Conference on Computer Systems, ACM, 2017, pp. 191–204.
[39] R. A. Popa, C. Redfield, N. Zeldovich, H. Balakrishnan, CryptDB: protecting confidentiality with encrypted query pro-

cessing, in: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, ACM, 2011, pp. 85–100.
[40] D. Liu, S. Wang, Programmable order-preserving secure index for encrypted database query, in: Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on, IEEE, 2012, pp. 502–509.
[41] T. Waage, L. Wiese, CloudDBGuard: Enabling sorting and searching on encrypted data in nosql cloud databases, in: International Conference on Big Data Analytics and Knowledge Discovery, Springer, 2018, pp. 261–270.
[42] E. A. Markatou, R. Tamassia, Full database reconstruction with access and search pattern leakage., IACR Cryptology ePrint Archive 2019 (2019) 395.
[43] P. Grubbs, M. Lacharité, B. Minaud, K. G. Paterson, Learning to reconstruct: Statistical learning theory and encrypted database attacks, IACR Cryptology ePrint Archive 2019 (2019) 11.

Lena Wiese is a member of the L3S Research Center Hannover. She also leads the research group "Knowledge Engineering" (at the Institute of Computer Science, University of Goettingen). She holds a PhD and a Master degree from TU Dortmund. After her PhD she worked as a postdoctoral researcher at the National Institute of Informatics in Tokyo and as a visiting lecturer at the University of Hildesheim and the University of Salzburg. Dr. Wiese is author of the text book [4] on Advanced Data Management. Her research interests cover Intelligent Data Management as well as IT Security. She is an active member of the German Informatics Society (GI) and regularly acts as a reviewer for conferences and journals.

Tim Waage holds a Master and a PhD degree in Computer Science from the University of Goettingen. He was an IT Coordinator at the University Medical Center Goettingen (UMG). Dr. Waage now works for a medical technology company.

Michael Brenner holds a PhD degree from Leibniz University Hannover. Dr. Michael Brenner is deputy head of the department of Communication Systems at the Leibniz University IT Services. At the Department of Computational Health Informatics (CHI) he is responsible for the area of encrypted computing. Dr. Brenner is co-founder of the workshop series "Encrypted Computing and Applied Homomorphic Cryptography" held in cooperation with the international Conference on Computer and Communications Security (CCS) of the ACM. He holds a patent on "Encrypted Computing" and is member of a standardization consortium for homomorphic cryptography. Dr. Brenner also has 10 years experience as a software architect and manager of software development projects in the financial industry.