A Framework for Property-preserving Encryption in Wide Column Store Databases

Dissertation

zur Erlangung des Doktorgrades
Doctor rerum naturalium (Dr. rer. nat.)
der mathematisch-naturwissenschaftlichen Fakultäten
der Georg-August-Universität zu Göttingen

im Promotionsprogramm Computer Science (PCS)
der Georg-August-University School of Science (GAUSS)

vorgelegt von

Tim Waage
aus Nordhausen

Göttingen
im Mai 2017

Betreuungsausschuss

Erster Betreuer	Dr. Lena Wiese Institut für Informatik, Georg-August-Universität Göttingen
Zweiter Betreuer	Prof. Dr. Carsten Damm Institut für Informatik, Georg-August-Universität Göttingen

Prüfungskommision

Referent	Dr. Lena Wiese Institut für Informatik, Georg-August-Universität Göttingen
Koreferent	Prof. Dr. Carsten Damm Institut für Informatik, Georg-August-Universität Göttingen

weitere Mitglieder

Prof. Dr. Marcus Baum
Institut für Informatik, Georg-August-Universität Göttingen

Prof. Dr. Jens Grabowski
Institut für Informatik, Georg-August-Universität Göttingen

Prof. Dr. Dieter Hogrefe
Institut für Informatik, Georg-August-Universität Göttingen

Prof. Dr. Stephan Waack
Institut für Informatik, Georg-August-Universität Göttingen

Tag der mündlichen Prüfung: 05.07.2017

Abstract

While traditional database software usually relies on the relational SQL model, a lot of alternative approaches, commonly referred to as NoSQL (short for "Not only SQL") databases, occurred within the last years to meet the new requirements of the so called "Web 2.0" services, that are hard to achieve with SQL based systems, especially in terms of availability, partition tolerance and scalability.

Nowadays, cloud storage providers widely utilize a particular sub-category of NoSQL databases, namely wide column stores, for outsourcing data, whether it is for backups or reducing operational costs. Unfortunately security was not a primary concern of the NoSQL systems designers. In most cases some sort of front end is assumed to take care of authentication, user management, etc. Hence these remote cloud servers are not trustworthy when it comes to storing sensitive data, since at least the administrators have full access. However, data confidentiality can still be achieved by using encryption before outsourcing the data, but that limits the options for interaction, since the encrypted data lacks properties of the plaintext data that the database systems rely on. For instance equality checks or order comparisons are either not possible any more or lead to wrong results.

Various schemes have been proposed for property-preserving encryption in order to overcome these issues, allowing a database to process queries over encrypted data. Only very few of them can be considered feasible or have ever been practically implemented. Beside rare exceptions none of them have been tested with existing database systems at all.

Hence, this thesis presents a simple to use a application programming interface that allows using property-preserving encryption in unmodified wide column stores. Therefore appropriate schemes have been identified and implemented. It hides the complexity of the encryption and decryption process and allows various adjustments on concrete use cases in order to achieve a maximum of security, functionality and performance.

Zusammenfassung

Während traditionelle Datenbankanwendungen oftmals auf relationalen Tabellen und SQL basieren, entstanden in den letzten Jahren viele alternative Ansätze, die häufig unter dem Sammelbegriff NoSQL (kurz für "Not only SQL") zusammengefasst werden. Ihr Bestreben ist es hauptsächlich den Anforderungen moderner "Web 2.0" Services gerecht zu werden, die mit SQL Datenbanken nur schwer erreicht werden können, vor allem in punkto Hochverfügbarkeit, Partitionstoleranz und Skalierbarkeit.

Heutzutage nutzen Cloud-Speicher Anbieter insbesondere NoSQL Technologien einer bestimmten Kategorie, namentlich der sogenannten Spaltenfamiliendatenbanken. Sie ermöglichen die Auslagerung von Daten in die Cloud, sei es für Backups oder einfach um die eigenen laufenden (Server-)Kosten zu senken. Leider wurden diese Technologien jedoch nicht unter Sicherheitsaspekten konzipiert. So wird in der Regel angenommen, dass der Datenbank ein Front-End vorgeschaltet ist, welches sich um Authentifizierung, Nutzerverwaltung etc. kümmert. Der entfernte NoSQL Datenbankserver ist also erst einmal nicht vertrauenswürdig, wenn es darum geht sensible Daten zu speichern. So können beispielsweise dessen Administratoren den Datenbestand ungehindert einsehen. Einen wirksamen Schutz bietet in diesem Fall die Verschlüsselung der Daten noch bevor diese in der Datenbank abgelegt werden. Leider büßt die Datenbank damit aber auch Funktionalität ein, denn den Schlüsseltexten mangelt es in der Regel an bestimmten Eigenschaften, welche die Datenbank zur Beantwortung von Anfragen benötigt. So ändert sich beispielsweise die Ordnungsrelation von numerischen Werten.

Es gibt jedoch viele Ansätze zur Durchführung von Verschlüsselung in einer Art und Weise, die diese Eigenschaften erhält und somit die Datenbank trotz Verschlüsselung weiterhin zur Beantwortung von Anfragen befähigt. Nur wenige davon sind jedoch praktisch einsetzbar. Implementationen, die auch eine Nutzbarkeit im Zusammenspiel mit den Datenbanksystemen bringen, existieren kaum (bzw. in Kontext mit NoSQL Datenbanken gar nicht).

Die vorliegende Arbeit präsentiert daher eine benutzerfreundliche Programmierschnittstelle, die es erlaubt eigenschaftsbewahrende Verschlüsselung in unmodifizierten Spaltenfamiliendatenbanken einzusetzen. Dafür wurden geeignete Ansätze identifiziert und implementiert. Die Komplexität der Ver- und Entschlüsselungsprozesse bleibt dem Nutzer dabei verborgen, umfangreiche Möglichkeiten zur Anpassung an konkrete Nutzungsszenarien sind aber trotzdem vorhanden, um ein Maximum an Sicherheit, Funktionalität und Geschwindigkeit zu erzielen.

Contents

List of Figures

List of Tables

Acronyms

ACID	Atomicity, Consistency, Isolation, Durability
AES	Advanced Encryption Standard
AMI	Amazon Machine Image
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
AWS	Amazon Web Services
CAP	Consistency, Availability, Partition Tolerance
CBC	Cipher Block Chaining
CCA	Chosen Ciphertext Attack
CKA	Chosen Keyword Attack
CPA	Chosen Plaintext Attack
CQL	Cassandra Query Language
DBMS	Database Management System
DES	Data Encryption Standard
DET	Deterministic Encryption
EMR	Elastic MapReduce
FHE	Fully Homomorphic Encryption
FKS	Fredman, Komlos and Szemeredi
FPE	Format-preserving Encryption

FPGA	Field Programmable Gate Array
GSM	Global System for Mobile Communications
HDFS	Hadoop Distributed File System
IaaS	Infrastructure-as-a-Service
IT	Information technology
IV	Initialization Vector
JCA	Java Cryptography API
JCE	Java Cryptography Extension
JCEKS	Java Cryptography Extension Key Store
mOPE	modular Order-preserving Encryption
NoSQL	Not only SQL
OACIS	Optimal Average-Complexity Ideal-Security
ODBC	Open Database Connectivity
OCPA	Ordered Chosen Plaintext Attack
OPE	Order-preserving Encryption
OPF	Order-preserving Function
P2P	Peer-to-Peer
PaaS	Platform-as-a-Service
POPF	Pseudo Random Order-preserving Function
PPE	Property-preserving Encryption
PRF	Pseudo Random Function
PRG	Pseudo Random Generator
RAM	Random Access Memory
RC4	Ron's Code 4
RC6	Ron's Code 6
RND	Random Encryption
ROA	Random Offset Addition

RSS	Random Subrange Selection
RUS	Random Uniform Sampling
SaaS	Software-as-a-Service
SCPU	Secure Co-Processor Unit
SDK	Software Development Kit
SE	Searchable Encryption
SEM	Semantic Security
SQL	Structured Query Language
SSH	Secure Shell
SUISE	Securely Updating Index-based Searchable Encryption
SWIFT	Society for Worldwide Interbank Financial Telecommunication
SWP	Song, Wagner and Perrig
TPC	Transaction Processing Performance Council
TTL	Time-to-Live
UDF	User Definded Function
URO	Unified Request Object
WCS	Wide Column Store
WOW	Window One-Wayness
XML	eXtensible Markup Language
XOR	eXclusive OR

1 Introduction

This chapter presents the problem statement and summarizes the contributions of this thesis. It then lists the author's publications, the majority of which presents intermediate results. Finally it outlines the structure of the thesis.

Contents

1.1. Problem and Motivation

In times of "Big Data" and the "Web 2.0" [67] traditional SQL-based database services have their difficulties with the changing demands. On the one hand there are new performance requirements. While SQL environments usually focus on being ACID [44] compliant, it is much more important for modern web services to deliver high availability, consistency and (since they naturally run in distributed environments) to be tolerant regarding network partitions in the underlying infrastructure [20]. On the other hand there are different demands regarding the data structures themselves. SQL tables are not well suited to represent loosely structured data items like they are typical for today's webservices.

As NoSQL (Not only SQL) databases [46, 96] were designed for meeting those new requirements, they attracted more and more attention over the last years. Many of the global players on the market developed their own solutions, especially in the sub-category of so called wide column stores (WCS). Examples like Google's Bigtable [24](used for instance in the Google search engine, Google Maps, Google Earth, Youtube) or Facebook's Cassandra [57] (used for instance in Twitter, Reddit and Facebook itself until 2011) show, that almost everybody uses services in their daily life, that heavily utilize NoSQL WCSs.

Nowadays it is also common to use WCSs directly. Countless cloud database providers offer flexible on-demand services for running Cassandra or other WCS databases remotely in their clusters. Popular examples are Google Cloud Platform, Microsoft Azure, Amazon EC2 (Amazon Web Services) and Rackspace. All of them even simplify the usage of WCSs by providing web interfaces or preconfigured images. Thus, there are a lot of easily usable options for outsourcing data to clouds services based on WCSs.

The benefits of doing so are well known. In particular, outsourcing database hosting to a cloud service can lead to lower costs by increasing the utilization of an IT infrastructure and sharing administrative staff. Various scenarios profit, starting from business, research and health care applications to social media and government scenarios. Hence the goal of many organizations today is to push as much data and as many computations into the cloud as possible.

However, storing and processing sensitive data on infrastructures provided by a third party increases the risk of unauthorized disclosure if the infrastructure is compromised by an adversary, which is a key problem. There are numerous examples for such sensitive data, for instance business secrets, credit card numbers, or personal health records. Thus, theft of this data is a serious problem. The fact that WCSs usually lack security features like authentication or user (rights) management exacerbates it. WCSs have not been designed having security aspects in mind. Instead an external front end is assumed to take care of such tasks.

Thus, despite the increasing adoption of cloud database technology, significant challenges still exist. So far the majority of such services requires their customers to absolutely trust the provider with full access to the outsourced data. Hence there is a strong need for providing security and privacy guarantees in those environments.

There are several ways of how sensitive data can be leaked. A data owner might trust a cloud provider to correctly operate provisioned services, but may not trust the employees of the cloud provider to keep data confidential. In certain cases, a data owner might not even trust their own employees who run a private cloud. Three main cases can be distinguished:

- An adversary can exploit software vulnerabilities to gain unauthorized access to servers[1].
- Curious or malicious administrators at a hosting provider can snoop on private data.
- Attackers with physical access to servers can steal data from disk or even memory [4].

Such scenarios are not theoretical, they frequently happen on small and large scale. Some popular examples that have made it into the press over the last years are:

- *2015/16: SWIFT banking hack* - A malware issued unauthorized SWIFT messages and concealed that they had been sent. After that it manipulated the database records of the transfers[2].
- *2014: Sony Pictures Entertainment hack* - A hacker group naming itself "Guardians of Peace" leaked confidential data from the film studio Sony Pictures Entertainment, including personal information of its employees and their families, e-mails between employees, salary information and even copies of films that had not been released yet[3].
- *2011: Sony Playstation Network hack* - Attackers gained access to about 77 million personal user profiles. Besides personal data most of them also included credit card information[4].

[1]NIST National Vulnerability Database. http://nvd.nist.gov (all URLs have been checked on May 19th 2016)

[2]Michael Corkery, New York Times (12/05/2016). "Once Again, Thieves Enter Swift Financial Network and Steal". http://www.nytimes.com/2016/05/13/business/dealbook/swift-global-bank-network-attack.html

[3]Mike Lennon, Security Week (19/12/2014). "Hackers Used Sophisticated SMB Worm Tool to Attack Sony". http://www.securityweek.com/hackers-used-sophisticated-smb-worm-tool-attack-sony

[4]Quinn, B., Arthur, C., The Guardian (26/04/2011). "Playstation network hackers access data of 77 million users" https://www.theguardian.com/technology/2011/apr/26/playstation-network-hackers-data

- *2010: German Tax Evasion Relevation* - Database administrators of several Swiss banks sold customer information to German and French tax authorities[5].

One possible solution to reduce the damage caused by server compromises is encrypting the data on a trusted client machine before it gets uploaded to the cloud servers, then processing queries by reading back the encrypted data from the server to the client, decrypting it, and executing the query on the client machine. However, this approach comes with a few downsides. Firstly, it requires transferring much more data than necessary. Typically large fractions of a database are read in order to create relatively small data aggregations. Secondly, many applications require servers to not just store data, but also perform computations on the data. Existing approaches (see Section 3.1) made use of property-preserving encryption (PPE) schemes (see Sections 2.4 and 4.1) to get rid of these problems, but they still leave issues to solve, for example

- bad efficiency in large datasets (the used encryption technologies do not scale well)
- no support for data distribution over multiple databases (mostly only one SQL based system is supported)
- bad security properties (the used encryption technologies are not state of the art)
- bad feasibility (most approaches still have the state of being research prototypes or being not even practically implemented)

Furthermore, a problem of cryptography in general is the additional effort necessary to use it. A user-friendly way to make it feasible in practice is the basic foundation for a broad adoption. Confronting the user directly with cryptographic algorithms has to be avoided.

1.2. Thesis Contributions

In order to overcome the problems described above, this thesis makes the following contributions:

- It identifies the requirements for utilizing property-preserving encryption PPE schemes in NoSQL WCSs.

[5]David Crawford, Vanessa Fuhrmans, Deborah Ball. Wall Street Journal (07/02/2010). "Germany Tackles Tax Evasion." `http://www.wsj.com/articles/SB10001424052748704197104575051480386248538`

- Based on these requirements it evaluates the practical strengths, weaknesses and feasibility of popular PPE schemes and proposes modifications to improve them (see Section 4.2 - 4.4).
- It introduces an easy to use Java Framework called "FamilyGuard" (see Section 5.6) for using PPE with WCSs, based on own implementations (see Chapter 6). The encryption/decryption behaviour can easily be set by using profiles for optimizing storage efficiency or processing speed (see Section 5.2.4).
- It conducts practical performance comparisons using the currently most popular NoSQL WCSs[6] Apache Cassandra [57] and Apache HBase [17] as underlying platforms (see Chapter 7). This allows making statements in the context of real world technologies in contrast to the unrealistic in-memory benchmarks usually provided in most other research works in this field.
- Apache Cassandra and HBase support is built-in. Other WCS database technologies as well as more PPE schemes can easily be added (see Section 6.4).

1.3. Thesis Impact

During the course of this work, intermediate results have been published in the following peer reviewed conference proceedings:

- **Tim Waage**, Daniel Homann, Lena Wiese. Practical Application of Order-Preserving Encryption in Wide Column Stores (SECRYPT2016), Proceedings of the 13th International Joint Conference on e-Business and Telecommunications, pages 352-359, Scitepress, 2016.
- **Tim Waage**, Lena Wiese. Ordnungserhaltende Verschlüsselung in Cloud-Datenbanken (DACH Security 2016), Proceedings of DACH2016, pages 75-86, syssec, 2016.
- **Tim Waage**. Order Preserving Encryption for Wide Column Stores (SICHERHEIT2016), GI Lecture Notes in Informatics volume 256, pages 209-216, Köllen Druck+Verlag, 2016
- **Tim Waage**. Durchsuchbare Verschlüsselung in NoSQL Datenbanken (INFORMATIK2015), Lecture Notes in Informatics volume 246, pages 1747-1758, Bonner Köllen Verlag, 2015

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

- **Tim Waage**, Ramaninder Singh Jhajj, Lena Wiese. Searchable Encryption in Apache Cassandra. In Foundations and Practice of Security - 8th International Symposium (FPS2015), Lecture Notes in Computer Science volume 9482. Springer, 2015
- Lena Wiese and **Tim Waage**. Benutzerfreundliche Verschlüsselung für Cloud- Datenbanken (DACH Security 2015), IT Security & IT Management, pages 12-23. syssec, 2015.
- **Tim Waage** and Lena Wiese. Benchmarking encrypted data storage in HBase and Cassandra with YCSB. In Foundations and Practice of Security - 7th International Symposium (FPS2014), Lecture Notes in Computer Science volume 8930, pages 311-325. Springer, 2014.

In addition, the following papers have been published in peer reviewed journal articles:

- **Tim Waage**, Lena Wiese. Implementierung von kryptographischen Sicherheitsverfahren für Apache Cassandra und Apache HBase, HMD Praxis in der Wirtschaftsinformatik 53.4, pages 499-513, Springer, 2016
- Lena Wiese, **Tim Waage**. A Fragmentation and Replication Scheme for Flexible Query Answering, The Computer Journal 60.3, pages 308-321, Oxford University Press, 2016.

Moreover, the author contributed to the following related work in the field of searchable encryption:

- Christian Göge, **Tim Waage**, Daniel Homann, Lena Wiese. Improving Fuzzy Searchable Encryption with Direct Bigram Embedding (TRUSTBUS 2017), to appear in the conference proceedings in August 2017.

1.4. Thesis Structure

The thesis is structured as follows. Chapter 1 is this introduction. Chapter 2 starts with the foundations. Cloud computing is introduced in general, but also NoSQL WCSs in particular. This leads to the adversary scenario and PPE as counter measure. Chapter 3 shows how related approaches in this field tried to solve the problem of maintaining privacy and confidentiality in scenarios, where sensitive data is outsourced to third party platforms. It also presents related problem formulations. Chapter 4 presents the theoretical core of the thesis. It explains how PPE can be utilized to store and process encrypted data in remote database systems. In particular it elaborates on the requirements of such encryption schemes in the context of NoSQL WCSs and shows how appropriate approaches have been selected and modified for this thesis. Chapter 5 presents the integration of encryption technologies into WCS databases, while Chapter 6 explains concrete implementation details. Chapter 7 gives an evaluation in terms of performance, functionality and security properties of the approach introduced by this thesis. Finally, Chapter 8 summarizes the thesis, presents a discussion regarding the strengths and weaknesses/limitations and points out directions of future work in this research area.

2 Background

This chapter presents the foundations of the thesis. It provides the required background knowledge on which the following considerations are based.

In order to provide a certain understanding of how NoSQL WCSs work it starts with introducing their technical background, including their working principles and security weaknesses. The following section presents a definition of and further elaboration on cloud computing. In particular, it clarifies the role of NoSQL WCSs in this area. Having settled on the technical details the adversary scenario is introduced followed by an introduction to PPE.

Contents

2.1. NoSQL Wide Columns Store Databases

2.1.1. History

Distributed data storage becomes more and more important due to the increased amount of data being produced every day, by private persons (for example in social media platforms) as well as by business or research. Especially modern web services have a high demand for consistency, availability, partition tolerance, as well as performance and scalability, that are at best difficult and expensive to achieve with traditional relational databases.

However, NoSQL databases, running in distributed cloud environments, were made to meet these requirements. Their development is often driven by so called "Web 2.0" services, e.g. Cassandra for Facebook's inbox search [57]. Strengths of traditional relational databases, like transaction consistency, realtime writing or complex (multi-table) queries on structured data are not that crucial in these environments, where the ability to process large amounts of unstructured data is much more important.

NoSQL is a collective term to describe many database designs, that are different from the traditional relational SQL-based architectures. Thus they come in a variety of working principles. Excellent introductions can be found for example in [23, 46]. The most popular categories (which sometimes do overlap) are key-value-stores (e.g. Redis [22]), document stores (e.g. MongoDB [27], CouchDB [5]) and as foundation for this work: wide column stores.

Since it is in the nature of their purpose NoSQL databases are distributed systems, which makes them often being mentioned in the context of the CAP theorem [21, 20]. Roughly summarized the CAP theorem says that a distributed system can only accomplish two out of the three properties: **c**onsistency (all data replicas have the same state after an update), **a**vailability (all queries to the system are answered within an acceptable time) and **p**artition tolerance (the system keeps working in case of a loss of messages or nodes). Two examples of how different concrete architectural designs fulfill the CAP theorem are given in Section 2.1.2.1 and 2.1.2.2.

2.1.2. Data Model and Operating Principles

WCSs (sometimes also called extensible record stores) are inspired by Google's BigTable architecture [24], but there are also publicly available open source databases, that rely on the same or a very similar data model (for example Hypertable [55], Apache Cassandra [57], Apache HBase [17] and Apache Accumulo [83]). In literature the term "table" is used synonymously to the term "column family". For the rest of this thesis we will prefer "table" as well to not get confused with separate columns.

The operating principles of WCSs can be roughly described as follows. While they use tables, rows and columns like traditional relational (SQL-based) databases, the fundamental difference is that columns are created for each row instead of being predefined by the table structure. Thus, except a row's identifier (see next paragraph), 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 documents. They can consist of an arbitrary number of fields, that are required to have unique names ("column qualifiers") and can be of any type (but some WCSs do not distinguish between data types). They can also be grouped, which results in column families, commonly also simply referred to as tables.

WCSs are distributed systems. They achieve scalability by splitting both rows and columns or in other words: their distribution strategy includes horizontal and vertical partitioning (simultaneously on the same table).

- **Rows (horizontal partitioning).** As mentioned above, every row has an identifier that is unique for the table, commonly referred to as "row key". However in this thesis the term "row identifier" is used to avoid confusion with cryptographic keys.

  Data is maintained in lexicographic order by that row identifier. 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 best case on neighbouring sectors on disk, but at least on the same node of a cluster for the purpose that reads of ranges require communication to a minimum number of machines.

  Because row identifiers are used for coordinating distribution, changing them would result in changing the data's physical position within the database (cluster), which is prohibitively expensive. That is why most WCSs do not even support changing row identifiers at all after a row has been inserted.

- **Columns (vertical partitioning).** Disk access and memory management in WCSs are performed at column family level.

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 key are of course table and column names.

Thus, more formally WCSs can be considered sparse, distributed, multidimensional maps of the form

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

(see [24]).

Even though all WCSs share the general data model described above with only minor individual modifications, they may differ fundamentally in the architectural concepts they are pursuing. Apache Cassandra and Apache HBase provide excellent examples here, as can be seen in the following sections (for a quick comparison of the two, see Table 2.1). Since both of them can be used for FamilyGuard, more detailed introductions follow in Section 2.1.2.1 and 2.1.2.2.

	Cassandra	HBase
file system	local file system	HDFS/local file system
architecture	P2P	master/slave
consistency	weak	strong
optimized for	writes	reads
node coordination	Gossip protocol	Zookeeper
data placement	virtual nodes	HRegionServers
data types	Strings, Integers, etc.	byte arrays only
query language	CQL	native Java API
custom code execution	UDFs via CQL	“Co-Processors”

Table 2.1.: Architectural principles of Apache Cassandra and Apache HBase

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.

2.1.2.1. Apache Cassandra

Apache Cassandra has been a top-level-project of the Apache Foundation since early 2010 and continuously gets updates. It appears in literature also as table store, extensible record store or column family store.

Cassandra follows a strictly symmetric P2P concept using the Gossip protocol for coordination purposes. It runs in a single Java process per node and can make use of the local file system. Nodes back up each other, depending on the chosen replication factor and replication strategy. Data is distributed using consistent hashing in order to perform the data allocation among the cluster. Thus, concerning the CAP Theorem introduced in Section 2.1.1 Cassandra is designed for high availability and partition-tolerance.

The writing process of Cassandra consists of four phases.

1. Firstly, the data operation gets recorded to a log file (“CommitLog”).

2. Then the update is written to a so called “memtable” for temporal caching. Memtables are sorted by row identifiers.

3. When the amount of cached data reaches certain thresholds, the data is written sequentially (“flushed out”) to a so called “SSTable” on disk for persistent storage. This process can also be triggered manually and can be done very quickly, which makes writes very fast. SSTables are final, once they are written, they can not be changed in the future.

4. To reduce the number of SSTables that have to be involved in read operations multiple SSTables are merged into single new ones, using a procedure called “compaction” in order to get rid of outdated records. There are a couple of different strategies for when and how to do that. They should be chosen carefully depending on the kind of workload. When the compaction process has finished, the old SSTables are deleted.

Apache Cassandra is the backbone of many popular services, e.g. Spotify, Twitter, Netflix and more[1].

2.1.2.2. Apache HBase

The process of writing data in HBase is similar to that of Cassandra. Only the terminology is slightly different, having an “HLog” instead of an CommitLog, a “MemStore” instead of a memtable and a “StoreFile” instead of an SSTable. However, unlike Cassandra HBase is not designed to provide the concept of data types like numbers (Integers) or text (Strings). Everything is treated as byte array instead.

Another fundamental difference to Cassandra is the master-slave concept. A master node (“HMatser”) monitors multiple slave instances (“HRegionServers”). A Zookeeper system is necessary for managing different HBase processes. While the system is robust against failures of region server nodes, the failure of a master can have a severe impact on the cluster. However, having a master also allows strictly consistent reads and writes. Thus, concerning the CAP theorem HBase covers consistency and partition tolerance. Master and region server nodes serve a variety of purposes and run multiple processes in the background:

- The *HMaster* is responsible for assigning regions (tables or ranges of rows) to HRegionServers. It also monitors the health of each HRegionServer and manages metadata changes as well as table related tasks (e.g. enabling/disabling of tables, table schema changes like adding, modifying or deleting column families). Furthermore it runs a load balancer in order to move regions within the cluster to balance its load.

[1]Planet Cassandra: Apache Cassandra Use Cases - `http://www.planetcassandra.org/apache-cassandra-use-cases/`

- The *HRegionServer* handles read and write requests (get, put, scan, delete). Furthermore it is responsible for flushing Memstores to StoreFiles and managing compactions.

HBase can use the Hadoop Distributed File System (HDFS) out of the box and supports parallelized data processing via MapReduce. Hadoop is a Java framework for large scale data processing in distributed environments. Amongst others[2] the most popular use case for HBase surely is Facebook's inbox search [17].

2.2. "Cloud" Computing

While the "Cloud" often is referred to as just "other peoples computers", there are indeed comprehensive definitions of what cloud computing is. They can be found for example in [65]. Roughly summarized cloud computing has the following characteristics. It is an *on-demand service*, made *available over a broad network access*. The cloud computing provider makes use of *resource pooling* in the sense that multiple customers share processing capabilities, memory and network bandwidth, all of which can scale rapidly to provide *elasticity*. These resources come as *measured services* and can be monitored, controlled, and thus be invoiced in certain intervals.

There are mainly two ways of categorizing cloud services, either by their form of deployment (private, community, public, hybrid, etc.) or as in particular relevant for this thesis: by their form of service or business model.

2.2.1. Technical Cloud Stack

The technical cloud stack consists of three layers, as shown in Figure 2.1.

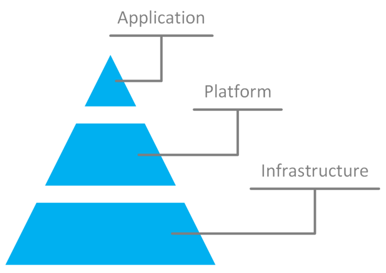


Figure 2.1.: Technical stack of cloud computing

[2]The Apache Software Foundation: Powered By Apache HBase - `https://hbase.apache.org/ poweredbyhbase.html`

2.2.1.1. Software as a Service

Software as a Service (SaaS) is an alternative to the traditional software license model. Its primary goal is to avoid a complex, expensive and risky development of IT solutions. Instead the customer rents all necessary components (applications running on a cloud infrastructure) on demand. The SaaS provider takes care of all maintenance tasks, which reliefs the customer from additional hardware costs, initial setup costs, taking care of updates, etc. However, the customer pays for certain intervals of usage instead of for separate software licenses.

2.2.1.2. Platform as a Service

Platform as a Service (PaaS) means the capability of the customer to deploy applications onto the cloud infrastructure using services of the provider. However the customer has in particular no control over operating systems, storage or network and servers. They only manage the applications and their configuration (usually via a web interface). Sometimes it is difficult to seperate PaaS from IaaS (see next section), since many providers bundle PaaS with the underlying IaaS. For instance AWS can also be considered a PaaS provider. Other popular examples are Microsoft Azure [99] or Google's App Engine[3].

2.2.1.3. Infrastructure as a Service

Infrastructure as a Service (IaaS) means rather than buying computing infrastructure the user rents it flexibly on demand. This results in a couple of advantages: applications can become affordable, workload peaks can be compensated and the system scales easily. A popular IaaS provider is Amazon Web Services (AWS)[4], offering products for computing ("EC2") and storing ("S3").

2.2.2. Cassandra and HBase in the Cloud

Cloud computing and storage providers offer infrastructures and platforms for a variety of services, starting from website hosting over large scale data processing to setting up virtual machines for countless applications. Widely used cloud service providers are the Google Cloud Platform, Microsoft Azure, Amazon Web Services and Rackspace. They differ primarily in price and service. Instead of a "manual" deployment of Cassandra and HBase clusters, they offer mechanisms to simplify the deployment process. This chapter gives a brief introduction on how that works.

[3]Zahariev, Alexander: Google App Engine - `http://www.cse.tkk.fk/en/publications/B/5/papers/1Zahariev_final.pdf`

[4]Amazon Web Services - `https://aws.amazon.com/`

- The Google Cloud Platform is based on the same infrastructure that Google uses for its end-user products like its web search or YouTube. It has a very good native support for Cassandra. A web interface ("cloud launcher") can be used to configure and run a Cassandra cluster on hardware of different performance levels. Googles cloud SDK provides tools for interacting with it mainly via SSH. A Hadoop cluster as required by HBase for running in distributed environments, can be created in the same way.

- Microsoft Azure is Microsoft's cloud computing platform, publicly available since 2010. It does not come with native support for Cassandra, which has opened the market for third party entities, e.g. "Instaclustr", which provide all the necessary tools to make running Cassandra on Azure as simple as on the Google Cloud Platform. In contrast, HBase is natively supported via "HDInsight", a service provided by Microsoft for managing hadoop-based projects, that comes with an appropriate web interface.

- Amazon Web Services is a set of online services that was started in 2006 and meanwhile became the foundation of many other popular cloud services, e.g. Dropbox or Foursquare. AWS (in particular Amazon EC2) does not support Cassandra in a native way. Instead third party developers provide so called Amazon Machine Images (AMIs) in order to simplify the management of Cassandra clusters. Using a web interface ("AWS Console") they can be installed on the desired hardware and provide another web interface ("OpsCenter") that is used to create and monitor Cassandra clusters. Managing HBase environments also requires some work in advance. AWS provides a service called "Amazon Elastic MapReduce" (Amazon EMR) that comes with a web interface for creating Hadoop clusters as they are required by HBase. It can be installed using AMIs similarly as required by the pre-installation process of Cassandra. HBase is part of these AMIs.

- Rackspace is primarily a web hoster that extended its business to cloud computing in 2010 with the OpenStack project, an open source platform for cloud computing. Rackspace provides hosting of Cassandra Clusters as managed service. Internally they use Ansible, which is an Open Source Platform for configuring and maintaining computers in general. HBase is supported with a set of tools that Rackspace has designed in cooperation with third party developers. However, both databases can also be deployed on Rackspace using Apache Whirr, a set of libraries provided by the Apache Foundation for managing cloud services. Unfortunately that has not been further maintained since early 2015.

2.3. The Adversary Scenario

FamilyGuard 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" [39]. Note that an active manipulation of the encrypted data without knowing the appropriate keys would render it useless anyway, but still: no information would leak.

The "honest-but-curious" threat model has applications in various practical scenarios. Examples for sensitive data range from electronic health records [94] to classified business data. As a matter of fact more and more individuals or companies outsource such data to the cloud in order to get rid of the burden of buying and maintaining additional infrastructure[5]. As mentioned in Section 1.1 use cases are for instance backups or the reduction of operational coasts. Unfortunately, in general it is not guaranteed that data confidentiality will be preserved. Problems can arise as well, if once trustworthy storage providers sell their business to untrusted companies.

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. This leads to the problem of data integrity, which is discussed further in Section 3.3.

2.4. Property-preserving Encryption

There are many cryptographic algorithms for providing data confidentiality, but besides the protection of data there is also the requirement for the ability to process it efficiently. Thus, in general there is a tension between these two needs. Ideally direct processing of encrypted data should be possible in order to avoid decrypting, processing and then re-encrypting again. 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 (Section 4.1 explains in detail, how this intentional leakage is utilized in the context of databases). This concept is commonly referred to as property-preserving encryption (PPE). The following sections provide a basic understanding of how that works.

[5]KPMG International: Cloud Survey Report 2014 - `http://www.kpmg.com/US/en/about/alliances/ Documents/2014-kpmg-cloud-survey-report.pdf`

2.4.1. Cryptography Basics

Before getting to the principles of PPE we introduce the fundamental cryptographic concepts and primitives, that are the foundation for the concrete schemes, that are used later on (see Section 4.2, 4.3 and 4.4).

2.4.1.1. Pseudo Random Generators

Pseudo random generators (PRG) are deterministic (hence "pseudo"-random) algorithms that generate long bit sequences from shorter randomly selected seeds. These sequences should be indistinguishable from truly random sequences. There are several definitions for what it really means to have such a random appearance, e.g. statistical measures or the Kolmogorov complexity [60]. This work however relies on the definition of computational indistinguishability.

Definition 2.1 (Computational Indistinguishability)**.** *Random variables X and Y having the form of $0, 1^m$ are (t, ϵ) indistinguishable, if for every non-uniform algorithm A running at most in time t, we have $|Pr[A(X) = 1] - Pr[A(Y) = 1]| \leq \epsilon$.*

The left-hand side of the above equation is also called the advantage of A, where as "Pr[S]" is short for the probability of a statement S to be true. Non-uniform means that A is allowed to work in different unrelated ways, depending on the input length m. That serves as foundation for defining the PRG formally, as it was already informally introduced at the beginning of this section:

Definition 2.2 (Pseudo Random Generator)**.** *A deterministic function $G : 0, 1^m \rightarrow 0, 1^n$ is a (t, ϵ) pseudo random generator, satisfying the following two conditions:*

1. *expansion: $m < n$*
2. *pseudorandomness: $G(X_m)$ and X_n are (t, ϵ) indistinguishable.*

PRGs are widely used in cryptography. Since perfectly secure encryption is expensive in terms of key lengths, a main application for them is the reduction of these key lengths. Another common use case is their usage in stream ciphers. This thesis utilizes them in both ways (see Section 2.4.1.3 and 4.4). Further descriptions and analysis of PRGs can be found in [38].

2.4.1.2. Pseudo Random Functions

Outputs of pseudo random functions (PRFs) cannot efficiently be distinguished from truly random outputs (see Definition 2.1) by an adversary A. This even holds, if A is allowed to chose own inputs depending on previously obtained outputs [38] (commonly referred to as "adaptive querying"). Having a key k, a

PRF F with domain D and range R, all of the form $\{0,1\}^*$, F can be defined as follows.

Definition 2.3 (Pseudo Random Function)**.** *A function $F : k \times D \rightarrow R$ is a (t, q, ϵ)-secure pseudo random function, if every adversary A executing at most q oracle queries with runtime at most t has an advantage $|Pr[A^{F_k} = 1] - Pr[A^X = 1]| \leq \epsilon$, where X represents a random function selected uniformly from all maps from D to R, and where the probabilities are taken over the choice of k and X.*

Popular applications for PRFs are key derivations and dynamic hashing [40].

2.4.1.3. Symmetric Cryptography

Using the same secret key k for encryption and decryption is the main characteristic of symmetric cryptography [39] in contrast to public key cryptography. That requires the participating entities to agree on k first, which can be done having one party generate the key and then send it to the other one over a supposedly secure channel. In the database scenario of this thesis both parties can be represented by the same entity (the data owner) or by different entities (data owner and database users). Cryptographic primitives of symmetric cryptography are usually either stream ciphers or block ciphers.

2.4.1.3.1. Stream Ciphers Stream ciphers [75] operate on single digits (bits) in the way that every digit of the plaintext is combined with the corresponding digit of the keystream in order to get a digit for the ciphertext stream. In practice the keystream often is created using pseusorandom generators (see Section 2.4.1.1) and the combination is done using an exclusiv-or (XOR) operation. Popular stream ciphers are RC4 [79] (used e.g. in BitTorrent and in a modified version in Skype, even though it is more and more considered insecure) and A5 [11] (used e.g. for voice encryption in GSM).

2.4.1.3.2. Block Ciphers Block ciphers [56] encrypt units of bits at once. Thus, fixed size blocks of the plaintext are encrypted to equally sized blocks of ciphertext (which means plaintexts have to be split and/or padded eventually). Block ciphers can formally be considered PRFs (see Definition 2.3), parameterized by a key with a length matching the blocksize. Popular examples are the Data Encryption Standard [77] (DES, widely utilized during the 1970s until the 1990s, but by now considered insecure), the Advanced Encryption Standard [76] (AES), and Blowfish [84] (see also Section 4.2).

The traditional interest of symmetric cryptography schemes is based on efficiency considerations. Due to the fact, that they are mainly based on simple binary operations, they can be implemented in hardware very easily and thus perform well in practice with low latency. However, this also comes with the disadvantage that all encrypted messages can of course be decrypted once k gets stolen.

2.4.2. Searchable Encryption

The purpose of searchable encryption (SE) is enabling a server to search over encrypted data without gaining any information about the plaintext data.

2.4.2.1. How it works

Schemes for SE usually use search indexes for sets of documents. Depending on the schema design these indexes are forward indexes or inverted indexes (see below), containing predefined or extracted keywords. Indexes are encrypted in a way, that only a trapdoor allows for comparing the searchword with the predefined keywords in the index. Index-based schemes for SE can be divided in three subcategories based on their index type: index per word, index per document or tree-based index. However, there are also schemes, that do not require an index at all.

2.4.2.1.1. Index-based

Index per Word This index structure contains one entry for every keyword, accompanied by a list of identifiers of all the documents in the dataset containing this keyword. Thus it is called an "inverted index". Searches are just a lookup in the index and hence highly efficient. The downside however are increased computational costs when adding data, since all the index entries containing the corresponding keywords have to be updated as well.

Index per Document This index structure is also known as "forward index" and based on building an index for each document (or one entry per document respectively) in the dataset, containing its keywords. This makes updating less expensive than working with inverted indexes.

Tree-based index SE can also be designed to work with trees, e.g. (multi-dimensional) B-trees. However, this thesis does not consider these schemes, because the data model of WCS databases is not well suited for maintaining tree structures. Thus working with them would be computationally expensive in practice.

2.4.2.1.2. Not Index-based

Schemes without index structures have to embed the trapdoor in a special format in the ciphertext itself. During search that trapdoor gets extracted again and can then be checked for that very format in order to determine whether there is a match or not. Because of not having an index SE schemes of this kind have to iterate by the nature of their design at least once over the complete plaintext for encryption and at least once over the complete ciphertext for decryption. On the other hand no resources are necessary for building, maintaining and storing indexes at all.

2.4.2.1.3. Related Problems

This thesis focuses on exact keyword search schemes for one searchword per query. However there are schemes designed to deliver results for multiple searchwords or even when the user entered a searchword with typos. They are commonly referred to as "multiple keyword search" and "fuzzy search" schemes, needing much larger data structures and more computational time than exact keyword search schemes, which makes them hardly feasible in practical database scenarios (for details, see Section 4.4.1).

2.4.2.2. Security

Mainly three kinds of information can leak unintentionally when using SE schemes:

- index information (e.g. number of words per document, number of documents, lengths of documents, document-IDs)
- search pattern information (what was searched for?)
- access pattern information (how many answers do I get from executing a certain query compared to executing another one?)

There are several security classifications in order to categorize this information leakage. Hence in general it makes no sense to use the word "secure" for a SE algorithm without defining what security really means, it has to be clarified what sort of security can be expected against what sort of attacks. Thus the schemes for searchable encryption that are used in this work, can be categorized in the form "$X - Y$", where X represents the security level that can expected to hold against an attacker model Y. Although there are lots of these classifications the following sections focus only on the ones that are crucial for this thesis.

2.4.2.2.1. Levels of Security

SEM - Semantic Security The concept of SEM was first introduced by [41] for public key cryptosystems, even though they started to use the actual term "semantic security" two years later [42]. According to their definition a system is semantically secure if "whatever an eavesdropper can compute about the cleartext given the ciphertext, he can also compute without the ciphertext". Concerning a symmetric-key cryptosystem, an adversary must not be able to compute any information about a plaintext from its ciphertext.

Definition 2.4 (Semantic security in symmetric-key cryptosystems)**.** *Given two plaintexts of equal length and their two respective ciphertexts, an adversary cannot determine which ciphertext belongs to which plaintext.*

Literature is in disagreement, if the definition of SEM means that no information about the plaintext can be revealed at all or just cannot be feasibly extracted [39]. However SEM is usually considered being equivalent to IND-CPA security [42] (see Definition 2.5 and 2.6) .

IND - Indistinguishability Ciphertext indistinguishability can not be defined clearly without making assumptions about the capabilities of a particular adversary. Common definitions present it as a game, considering a cryptographic system to be secure, if an adversary can not win with significantly greater probability than an adversary who guesses randomly. Thus, the following minimum requirement holds for every attacker model.

Definition 2.5 (Ciphertext Indistinguishability)**.** *Given an encryption of a plaintext randomly chosen from two plaintexts, any adversary is not able to identify the message that has been encrypted with probability better than that of random guessing (0.5).*

Definition 2.6 clarifies, what "better" means in that context.

2.4.2.2.2. Attacker Models

CPA - Chosen Plaintext Attacks As the the name of the concept already suggests, the CPA attacker model gives the adversary as described in Section 2.3 the ability to chose the plaintext space. This leads to the following definition for IND-CPA security.

Definition 2.6 (IND-CPA: Indistinguishability under Chosen Plaintext Attacks)**.** *IND-CPA security for symmetric crypto systems can be defined using a game between a (probabilistic polynomial time) adversary A, a challenger C and an encryption oracle O, consisting of the the following steps:*

- *The challenger C allows the adversary A to perform a polynomially bounded number of encryptions or other operations using the encryption oracle O.*
- *A submits two distinct chosen plaintexts P_0 and P_1 to C.*
- *C selects $x \in 0, 1$ uniformly at random and returns the challenge ciphertext $Enc_O(P_x)$ to A.*
- *A is allowed to perform any number of computations and outputs a guess x' for the value of x.*
- *A wins, if $x' = x$.*

A cryptographic system is IND-CPA secure, if A has only a negligible advantage over random guessing to win the game. A negligible advantage is given, if A wins the game with probability $0.5 + \epsilon(k)$, where ϵ is a negligible function and k is a security parameter in the way that for every polynomial function p, there exists a k_0 such that $|\epsilon(k)| < |\frac{1}{p(k)}|$ for all $k > k_0$.

The concept of IND-CPA security was first introduced by [42], calling it "polynomial security". Note, that IND-CPA completely ignores whether a scheme for searchable encryption has an index or not. Thus, it is by definition not applicable for index-based SE schemes.

CKA - Chosen Keyword Attacks The main cause for information leakage in index-based schemes for SE are are the indexes themselves. Thus, IND-CPA can not be considered appropriate for such schemes. The first to realize that were [36], proposing the notion of IND1-CKA (indistinguishability against adaptive chosen keyword attacks), which guarantees a scheme for index-based searchable encryption produces indexes of equal size for documents of equal size. In other words: given an index I and two documents D_1 and D_2 of equal size, an adversary A is unable to decide which of the documents D_1 or D_2 is encoded in I. Later [36] introduced the stronger IND2-CKA notion, based on [25], in which A cannot even distinguish indexes from two documents of unequal size. In other words: all indexes have to look like they were containing the same amount of words. Still, those notions are not appropriate for SE, because they do not consider trapdoors (and have later been proven to be incorrect, see Section 4.4.3). [31] finally shows that the security of indexes is strongly connected to the security of trapdoors and introduced two adversarial models, IND-CKA1 and IND-CKA2, that can be considered the current standard security definitions for SE. A compressed informal version of those notion definitions can be given as follows.

Definition 2.7 (IND-CKA1: Non-Adaptive Indistinguishability Security). *Let "history" be the interaction between client and server, in particular containing a document collection and a set of searchwords. Let the history's "view" be the* *corresponding index, trapdoors, the number of documents in the collection as well as their ciphertexts. Note, that the view should not leak any information about the history, that is not supposed to leak, in particular the search results. That information is denoted as "trace". A trace includes the identifiers of the documents containing searchwords as well as the corresponding trapdoors.*

A cryptosystem is IND-CKA1 secure, if "for any two adversarially constructed histories with equal length and trace, no (probabilistic polynomial-time) adversary can distinguish the view of one from the view of the other with probability non-negligibly better than 0.5." [31].

Non-negligibility is defined as in Definition 2.6. Thus IND-CKA1 obviously includes security for trapdoors. Furthermore it guarantees that trapdoors do not reveal any information about the corresponding keywords, except for access and search patterns. "Non-adaptive" means the assumption that the clients executes all searches at once. Knowing that this is not realistic in practice, there is also an adaptive version of IND-CKA1:

Definition 2.8 (IND-CKA2: Adaptive Indistinguishability Security)**.** *Let "history", a history's "view" and "trace" be defined as in Definition 2.7.*

A cryptosystem is IND-CKA2 secure, if "for any two adaptively-constructed histories with equal length and trace, no (probabilistic polynomial-time) adversary can distinguish the view of one history from the view of the other with probability non-negligibly better than 0.5." [31]

Again, non-negligibility is defined as in Definition 2.6. Thus, IND-CKA2 allows the adversary to adapt its queries to previously obtained trapdoors and outcomes, which can lead to more sophisticated attacks. Hence, IND-CKA2 is a strong security notion for SE.

2.4.3. Order-Preserving Encryption

The purpose of order-prerserving encryption (OPE) is enabling a server to learn about the relative order of data elements without gaining any information about their exact values. Thus, its main use cases are sorting and range queries over encrypted data.

2.4.3.1. How it works

[2] were the first to introduce the problem of OPE and proposed a theoretical scheme to address it. Formal definitions followed mainly by [13]. In summary, OPE schemes have to satisfy the conditions as given in Definition 2.9.

Definition 2.9 (Order-Preserving Encryption)**.** *An order-preserving (symmetric) encryption scheme with plaintext space D (domain) and ciphertext space R (range) is a tuple of algorithms $(KGen, Enc, Dec)$ satisfying the following conditions:*

- *The key-generation algorithm $KGen$ outputs a random key k.*
- *The encryption algorithm Enc uses k and a plaintext p to output the ciphertext $c = Enc_k(p)$.*
- *The decryption algorithm Dec uses k and a ciphertext c to output the plaintext p. Thus it holds: $Dec_k(Enc_k(p)) = p$.*
- *The order relation of plaintexts is preserved, i.e. $p_1 \leq p_2 \Rightarrow Enc_k(p_1) \leq Enc_k(p_2)$ for all $p_1, p_2 \in D$.*

Approaches to OPE can be divided in two groups, index-based and not index-based schemes.

2.4.3.1.1. Index-based The index of an OPE scheme is its state, containing plaintext-ciphertext-mappings. That state can also be considered the scheme's key k, playing the role of the secret information. $KGen$ initializes k. Performing the encryption algorithm Enc with not yet encrypted plaintexts as input updates k. The decryption algorithm uses k to perform lookups.

2.4.3.1.2. Not Index-based Not index-based OPE schemes generate plaintext-cipher-text-mappings independent from any internal state. They use the key k for their underlying cryptographic primitives.

2.4.3.2. Security

Compared to SE (see Section 2.4.2), specifying the information leakage of OPE is an open problem. An indistinguishability notion like IND-CPA (see Definition 2.6) can not be achieved by design, because OPE schemes reveal more about the plaintexts than their order, for example information about the relative distance between plaintexts. Since the encryption algorithm Enc has to use a monotone increasing function the distance between two ciphertexts $Enc_k(p_1)$ and $Enc_k(p_2)$ is always related to the distance between the corresponding plaintexts p_1 and p_2.

Unfortunately the number of ways to define OPE security in literature is almost equal to the number of approaches to OPE itself. However, there are a few frequently appearing security notions, mainly defined by [13, 14].

IND-OCPA IND-OCPA (*indistinguishability under ordered chosen-plaintext attack*) was introduced by [13]. It is a weakened form of IND-CPA defined as follows: the adversary is allowed to chose plaintexts adaptively and should not be able to find out about a secret bit of corresponding ciphertexts better than with random guessing (similar to Definition 2.6). [13] showed further, that IND-OCPA can not be achieved, unless the size of its ciphertext-space is exponential in the size of its plaintext-space. Thus, even though the authors describe this definition as useless for most practical cases, the IND-OCPA notion became quite popular in the field of OPE to date.

POPF-CCA In order to avoid further restricting the IND-OCPA definition [13] also introduced another approach, called POPF-CCA (*pseudo random order-preserving function against chosen-ciphertext attack*). For an OPE scheme to fulfill this notion an adversary must not be able to distinguish between the outputs of the encryption/decryption oracle and a *truly* random order-preserving oracle and its inverse. Unfortunately no statements are made concerning what actually leaks or is hidden.

(r,z)-WOW r,z-Window One-Wayness (sometimes also referred to as (r,q+1)-WOW) is defined as follows: the adversary A gets a set of z ciphertexts of randomly chosen plaintexts. Then A has to come up with an Interval I of maximum length r. A wins, if at least one of the underlying plaintext lies within I. Note that this notion captures a very practical aspect of database security: consider z data items are stored in a database using OPE encryption and an adversary who wants to know one of them breaks into the system and steals all ciphertexts.

Compared to the security definitions for SE all of these security notions are not really helpful in stating what information about the plaintexts can be semantically hidden. However there are approaches (e.g. [63]) trying to answer this question, but none of them can be considered a standard OPE security notion so far.

2.4.4. Homomorphic Encryption

Homomorphic encryption allows performing computations using ciphertext data as input with the output being in encrypted form as well. When decrypted, it matches the result as if it was carried out on plaintext. Since NoSQL wide column stores and their query methods barely feature server-side computations (rare exceptions are for instance the `SUM` and `AVG` operators in Cassandra's query language), homomorphic encryption would only be of little use for this thesis. However for the sake of completeness and to understand how other architectures for encrypted databases use it (see Section 3.1), it is discussed here briefly in that regard. Thus, the following elaboration on it is limited to its important definitions and usage for encrypting databases.

2.4.4.1. Homomorphic Cryptosystems

Homomorphic encryption schemes can be divided into two sub-categories: partially and fully homomorphic encryption schemes. However, a small number of approaches does not really fit in either of these categories, for instance [15], allowing multiple additions, but only one multiplication.

2.4.4.1.1. Partially Homomorphic Cryptosystems

Partially homomorphic encryption schemes allow only certain computations, while others are not supported. Thus, these schemes can be divided in further sub-categories, the most important of which are additive and multiplicative homomorphic encryption.

Additive Homomorphic Encryption Schemes of this category allow carrying out additions on ciphertexts. Thus, their homomorphic property is:

$$Enc(x_1) \cdot Enc(x_2) = Enc(x_1 + x_2)$$

with x_1 and x_2 being the plaintexts. That means a multiplication of the ciphertexts is equivalent to the encrypted sum of the plaintexts. Examples for those schemes are Goldwasser-Micali [41], Benaloh [10] and Pallier [68], with Paillier being the most practically relevant scheme (publicly available implemented[6] and also used in practical approaches for encrypted databases, see Section 3.1.1).

Multiplicative Homomorphic Encryption Schemes of this category allow multiplications on ciphertext, which means their homomorphic property is

$$Enc(x_1) \cdot Enc(x_2) = Enc(x_1 x_2).$$

Well known examples are RSA [80] and ElGamal [32].

2.4.4.1.2. Fully Homomorphic Cryptosystems

Fully homomorphic cryptosystems allow arbitrary computations on ciphertexts. They are additive and multiplicative at the same time. The first scheme for fully homomorphic encryption [34] was proposed by Craig Gentry in 2009.

2.4.4.2. Encrypting Databases with Homomorphic Encryption

Since the input and output of computations using homomorphic encryption are encrypted, different steps in processing database queries can be chained without leaking information. This is very appealing, especially in the SQL world, where it is possible to have calculations in the query, like for instance `SELECT ... FROM ... WHERE x + 10 < 50`).

Unfortunately homomorphic encryption is computationally expensive. In 2009 Craig Gentry estimated, that processing a Google search request based

[6]Google Code Project: “The Homomorphic Encryption Project - Computation on Encrypted Data for the Masses” `https://code.google.com/archive/p/thep/`

on his approach would multiply the necessary time by 1 trillion[7]. Partially homomorphic cryptosystems usually offer better performance. They are still slow, but efficient enough for specific operations to a certain degree. For instance the Paillier scheme can be used to sum encrypted data on the server side, but the cost of decrypting its ciphertexts at the client can be prohibitively high when computing the sum of a couple of values. That means in certain situations, it can be more efficient to decrypt and sum individual data items at the client rather than run aggregates on the server [95]. Thus, improving Pailliers performance is subject to ongoing research [48] (see also Section 3.1.3).

Besides the performance issues, homomorphic encryption comes with another fundamental problem. Due to randomization steps in the algorithms the results start to get prohibitively imprecise after a certain number of operations. This phenomenon is referred to as "ciphertext noise" or "ciphertext dirt" [88]. Making them "cleaner" (meaning more precise again) is possible, but is again computationally expensive.

[7]Michael Cooney, Network World (25/06/2009). "IBM touts encryption innovation" `http://www.networkworld.com/article/2259168/data-center/ibm-touts-encryption-innovation.html`

3 Related Work

The previous chapters introduced the background of NoSQL WCSs and PPE. After these theoretical concepts were described in detail, the following chapter provides a survey on related approaches for building encrypted databases. The review is started with an overview about how this can be achieved in software for relational databases, for non-relational databases and platform independent using fully homomorphic encryption. Then the focus is switched to architectures involving specialized trusted hardware. Finally, the chapter is concluded discussing a problem that is closely related to the problem of providing data confidentiality: providing data integrity.

Contents

The research interest in processing queries over encrypted data continuously grows with the increasing number of offers by cloud service providers. Since the area covering related topics is very wide, comparability to this thesis can be lost easily. Hence, in order not to lose the scope this chapter is limited to describing approaches, that

- are designed for the honest-but-curious adversary, as discussed in Section 2.3. As this scenario reflects the typical cloud database provider setting, it is of high practical relevance.
- actually compute on encrypted data, in contrast to just storing it, like for example in [37, 59].
- rely on encryption only to provide data confidentiality. Apart from that there are approaches also using data fragmentation as an additional mandatory mechanism to achieve their privacy goals (while that is only an optional feature of the architecture described in this thesis). Examples can be found in [30, 1, 29].

3.1. Software Architectures

3.1.1. Approaches for Relational Databases

A first algebraic approach by Hacigumus et. al [43]

One of the first approaches to processing queries over encrypted data in the context of outsourced databses to untrustworthy providers comes from Hacigumus et. al [43], published in 2002. Assuming the database server only stores data in encrypted form, it introduces an algebraic framework in order to split SQL queries in a way that only a minimum of computation has to be done on client side.

Unfortunately PPE was not very sophisticated back in the year of this work's publication, which is why the key idea here is to transform SQL queries into query trees and then pull selections and projections as high as possible "above" all other relational operations. This separates the tasks doable on server side from the tasks doable on client side. While the database performs relational operations on encrypted versions of the data, the final selection and projection operations along with decryption steps take place on the trusted client machine. This results in hardware requirements on client side that are similar to the ones on the server side, which makes this approach unfeasible in today's cloud computing scenarios.

CryptDB by Popa et. al [74]

The most popular approach to using databases with encrypted content is CryptDB [74] for mySQL and PostgreSQL. It was the first system that could be considered practical, introducing a variety of innovative features, most important: the onion layer model, which this thesis uses as well in an improved adaptation (see Section 5.2.1). Data items are encrypted multiple times using PPE, starting with the weakest scheme, that leaks enough information to process the desired query, up to a very strong scheme, that leaks no information (usually random encryption). If an SQL query has to be processed, only the outer encryption layers are removed, until the query can be processed with the strongest possible scheme in order to leak as little information as possible. The authors call that "adjustable query-based encryption". A proxy client acts as middleware between the client application and the unmodified database server that takes care of adjusting the onion layers as well as the key management.

A major drawback of CryptDB is the fact, that it only uses quite slow PPE schemes and does not provide any alternatives. It seems the authors wanted to avoid (client side or server side) indexes at all costs.Thus, CryptDB does not scale well when datasets reach a certain size. In contrast, this thesis also utilizes faster index-based schemes (see Section 4.3.2.1, 4.3.2.2 and 4.4.2.2).

However, it still receives a lot of scientific attention, in favorable as well as critic ways. A few examples of projects involving it directly are given in this paragraph. Focusing on performance rather than security Shahzad et. al come to the conclusion that CryptDB is well suited for the usage as backbone of health record management systems like OpenEMR[1] [87]. At least when deployed in a cloud environment with reasonable hardware underneath (for instance Amazon EC2 m3.medium machines[2]) the average response time in the tested workloads never exceeded more than two seconds. Tetali et. al use CryptDBs implementations of various PPE schemes for "MrCrypt" [91], a a system that provides data confidentiality in Java programs. It works by statically analyzing a concrete program in order to identify the operations performed on each input data which is a column in a confidential database. Then, it selects an appropriate encryption scheme from CryptDBs stack for that column and transforms the program correspondingly. Akin et. al showed in the context of using CryptDB for web applications in multi-user settings that adversaries or malicious database administrators can easily steal information and even become the administrator of the application [3]. Strikingly they achieve that without targeting the proxy or the web application server itself. Their attack is based on the fact, that CryptDB does not change the order of rows and columns in a table during creation. Both get scrambled in the approach of this thesis, which renders the attack useless when attempted. Another extension of CryptDB is Cipherbase [7] (see Section 3.2).

[1]`http://www.open-emr.org/`
[2]2 CPUs, 4GB RAM, 8GB SSD

Monomi by Tu et. al [95]

Monomi can be considered being an extension of CryptDB as well, trying to support even arbitrary SQL queries. It introduces a so called “designer” for choosing an efficient physical design at the server for a given workload, and a “planner” for selecting efficient execution plans for a given queries at runtime.

The key idea is to split every query into parts for the untrusted server and the trusted client machine. Unlike using transformations of query trees as done in [43] the server utilizes the architecture of CryptDB with the same PPE schemes. Hence it comes with its benefits (more computation possible on server side), but also with the same limitations (slow encryption schemes). Moreover there are still (parts of) queries that are unprocessable for the server and thus have to be done by the client.

While this indeed allows for a lot more queries to be executed at all (for example 19 of 22 TPC-H[3] queries compared to 4 using CryptDB), it also leads to higher requirements for the client machine in order to still allow reasonable computation times. Furthermore, depending on the concrete dataset and queries, a large amount of the data may have to be stored (at least temporarily) on client side, which also causes more network overhead. In some cases this problem can be avoided by pre-computing values, that only depend on other columns in the same row, which results in processing overhead and the need of additional storage capacity for columns that store intermediate results.

BlindSeer by [69]

BlindSeer particularly addresses sub-linear searches for arbitrary boolean SQL-queries. It is based on using two additional entities called index server (that receives the client’s queries) and query checker (that privately enforces policies over queries). The key idea is to identify small privacy relevant sub-problems, solve them securely and use their outputs for completing the overall task, which is search over a large encrypted database. Therefore the authors propose a data structure called bloom filter search tree, that stores collections of keywords in bloom filters in its nodes. Then the query processing works roughly as follows:

Before being able to perform queries a pre-processing step is required. Therefore the server permutes the contents of the database, so that no information about their order can be inferred later on. Then it creates the bloom filter search tree based on that. The permuted database as well as the search tree are sent in encrypted form to the index server. A search gets done by traversing the tree (that is global for the entire database) starting from the root. The client transforms the query in a boolean circuit, where variables correspond to keyword matches, ranges or negotiations. If the circuit outputs true, all children in the tree are evaluated recursively until the leaf nodes are reached. The client receives tokens for the database records associated to these leafs from the index

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

server and can request the final encrypted output from the server for decryption. The speed of this approach depends heavily on the size of the result set of the query.

FamilyGuard avoids the expensive process of building a bloom filter search tree as well as the two additional architectural components (index server and query checker).

L-EncDB by [58]

The authors of [58] use the general architecture of CryptDB, but they replace a few parts with new ideas. They also use a trusted instance between the application and database for tasks like query rewriting and PPE encryption. But instead of using SE in connection with server side UDFs, they use so called format-preserving encryption[4] (FPE) to realize fuzzy searches. They do not adapt the concept of layered encryption.

Even though the authors claim their work to be a solution for cloud computing, their framework relies completely on SQL only. They mention the option of extending their work to the syntax of NoSQL databases only very briefly and not very specifically, leaving out the fact, that most NoSQL databases do not possess query mechanisms that can be mapped to L-EncDB's features like fuzzy search (see Section 4.4.1). Furthermore OPE and FPE ciphertexts are stored without an additional layer of a more secure encryption scheme, leaking order and format information even if that is never required by a query. FamilyGuard provides better security guarantees from the start due to its RND layer (see Section 5.2.1) and more flexibility regarding the underlying data structures due to its database independent implementation (see Section 6.3).

Summary

As could be seen approaches for encrypted relational (thus, SQL based) database systems have two main problems: scalability and/or limited query expressiveness. The most popular and practically feasible designs are extensions of the idea of CryptDB. All approaches require some kind of query pre-processing.

3.1.2. Approaches for Non-relational Databases

Search on Encrypted Graph Data by [52]

An approach aiming for executing queries over encrypted triple patterns using SPARQL is presented by [52]. Here the data owner chooses eight different basic keys for every plaintext document corresponding to the eight binding possibilities of a triple pattern. Additionally, so called restriction patterns can be defined

[4]FPE means encrypting in a way that the the ciphertext is in the same format as the plaintext. In other words: domain and range are equal, for instance 16-digit numbers or german words.

to further restrict the set of allowed queries. The data owner then generates a symmetric encryption key for every plaintext triple, that encodes the basic key and the bound parts required for the corresponding query. It is used for all unbound parts, that are not already encoded in itself. The data owner then creates query keys for every allowed query, that encode basic keys and restriction patterns. They are distributed to the users and used for the final decryption. The decryption process then is quite simple: if a triple can be decrypted using the custom query key, it fulfills the desired query.

Note that the number of keys in this approach is quite high and every plaintext triple results in eight ciphertext triples. That leads to high overheads in terms of processing (various key generations and encryptions) as well as to obvious storage inefficiency. The authors' evaluation also revealed, that the encryption is relatively slow in practice with only about 15 triples per second. FamilyGuard is more efficient in all of these aspects.

An Encrypted, Distributed, Searchable Key-Value Store by [102]

In addition to the client application and database server nodes this very recent approach introduces an additional entity called dispatcher, that distributes encrypted data to all the database server nodes evenly to build a distributed key-value store. It also handles put/get requests generated by the client. However, the database nodes send the encrypted values directly back to the client. They also maintain local indexes to allow secure querying using secondary attributes of data. Keys are encrypted by the client in a certain fashion that allows the dispatcher to locate the right database server node containing the corresponding values. In order to support more advanced queries than just put and get requests, the authors propose using a set of PPE schemes similar to CryptDB, depending on the desired functionality (meaning deterministic encryption for equality checks and so on), but without organizing them in a layer model.

With FamilyGuard, the dispatcher functionality can be provided by the databases native mechanisms to distribute data.

Arx by [70]

Another very recent approach is Arx, implemented on top of MongoDB. It encrypts each data item with IND-CPA security, mostly using AES. Thus, it provides the same level of security as fully homomorphic encryption or "regular" encryption (that usually is unfunctional for computations).

In contrast to most other approaches Arx introduces two proxy servers, one on client side and one co-located to the database server. The client side proxy has the master key, stores metadata (schema information), rewrites queries and encrypts sensitive data. Arx needs to know in advance what operations will be performed on what fields in order to let the client side proxy maintain the required indexes on the server. When a query is issued the server side proxy receives cryptographic tokens from its client side pendant, that it uses to traverse the index structures on the database server.

The key idea of Arx is to embed computations into special data structures ("Arx-Range" for range and order-by-limit queries and "Arx-EQ" for equality checks) on top of AES, instead of embedding it into PPE schemes. Arx-EQ comes in three versions, depending on the usage of the encrypted values: using SE for regular fields, deterministic encryption for fields containing only unique values and additionally homomorphic encryption, if computation is required on the encrypted fields. However Arx-Range is more complicated. It is based on a B+ Tree, that the server side proxy has to traverse. Since this tree is stored on server side, it is not allowed by Arx's concept to store processable values (e.g. generated using PPE) in its nodes. Instead the authors utilize garbled circuits [101]. Each garbled circuit's output is used for the relevant child's garbled circuit to traverse the tree. Unfortunately these circuits are only secure if used once. Thus, the client side proxy generates and supplies new ones after a query has been processed, which is quite an expensive thing to do.

As can be seen, the price for IND-CPA security of the entire database is quite large, architectural as well as computational. In contrast, FamilyGuard does not need two additional proxy servers or knowledge about queries in advance.

Summary

Due to the different working principles of the database systems in this category the approaches are hardly comparable. So far there are much less approaches for building architectures for encrypted non-relational databases compared to the number of proposed systems for relational databases (where most works focus on extending CryptDB, thus the number of key ideas is relatively small as well). A really practical approach still seems to be missing.

3.1.3. Approaches relying on Fully Homomorphic Encryption

Architectures like CryptDB would not be needed and the query processing could still be done straight forward, if fully homomorphic encryption (FHE) could be used consequently to perform all computations. Indeed it has been proved that it is possible to perform arbitrary computations over encrypted data (and thus arbitrary queries) this way [34]. Unfortunatley, as mentioned earlier (see Section 2.4.4) such constructions are prohibitively computational expensive in practice. For instance, the same authors' homomorphic evaluation of the AES circuit [35] showed a slow down by the order of 10^9 compared to computations on plaintext data. Most recent research in the field [19] makes progress, but still states their own scheme to be "not by itself practical".

3.2. Hardware Architectures

In contrast to the previously discussed approaches there are schemes relying also on cryptographic hardware instead of software only. The following section briefly discusses two examples.

Cipherbase by Arasu et. al [7]

Cipherbase is an extension of Microsoft's SQL Server with two modified parts: the ODBC driver at the client side and the query processor at the server side. The cipherbase ODBC driver holds an 128bit AES key in order to encrypt data, constants and parameters of queries and updates, as well as to decrypt the server's results. It also keeps track of statistics for query optimization. Cipherbase's query processor integrates a secure coprocessor within a so called "trusted machine" on server side, used as a submodule for operations over encrypted data. It is realized utilizing field programmable gate arrays (FPGAs), which are, briefly described, reconfigurable hardware devices. They are equipped with an own encryption key burned into hardware, used to encrypt the application keys provided by the ODBC driver.

Of course this concept requires additional steps for computing a query. Encrypted tuples have to be transferred from the untrusted part of the query processor to the trusted machine, where they are decrypted, processed an then re-encrypted to be transferred back to the untrusted query processor part. After all computations are done the encrypted results can finally be sent back to the client.

TrustedDB by Bajaj et. al [9]

Like Cipherbase TrustedDB aims on SQL processing using trusted hardware in a similar way, but with tamper-proof cryptographic coprocessors (SCPUs) such as the IBM 4764[5] instead of FPGAs.

In brief, the query execution of TrustedDB works as follows. The user creates and populates an SQL schema, marking sensitive data items with the keyword "SENSITIVE" in the queries. The client then sends them through a standard SQL interface, encrypted using the public key of the SCPU. Thus, the database server cannot decrypt the query. The encrypted query is then forwarded to the request handler of the SCPU, that decrypts it and then forwards it to the query parser. During the query parsing a set of execution plans is constructed by rewriting the client's original query into a set of sub-queries. Each sub-query is again classified as either public or private according to their target data set classification of the original query. Afterwards the query optimizer estimates the runtime of each plan and choses the one with least execution cost. Then the query dispatcher forwards public queries to the "regular" database server and

[5] `https://www.ibm.com/support/knowledgecenter/ssw_i5_54/rzajc/rzajcco4758.htm`

the private ones to the SCPU, while at the same time taking care of eventual dependencies. Thereby it tries to move as much work as possible to the database server. Finally, the client receives the results back, assembled, re-encrypted and signed by the SCPU dispatcher.

Summary

Even though hardware approaches like these overcome the limitations of CryptDB-based techniques in many points (particularly regarding the query expressiveness), they rely on expensive trusted hardware at the server side, which makes them unattractive (and in contrast to FamilyGuard not out-of-the-box usable) for today's typical cloud services providers. Note that they also require the database to have the user's decryption keys. Furthermore there are apparently no approaches for non-relational databases in this field.

3.3. A Related Problem: Data Integrity

The main difference between problems of data confidentiality and data integrity is the attacker model they take as basis. The honest-but-curious model describes the adversary as a completely passive entity, that does not manipulate the data items or their transportation to the client. However, there are many possibilities for a malicious attacker to interfere actively, for example: returning incomplete or outdated query result sets. Thus, this tackles not only data privacy, but also the related problem of data integrity, where correctness (did I get the *right* results?), freshness (did I get the *most recent* results?) and completeness (did I get *all* results?) of result sets are in focus. A couple of approaches for detecting issues of that sort are discussed briefly in this section to present the key ideas.

One way to achieve data integrity independently from the database is to integrate mechanisms for data integrity checking already on file system level. [37] proposes doing this using an extra layer on top of the unmodified file system. That allows the file server to remain unchanged. Files are kept in two sections, one contains the file data (symmetrically encrypted content using a unique key per file) and the other one contains the metadata (access control information). Files are also signed, using a unique key per file. While possession of the encryption key is enough for reading data, the signing key is required to also write. Additional "metadata freshness files" are located in every directory, containing the root of a hash tree that was build from all the metadata files in the directory and its sub-directories. The root metadata freshness file gets continuously signed by the client. [59] additionally introduces block servers to store blocks of data that clients interpret as a file system. Here, the file system implementation resides entirely on the client, which requires new storage servers in contrast to [37]. Apart from that the ideas are similar: encryption is done using per-file keys and file integrity checks are implemented using hash trees. Note that both approaches can detect attacks, but do not resolve them.

[66] proposes an approach for integrity checks on XML documents. Therefore the authors introduce an authenticated structure indexing all elements of the XML document (elements as well as attributes). This gets done using a data structure called "Nested Merkle B^+ Tree", a combination of various auxiliary data structures, organized in the fashion of a nested B^+ tree. Complemented by different hash chains the nested Merkle B^+ Tree is formed. Correctness and completeness can be verified by checking the root's hash value and signature similar to the previously discussed approaches. For also guaranteeing freshness of the data the authors use the timestamp value of the root node, that has to match the timestamp broadcasted by the data owner.

[93] proposed a scheme that aims at verifying results of aggregation queries in particular. Database storage providers hosting outsourced databases compute aggregate queries collaboratively without gaining knowledge of intermediate results. The users are able to verify the results of these queries relying only on their trust of the data owner.

[72] introduces "Cloudproof", a storage system that runs on top of cloud storage services like Amazon S3. It is designed to not only detect data integrity violations, but also to prove the occurrence of these violations to a third party. It is realized in the form of a key-value store. The data is organized in blocks with corresponding access control lists, naming users with permissions. The interface provided consists of a get(BlockID) and a put(BlockID, byte[] content) method. When users access cloud storage using these methods, each request and response is associated with an attestation, which then can be used by any client to check data integrity later on. The data owner checks integrity during an auditing process that is performed once in a certain time slot ("epoch"), in which for efficiency reasons, each block is not guaranteed to, but only has a certain probability of being audited. Therefore, users send the attestations they receive from the cloud to the owner, who can detect any violations and construct proofs for convincing third parties.

Summary

The field of data integrity involves data correctness, freshness and completeness. It can be treated without manipulating the data itself (e.g. on file system level or using auxiliary data structures). The problems of data integrity and privacy can be mainly separated by the attacker model (active vs. passive). Interestingly, to the best of our knowledge mechanisms for protecting integrity and (property-preserving) encryption were never combined in a single system.

4 Selecting and Modifying Appropriate Encryption Schemes

The following chapter lays the foundation for the architecture of FamilyGuard as introduced in the Chapter 5. It explains the concepts behind using PPE in the context of databases, gives an overview about existing encryption schemes and introduces the particular requirements of using PPE in practice with WCSs. Considering these requirements a number of schemes are then selected for the use in this work and discussed in detail. However, some chosen schemes still need modifications or leave room for improvements, which are elaborated on as well.

Contents

4.1. Overview

Traditional[1] encryption schemes can provide strong security guarantees (like for example AES provides IND-CPA security). However, using such schemes for encryption unavoidably leads to the loss of certain plaintext characteristics, that database systems rely on for processing data. Mainly three of these characteristics are important:

- *Equality:* After a plaintext has been encrypted using a traditional scheme, it cannot be used by the database for equality checks with plaintexts anymore (and neither for joins, grouping or counting). Note that a special instance of this problem is text search. Furthermore, if different keys or initialization vectors were used, equality checking can not even performed between ciphertexts only.
- *Order relations:* Once plaintexts are encrypted using a traditional scheme, they lose their (usually numerical or lexicographical) order relation. This is a problem for ordering, sorting or finding a minimum or maximum value in a given dataset.
- *Computability:* Ciphertexts produced by traditional schemes cannot be used for computations like additions or multiplications. This is important when it comes to aggregations, for example when computing the sum or average of all values of a column.

This thesis focuses only on aspects regarding equality and order relations, since computations are only needed in a very small subset of most NoSQL database's query mechanisms. For example the `SUM` and `AVG` commands are the only ones in Apache Cassandra's query language, that would profit from computability of ciphertexts. Out-of-the-box HBase even has no use for it at all. Thus, the following sections describe how to select appropriate and feasible schemes in order to enable the database server to perform

- equality checks, grouping and counting, using deterministic encryption (Section 4.2)
- range queries, sorting, ordering, using OPE (Section 4.3)
- text search, using SE (Section 4.4)

Table 4.1 gives an overview on all PPE schemes that are discussed in this thesis.

[1]meaning *not* property-preserving

Table 4.1.: Overview about the PPE schemes discussed and used in this thesis

functionality	approaches
deterministic encryption	[76], [84]
order-preserving encryption	[54], [100], [14], [71], [81], [61], [62], [13], [64], [50]
searchable encryption	[89], [45], [31], [51], [36], [25], [97], [86], [47]

[x] discussed in this thesis, but not practically used due to lack of feasibility
[x] third-party implementation used for this thesis
[x] implementation used for this theses
[x] implementation with own improvements used for this theses

4.2. Deterministic Encryption

4.2.1. Requirements

With deterministic encryption the server is able to learn which encrypted data items correspond to the same plaintext data value, since when deterministically generated, the same ciphertexts are mapped to the same plaintexts. In contrast to OPE and SE, there are no other relevant practical requirements to this type of PPE besides this determinism (except of course for a certain level of runtime performance and security, which will be discussed for the applicable schemes individually). Deterministic encryption is mostly realized using a block cipher (see Section 2.4.1.3).

4.2.2. Applicable Schemes

4.2.2.1. Advanced Encryption Standard (AES) by [76]

Description AES is a symmetric block sipher. It allows block lengths of 128, 160, 192, 224 and 256 bits, as well as key lengths of 128, 192 or 256 bits. It can be used to achieve strong security guarantees, like IND-CPA. Furthermore a lot of widely used Intel and AMD mainstream processors are equipped with AES optimized instruction sets[2], which is why AES usually performs well in real-world applications.

As AES is a well known and publicly available encryption standard for over 15 years now, this thesis will only give a high level explanation of its working principles. AES operates on 4x4 Byte matrices on which it performs transformation rounds. The cipher key length specifies the number of repetitions of these rounds (128 bit = 10 cycles, 192 bit = 12 cycles, 256 = 14 cycles). Therefore firstly, round keys are derived from the cipher key. Afterwards, it follows an

[2]see `https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf`

initial round, in which each byte of the matrix is combined with a block of the round key using a bitwise XOR operation. Then the transformation rounds are performed, each consisting of four steps: (i) each byte is replaced with another one according to a lookup table using a so called "S-box" (where S stands for substitution), (ii) the last three rows of the matrix are shifted cyclically by a certain number of steps, (iii) for diffusion in the cipher, every column's four bytes are combined to another four bytes in a way that each input byte affects all four output bytes using linear transformation with a fixed matrix and (iv) the initial round step. After the required number of transformation round cycles was processed, a final round concludes the algorithm, that is basically like a regular transformation round, but without step (iii).

Weaknesses There are a couple of theoretical attacks on AES, but none of them is of practical relevance with today's hardware capabilities. The most popular and "efficient" one was proposed in [12]. It is only four times faster than bruteforcing and thus poses no security risk. Note that side-channel attacks do not attack ciphers, but the underlying hardware and thus, there are not related to security in the context of this thesis.

4.2.2.2. Blowfish by [84]

Description Blowfish comes with a fixed block length of 64 bit, while the cipher key length can vary between 32 and 448 bit. Similar to AES the first step is to derive round keys from the cipher key. Then the plaintext is split into a left and right half. After that the algorithm continues with 16 rounds, in each of which the following happens: (i) the left half gets XORed with the current round key, (ii) the output of (i) is used as input for Blowfish's round function. The result gets XORed with the right half. (iii) Both halves are swapped. After the 16 rounds are processed both halves get XORed with the round keys of round 17 and 18. Eventhough meanwhile the successor "Twofish" [85] is available, Blowfish is still of special interest for this thesis due to the fact, that its 64 bit block length is enough to cover a lot of basic numeric datatypes (in particular the most ciphertexts produced by OPE, see Section 4.3). That makes it more storage-efficient than Twofish.

Weaknesses So far there are no efficient attacks on Blowfish, when executed with the full 16 rounds. Unfortunately, compared to AES there are not many specialized hardware implementations either.

4.2.2.3. Others

There are a couple of other block ciphers, that would also work fine for this thesis, like RC6 [78], Serpent [6] and the previously mentioned Twofish. However, the discussion and practical usage for this thesis is limited to AES due to its wide adoption and performance as well as to Blowfish due to its storage efficiency. No other schemes are needed.

4.2.3. Inapplicable Schemes

The main reason for a block cipher based scheme to be inapplicable for the architecture of this thesis is a lack of security. A typical example is DES [77]. Once widely used, it can be broken by specialized hardware in a few hours since 2010[3].

4.3. Order Preserving Encryption

4.3.1. Requirements

Due to the general working principles of wide columns stores as described in Section 2.1.2, OPE schemes have to satisfy certain requirements, leading to the following criteria for evaluating their practical feasibility:

(I) Ciphertext (im-)mutability. The ciphertext produced by an OPE scheme is called *mutable*, if it may change as more and more input gets encrypted. An example of this category is [54], described in detail in Section 4.3.2.2. In contrast, there are OPE schemes producing *immutable* ciphertexts. Immutable means once a plaintext is encrypted, the corresponding ciphertext is final. This avoids a possible re-encryption overhead. An instance of this category is [100], described in Section 4.3.2.1.

As discussed in Section 2.1.2 the encryption of row identifiers must be order-preserving to preserve the way data gets distributed in the cluster. The usage of a mutable OPE scheme for the row identifiers would cause row keys to change over time and hence would result in changing the data's physical position inside the database (cluster), which is prohibitively expensive (and thus, generally not supported at all by WCS databases). Hence, at least for row identifiers an immutable OPE scheme is required. However, mutable OPE schemes can still be used for other column data to gain more performance, as can be observed in Section 7.1.1.2. Note that ciphertext mutability is often strongly related to criterion II and V.

(II) Need for additional data structures. Most OPE schemes need to maintain an inner state, for example to maintain plaintext-ciphertext-mappings. Therefore they need to maintain appropriate data structures for indexes, trees, dictionaries etc., either on client side (or at least a trusted enviroment), e.g. [54], or on server side, e.g. [71, 81]. Note that in particular maintaining tree structures is expensive for (non-graph-based) database systems. Hence, additional

[3]see `http://www.sciengines.com/company/news-a-events/74-des-in-1-day.html`

components on server side are sometimes proposed for performance reasons (see criterion III), which makes practical implementations rather complex. Furthermore such architectures do not fit today's usual cloud computing offers well. Hence, OPE schemes are well suited for this thesis, if they use simple and fast accessible data structures for maintaining their state. Note that also a stateless scheme is used later on (see Section 4.3.2.3).

(III) Need for additional architectural components Client applications and database platforms normally do not have built-in mechanisms for OPE. Thus additional components are required for both rewriting queries to make them work with the server side data structures (as they might have to be altered for functioning with the OPE schemes) as well as for performing decryption and encryption itself. Usually those components have to reside in the trusted (client side) environment (e.g. [73, 95]), but some OPE schemes even require components running co-located to the database server (e.g. [71]), which cannot be considered practical due to the architectural overhead. In particular (like in criterion II) SaaS providers usually do not support that. Hence, OPE schemes are well suited for this thesis, if they do not require any further components besides the client application and the database server.

(IV) Input capabilities This criterion can be further sub-divided into three problems:

- **IV-a** The authors of all OPE proposals discussed in this thesis assume only positive integer input for their schemes. This is hard to apply to real world datasets in which we also find negative or floating point numbers. One option to deal with negative input would be adding an offset value to the plaintext space, that is large enough to push every value above zero. The question is how to determine this offset, when the entire plaintext (dataset) space is not known in advance.
- **IV-b** Handling floating point input is an even bigger problem. To our knowledge there is no technique for encrypting floating point numbers in an order-preserving way. This rises the question whether existing OPE schemes can be modified to also work with negative and/or floating point input. We will answer that for the schemes we have investigated in Section 4.3.2.1 - 4.3.2.3.
- **IV-c** Independent from the input type, some OPE schemes further require detailed knowledge of all the plaintexts before encryption (e.g. [61]), which is hard to realize in practical scenarios as databases may grow unpredictably over time. Some schemes even need to encrypt the whole plaintext space D in advance [100, 62], instead of encrypting only the desired values on demand. The unfeasibility of such an approach can be

illustrated easily using the following example: let D be defined by a common Integer datatype. Having a typical length of 32 bit, $|D|$ would be of size 2^{32}, which means 4.3 billion items would have to be pre-computed and stored in an index (even if the majority is never used).

Hence, since to the best of our knowledge there are no OPE schemes available for negative and/or floating point input, schemes are suitable for the architecture of this thesis, if they at least support on demand encryption of arbitrary values of the plaintext space.

(V) Security The first formal security analysis of OPE [13] proved that ideal security[4] with immutable ciphertexts can only be accomplished, if the ciphertext space size $|R|$ is exponential in the plaintext space size $|D|$, which is hard to achieve in practice. OPE schemes deal with this problem in different ways (which often has a direct impact on the criteria II and III). One example is modular plaintext shifting [14], meaning adding a fixed offset to all ciphertexts, starting again from zero when the cipertext space upper bound is reached. This is easy to implement, but only a small security enhancement. Another example is using fake queries to hide the query distribution [64], but that causes communication and computation overhead. In practice ideal security can be achieved more easily by OPE schemes producing mutable ciphertexts, because they do not have the requirement of a ciphertext space size being exponential in the plaintext space size. They also hide the frequency distribution of plaintext-ciphertext assignments much better, thus being able to achieve an almost uniform distribution (as shown e.g. by [100]). Still, that also means dealing with unavoidable re-encryptions of (at least parts of) the ciphertext, that is already stored in the database. Recent schemes try to keep the number of such updates to a minimum [54] or take the burden of reassigning ciphertexts to components on server side [71] to reduce at least communication costs. An alternative approach to avoid re-encryption in the first place is pre-encrypting the whole plaintext space in advance as discussed in criterion IV-c.

For a quick overview and brief evaluation of the schemes that were investigated based on the above described criteria, see Table 4.2.

4.3.2. Applicable Schemes

For this thesis the three schemes from Table 4.2 were selected for implementation and testing, that fulfil the above introduced criteria the best, namely [14, 100, 54]. Detailed explanations of these schemes and our modifications to improve the practical feasibility of two of them are given in the following Sections 4.3.2.1 - 4.3.2.3. For not losing scope of this thesis the concepts of the schemes that were ruled out are not explained in detail, but their flaws are discussed in Section 4.3.3.

[4]meaning IND-OCPA: ciphertexts reveal nothing, but their order

Table 4.2.: Evaluation of the practical feasibility of the OPE schemes discussed in this chapter based on the criteria introduced in section 4.3.1, ordered chronologically by date of publication

Scheme	I	II	III	IV	V
[50]	+	−−	+	−	−− (?[2])
[14][1]	+	++	+	−	+ (POPF)
[61]	+	−−	+	−−	−− (?[3])
[71]	−	−−	−	++	+ (IND-OCPA)
[100][1]	+	−	+	+	++ (> IND-OCPA[4])
[62]	+	−	+	−	−− (?[2])
[54][1]	−	−	+	++	+ (IND-OCPA)
[26]	+	+	+	−	++ (> POPF[4])

[1] scheme used and implemented in this thesis
[2] only rather informal security analysis provided by the authors
[3] no security analysis provided by the authors
[4] ">" = proved by the authors to be better than...

4.3.2.1. Random Subrange Selection using Random Uniform Sampling by [100]

Description In their work the authors introduce not one, but three OPE schemes, namely random offset addition (ROA), random uniform sampling (RUS) and random subrange selection (RSS). Since ROA is somewhat trivial and an attacker only needs to know a single plaintext-ciphertext-pair to break the encryption, this thesis focuses on RSS with RUS being a sub-procedure of it.

RSS can roughly be described as follows. First of all, randomly decide how to draw the lower and upper bounds r_{min} and r_{max} of the ciphertext range R, either by choosing $r_{min} \in [1, |R| - |D| + 1]$ and $r_{max} \in [r_{min} + |D| - 1, |R|]$ or by choosing $r_{max} \in [|D|, |R|]$ and $r_{min} \in [1, r_{max} - |D| + 1]$. Afterwards an order-preserving function (OPF) from $D = [1, |D|]$ to $R = [1, r_{max} - r_{min} + 1]$ is sampled using an alternative OPE construction scheme. Therefore this thesis uses the authors' RUS as described in the next paragraph. Finally add $r_{min} - 1$ to all ciphertexts.

RUS gets initialized with an empty OPF f and the minimum and maximum elements of D and R as specified by RSS before. A recursive sample procedure then randomly selects an element $p \in [d_{min}, d_{max}]$ and $c \in [r_{min}+p-d_{min}, r_{max}+p-d_{max}]$. Thus, p splits D in a lower and a higher sub-domain and c splits R in a lower and a higher sub-range. The pair (p, c) is then added to f and the sample procedure continues recursively as before with the new sub-domains and sub-ranges until D is completely covered.

Practical Strengths and Weaknesses. The following list provides an overview of the Pros(+) and Cons (−) of the RSS scheme when it comes to putting it into practice.

+ Ciphertexts are immutable.

+ No cryptographic primitives are needed, just random selections.

− RSS with RUS can handle only positive numerical inputs.

− It processes the whole domain D at once instead of computing and returning only the ciphertexts for actually desired plaintexts on demand (an example illustrating the impracticability of this approach was given in Section 4.3-IV)

Modifications We now describe our approach to make RSS with RUS practical in real-world applications. We can get rid of the first weakness trivially by initializing the sample function in RUS with a negative value for d_{min} instead of 1. This extends the domain D into the range of negative numbers (as far as we want). Since the algorithm only works with random selections in intervals and some additions and subtractions, that does not affect its working principle. We can eliminate the second weakness by modifying RSS and RUS in the following way.

First of all, we define p' specifying the plaintext value that we are actually aiming for in the encryption process (instead of the whole domain D). We modify the sample procedure of RUS by adding an extra parameter for p'. Now instead of always continuing recursively after a split for the lower sub-domain $[d_{min}, p-1]$ and the higher sub-domain $[p+1, d_{max}]$, we only process the lower sub-domain if $p' \in [d_{min}, p-1]$ or the higher sub-domain if $p' \in [p+1, d_{max}]$. This reduces the average number of sample function executions (in the following short: "samplings") from $|R|$ to $log_2(|R|)$.

Then we modify RSS itself. Instead of always starting with the full domain $|D|$ (which in combination with our RUS sample function modification would result in an inconsistent encryption anyway), we now initialize the sample function of RUS only with the sub-domain $[d_1, d_2]$, in which d_1 is the highest already encrypted value smaller than p' and d_2 is the smallest already encrypted value greater than p'. As more and more values get encrypted, this reduces the average number of samplings further (see Figure 4.2). In order to make that work for the first p' that we would like to encrypt after we have determined r_{min} and r_{max} in the initialization phase of RSS, we add the minimum and maximum pairs (p_{min}, c_{min}) and (p_{max}, c_{max}) to f by sampling c_{min} from $[r_{min}, r_{max}-1]$ and c_{max} from $[c_{min}+1, r_{max}]$.

Figure 4.1 and 4.2 present an example that illustrates the improvements by showing the average number of necessary samplings for computing the ciphertexts of 10000 uniformly and randomly generated 32 bit Integer plaintext values 20 times. Instead of always having to sample $|D| = 2^{32} = 4294967296$ times to cover the whole domain and then pick the 10000 (p, c) pairs that are actually

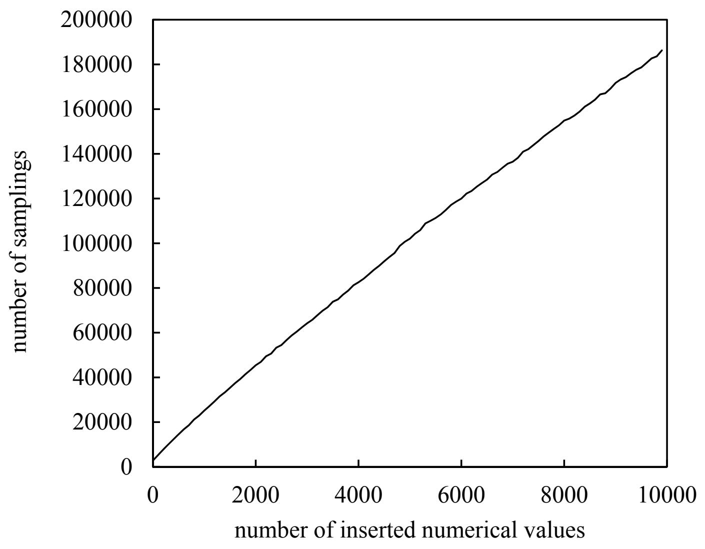


Figure 4.1.: Average number of samplings required in [100] with increasing dataset size.

required, only 186.287 samplings (= 0.004%) for those 10000 values are necessary on average. Note that the implementation done for this thesis is able to do this in less than a second (for details see Section 7.1.1.2). Of course this number decreases, if less values are supposed to be encrypted (see Figure 4.1). Furthermore it can be observed, that as more and more values have been encrypted already, the average number of necessary samplings required per value decreases from the expected $log_2(|R|) = log_2(2^{32}) = 32$ for the first encryption to 21 for the 10000th encryption (see Figure 4.2).

4.3.2.2. Optimal Average-Complexity Ideal-Security (OACIS) OPE by [54]

Description The OPE scheme introduced in [54] can be briefly described as follows. The OPF f is initialized with two plaintext-ciphertext-pairs, namely $(-1, -1)$ and $(|D|, |R|)$. New pairs (p, c) are always inserted between (p_n, c_n) and (p_{n+1}, c_{n+1}) with $p_n \leq p < p_{n+1}$ and $c = c_n + \lceil \frac{c_{n+1}-c_n}{2} \rceil$. If $p = p_n$, the value was already encrypted. If $c_{n+1} - c_n = 1$ there is no gap large enough to accommodate the new ciphertext c. In this case a re-encryption procedure is executed: From all the sorted and distinct plaintexts $p_1 ... p_m$ that have already been encrypted, start over like described above with $p = p_{\lfloor \frac{m}{2} \rfloor +1}$ and continue recursively with the interval $p_1 ... p_{\lfloor \frac{m}{2} \rfloor}$ if $m > 1$ and $p_{\lfloor \frac{m}{2} \rfloor +2} ... p_m$ if $m > 2$.

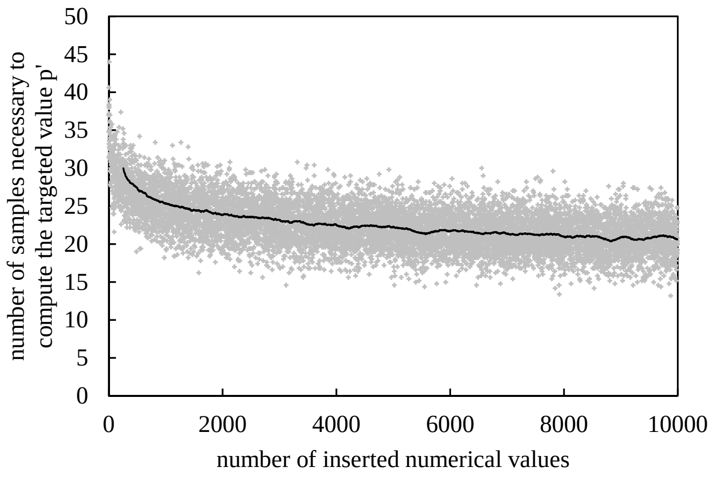


Figure 4.2.: Average number of samplings required per encryption in [100] with increasing dataset size (the black line represents the average of 20 runs, as indicated by the grey dots)

Practical Strengths and Weaknesses. Like done before for the RSS scheme the following list provides an overview of the Pros(+) and Cons (–) of OACIS OPE.

+ This scheme works with very simple computations, that do not even involve randomness. Thus it can be computed very fast.

– Ciphertexts are mutable. Thus, the most obvious weakness is the re-encryption phase, because in practice that means reading all already encrypted values from the database, re-encrypt them and finally write them back into the database. In order to keep the occurrences of those re-encryptions as rare as possible, the ciphertext space should be chosen large enough. Having a plaintext space of length n bits the authors recommend a ciphertext space of size λn bits, with a theoretical safe upper bound of $\lambda = 6.31107$, but they also show in their practical experiments that $\lambda = 3$ (sometimes even $\lambda = 2$) is already sufficient for most datasets.

– Insertion order matters. The best case is when all elements of a perfectly balanced binary search tree are inserted in pre-order traversal order. The average case is a uniform input distribution. The worst case is inserting pre-sorted values, which should be avoided at all (see Section 7.1.1.2).

– By the authors' definition the scheme cannot handle negative input.

Modifications Since we cannot avoid the re-encrytion phase other than by defining the range large enough and we also might not have any influence on the insertion order of the plaintexts later on, the only modification we can apply is initializing f with $(-|D|, -|R|)$ and $(|D|, |D|^\lambda)$, instead of $(-1, -1)$ and $(|D|, |R|)$. Similar to our modification for [100] this extends the domain to also cover negative input. To make sure this does not increase the number of necessary re-balancings, we adjusted the ciphertext space as recommended by the authors.

4.3.2.3. mOPE by [14]

Description mOPE is an extension of [13], which is the only OPE scheme of practical relevance so far (implemented in [73, 95]). It is based on the fact that any OPF from $\{1...M\}$ to $\{1...N\}$ can be represented by a combination of M out of N ordered items. Thus, ciphertexts can be computed by sampling values according to the hypergeometric distribution. mOPE adds a secret modular shift to the encryption in the following way: if DEC_{OPE} and ENC_{OPE} are the decryption and encryption function of the standard Boldyreva OPE [13], then $ENC_{mOPE}(x) = ENC(x+m)$ (where m is a secret offset) and $DEC_{mOPE}(x) = DEC_{OPE}(x) - m \mod |D|$ (where $|D|$ is the size of the plaintext space). For not losing scope at this point, the reader is kindly referred to [13, 14] for more details.

Practical Strengths and Weaknesses.

+ This scheme is the only OPE scheme that does not require maintaining a state at all. That makes it easy to implement the algorithm for client server scenarios. No indexes are required.

− The core element of this algorithm is sampling from the hypergeometric distribution, which is computationally expensive and requires the input to be a positive integer value.

4.3.3. Inapplicable Schemes

To give an idea of why other OPE schemes from Table 4.2 have been considered impractical, a few of their characteristics that cannot be read from this table are pointed out in the following paragraphs.

The approaches of [49] and [62] require splitting and partitioning of the plaintext space. Hence, they have to keep track of more metadata than most other schemes. Furthermore the pre-processing of query conditions is quite complex, in contrast to most other schemes that simply replace plaintext values with ciphertext values in the query.

The scheme of [61] requires detailed knowledge of the plaintext space. In particular it needs to know the smallest distance between two input values for adding random noise to the ciphertexts in a way that does not corrupt the original order. As mentioned before, in practice usually there often is no detailed information about the plaintext in advance. Furthermore, this is a problem when encrypting floating point numbers, since in theory the minimum distance between such numbers can be arbitrarily small. Moreover, the indexing mechanism is complicated, working with an index of indexes and storing a single plaintext value requires storing three values in the database (for a hash, a index value and an encrypted representation).

The approach of [71] needs an additional component running co-located to the database server, which the authors call "OPE-server". It is responsible for performing re-encryption operations as described in Section 4.3-I and holds two data structures: a tree containing deterministic ciphertexts in each node and a table mapping these ciphertexts to OPE encodings. In real world scenarios running additional applications on the same platform as or co-located to the database server often is not possible or does not fit the offers of cloud database providers. Furthermore it makes this approach very expensive in terms of network communication overhead.

The approach of [26] (calling it order-*revealing* encryption) completely lacks a practical decryption functionality. Instead it comes with a custom compare operator. Thus it is not applicable for a database scenario, since it surely would preserve the order of the plaintext, but their exact values would not be recoverable with regular query mechanisms. Instead, decryption requires applying the compare operator multiple times and then traversing a binary tree. Alternatively the authors propose to store every encrypted value together with a CPA-secure encrypted representation of the plaintext in order to avoid the tree traversal.

4.4. Searchable Encryption

4.4.1. Requirements

Like previously done for OPE this chapter discusses practical requirements of schemes for SE in the context of databases. However, in contrast to the previous chapter, some restrictions can be made in the selection process of available schemes, given by the database scenario of this thesis. That significantly limits the number of possible candidates in advance. Thus this thesis does not discuss schemes involving

- *fuzzy search* - Query mechanisms for NoSQL databases do not support fuzzy search functionality so far[5] (unlike SQL's `LIKE` operator, that supports at least wildcards). Hence, most applications that rely on NoSQL

[5]with rare exceptions, for example MongoDB and HBase realize fuzzy search with regular expressions, which has major impacts on query runtime.

technology today are not designed to work with fuzzy search. They ask questions to the database and expect exact answers, not fuzzy ones.

- *multi-keyword search*[6] - Searching for multiple keywords at the same time could improve the performance of a small subset of possible queries, but like fuzzy search schemes, multi-keyword search is very complex and requires a lot of extra pre-processing and data structures. Thus, for this thesis the extra effort is in no relation to the benefits. However this can be considered an option for future work (see Section 8.2).
- *multi-user functionality* - User rights management in the scenario of this thesis is performed on another (higher) level (e.g. by the database itself or a frontend), rather than encoded in the layer for searchable encryption. Other layers (for example OPE) do not support it either, which would be necessary for a consequent implementation.
- *ranked search* - In the scenario of this thesis ordering search results is task of the database, since the user might have specified certain criteria for that in the query. It cannot be left to the searchable encryption layer.

An evaluation of practical feasibility of the remaining schemes for SE can be done based on the following criteria (some of them similar to the criteria for OPE):

(I) Need for additional data structures. Searchable encryption can be done in two ways. On the on hand, the plaintext can be encrypted directly in a certain fashion. That means the ciphertext can be queried directly as well, but since every word has to be checked for a match, this results in a search time linear to the length of the plaintext data. That is why, on the other hand, indexes can be used to significantly speed up the search process and achieve sub-linear search time (Section 2.4.2.1 described, how indexes can be constructed). Unfortunately using indexes also has downsides. It comes with the cost of additional preprocessing steps (e.g. selecting keywords, set up the index data structure, etc.), requires index maintenance and often an extra round of communication (first for querying the index and then for getting the actual results). Furthermore the underlying data structure of the index should be manageable for the database without too much effort (for instance tree structures are hard to maintain within a WCS, but easy for a graph database). Hence, a good SE scheme in the context of this thesis has either no index at all or an index with a data structure that is simple to maintain.

Interestingly, the scheme of Song et. al [89] (see Section 4.4.2.1) is the only approach that does not rely on an index. However, other schemes even combine different index types (as explained in Section 2.4.2.1, for example the approach of Hahn et. al [45], see Section 4.4.2.2).

[6]sometimes also referred to as *conjunctive keyword search*

(II) Support for Updates. When used in databases, a scheme for performing SE needs the ability to process updates, since in most practical cases it is unlikely that the complete data set is known in advance or remains completely unchanged once written to the database. This can be a problem for index based schemes, since new document identifiers or keywords have to be added to existing indexes. Thus, schemes requiring the exact number of documents or keywords per documents (see Section 4.4.3) prior to encryption can hardly be used, except for so called "write once"-databases[7]. By design the problem of updates is only a problem for index-based schemes. SE schemes supporting update operations are usually referred to as "dynamic" schemes.

(III) Performance Requirements Checking for matches does not work with simple operations like in DET or OPE. Instead, more complex procedures are necessary, dependent on how the scheme was designed. This can involve performing lookups in auxiliary data structures like bloom filters or traversing trees, using cryptographic primitives or concatenate strings. For efficiency reasons this cannot become too complex and should involve only data structures that can be held by WCSs without too much effort.

(IV) Security vs. (V) Search Efficiency. A broad introduction into security considerations for SE was already given in Section 2.4.2.1. Note that security for index-based schemes and not index-based schemes is hardly comparable due to the information leakage that comes with querying indexes. Thus, in practice there are two competitive requirements here: avoiding an index leads to more security but is generally slow for searches. Using an index may slow down encryption and leaks additional information, but speeds up querying significantly (see Section 7.1.1.3).

For a quick overview and brief evaluation of the schemes that were investigated in this thesis based on the above described criteria, see Table 4.3.

4.4.2. Applicable Schemes

This thesis uses the two schemes for SE that appear to be best suited in Table 4.3. Both follow very different approaches.

Firsty, the scheme proposed by Song et al. [89] (short "SWP" scheme, which is an abbreviation for the author's names Song, Wagner and Perrig) is one of the first approaches to searchable encryption at all. It is based on linear scans, which means, it most likely will be not as fast as index-based schemes, but is has a couple of features that still make it interesting to use in a practical environment.

[7]like the name suggests: databases that are written to only once and then solely queried

Table 4.3.: Evaluation of the practical feasibility of the SE schemes discussed in this chapter based on the criteria introduced in section 4.4.1, ordered chronologically by date of publication

Scheme	I	II	III[2]	IV	V[3]
[89][1]	++	++	− (XOR, PRF)	++ (IND-CPA)	$O(n)$
[36]	−	++	++	−− (IND1-CKA)	$O(d)$
[25]	−−	+	++	−− (IND2-CKA)	$O(d)$
[31]	−−	−−	−− (XOR, SC, DED)	+ (IND-CKA2)	$O(m)$
[86]	−	+	− (XOR)	+ (IND-CKA2)	$O(log(u))$
[97]	−−	++	−− (XOR, PRF, HSH)	+ (IND-CKA2)	$O(log(u))$[4]
[51]	−−	+	−− (XOR, PRF, HOM)	+ (IND-CKA2)	$O(m)$
[45][1]	−	++	− (SC, PRF)	+ (IND-CKA2)	$O(n/u)$
[47]	−−	−−	−− (XOR, SC, DED)	+ (IND-CKA2)	$O(m)$

[1] scheme used and implemented in this thesis
[2] XOR = perform bitwise exclusive OR operations, PRF = execute pseudo-random functions, SC = string concatenation, DED = perform deterministic encryption/deccryption, HOM = perform homomorphic encryption, HSH = compute hash functions
[3] searching on a dataset with size of n words (u of which are unique) in d documents, resulting in m matches
[4] more precisely: $O(log(u) + N/2 + l)$, with N = length of the used hash chain, l = average number of encrypted document identifiers added since the last search

Secondly, we focus on a fairly new scheme proposed by Hahn and Kerschbaum [45] (short "SUISE" scheme, which is the authors' abbreviation for "securely updating index-based searchable encryption"). It is index-based and thus expected to perform better than the SWP scheme, but it comes with the price of building, maintaining and storing two indexes.

In this section we briefly introduce both schemes and discuss their strengths and weaknesses in practical environments.

4.4.2.1. SWP

Overview. [89] is to our knowledge the only scheme for searchable encryption, that has been implemented for the use with real world database software before [73] (see Section 3.1.1), namely for Postgres and MySQL, which are in contrast to the databases considered in this work both SQL-based.

The SWP algorithm is basically the only choice when it is desired to avoid having an index (e.g. for practical reasons) [18]. It is presented in four schemes. The first one is the basic scheme, that is already provably secure. The *encryption phase* basically works as follows: every word W in the plaintext is modified by padding and/or splitting to have a length of n bytes. For every one of those words a pseudorandom value S of length $n - m$ bytes (with $m < n$) is created using a pseudorandom number generator G. S is then used to compute a (hash)value $F(S)$ of length m using some key k. S and $F(S)$ are now concatenated and after that XORed with W to produce the final ciphertext C.

The *search phase* is rather simple then. If the search aims for a certain word W, it just has to be checked for every ciphertext word C whether $W \oplus C$ is of the form $S||F(S)$ for some S. If so, there is a match.

All the other schemes presented by the authors are extensions of that basic version to provide additional features. Thus the second scheme ("controlled searching") extends the basic scheme in order to allow the data storage provider to search only for desired search words. Therefore it modifies the key generation process to be dependent on the encrypted words. The third scheme ("hidden searches") takes care of the fact that the data owner has to reveal the search word to the data storage provider in order to search for it. To address this issue it adds a pre-encryption step $E(W)$ to the scheme. Unfortunately considering the extension of the second scheme that means the key generation now depends on pre-encrypted words with the consequence that data owners are no longer able to decrypt their own data. The last scheme ("final scheme") fixes this problem by splitting the pre-encrypted word $E(W)$ in two parts, one of which can be reconstructed, knowing the seed of the used PRG G. As only this very brief overview can be given here without losing the scope of this thesis, we refer to the original work [89] for more detailed explanations. We only further discuss the aspects that are relevant for the architecture introduced in Chapter 5. This thesis focuses exclusively on the "hidden searches" version, because it includes all possible security features and the ability to decrypt is not needed, since other copies of the data will be available in different encryption layers (see Chapter 5.2.1). Thus, whenever the SWP scheme is referred to in the following paragraphs, the "hidden searches" version is subject of discussion.

Practical Strengths and Weaknesses. The following list provides an overview of the Pros (+) and Cons (–) of the SWP scheme when it comes to putting it into practice.

+ The SWP algorithm does not require any state maintenance. Due to its nature as linear scan algorithm there is no need for storing an index on server side or other state information, that would require any additional space besides the data to be encrypted itself.

+ In contrast to most index-based algorithms the SWP algorithm is able to deliver not only information on whether a document contains a search word or not, it can also tell how often and where exactly the search word occurs in a document.

+ The SWP algorithm is very easy to handle for the underlying database (as we will discuss further below).

– As being typical for a linear scan algorithms, searches can take a long time for large datasets.

– The SWP algorithm assumes that all words in the plaintext have a length of n bytes. In practice shorter words get padded, longer words are split in multiple parts of n byte length (while if the last part is too short, it gets padded again). Thus the ciphertext can become much larger than the plaintext, depending on n. We discuss below, how this effect can me minimized.

Practical Issues When putting the SWP algorithm into practice there are a few things to consider the authors did not mention explicitly. This section presents our critic view on these issues one has to be aware of when implementing it.

- **How to split words.** The first issue comes with the fact that the SWP algorithm encrypts the plaintext word by word. That means there has to be some kind of definition where exactly a word ends. That is important, because after the encryption is done, only those exact words can be searched for. This problem can get very tricky. Consider the following example: a simple solution that comes to mind intuitively might be to use just whitespaces as markers for word's ends. But if one only relies on that, the data owner would not find the word "ends" in the previous sentence, because it would have been encrypted as "ends." So other characters have to be considered to be markers for a word's end as well. For example if one wants to be able to find the word "ends" even though it is followed by a ".", one might be tempted to use the "." as marker, too. That would raise a new problem. For example in case the plaintext contains email addresses of the form "abc@xyz.com". The algorithm would split that into "abc@xyz" and "com". The data owner would not be able to search for complete email addresses anymore. Other types of plaintexts contain similar challenges, which means markers for signalizing a word's ending have to be chosen very carefully, depending on the plaintext itself and what the data owner wants to be able to search for later on.

  There are basically two solutions to overcome this problem. The first one makes use of regular expressions. They can be used to check for semantics of words during the encryption phase. For example a regular expression could check, whether a word represents a valid email address, so that splitting it could be avoided. Depending on the plaintext one might choose a certain number of regular expressions to check for. Of course this would slow down the encryption process significantly. Hence a different solution might be more feasible. Besides whitespaces one could just add more characters to the list of markers that define word endings. Obviously this method is not as flexible and sophisticated as the first solution, but it is much faster and with a careful adjustment to the plaintext the results can still be satisfying.

- **How to choose a good word length n.**

  The SWP algorithm assumes a fixed word length n throughout the whole plaintext. That means smaller words have to be padded and larger words have to be split (maybe multiple times). Padding leads to more bytes in the ciphertext, while splitting words leads to more words in the ciphertext. That means the ciphertext is larger in terms of the amount of words as well as in size in bytes compared to the plaintext. The exact impact of both depends on the actual n.

  With a growing n there are less words needed to be split, so the amount of additional words due to splitting during the encryption phase decreases. If n gets bigger, every word in the ciphertext gets bigger as well, no matter how small it was originally in the plaintext, which means the size of the ciphertext increases. Less words mean the encryption/decryption process can be done slightly faster, but the ciphertext will need more space in the database. In contrast, a small n means a smaller ciphertext in terms of size, but also more chunks of split words, which cause more iterations when encrypting and searching and thus, more runtime. This issue gets investigated in detail in Appendix A.1.

- **What is needed for decryption.**

  According to the scheme description in [89] everything necessary for the data storage provider in order to perform a search is a searchword and a (number of) key(s). In practice it is necessary to reveal n and m as well. Without the data storage provider knowing the word length n, it is hard to do the decryption. A couple of decryption phases with n running through a certain range would be necessary. Based on the number of matches the data storage provider then would have to guess the right n. Note, that false positives are possible and there is no way to tell n if there are no matches at all. Basically the same applies for m. During the decryption phase there is a check required looking for whether $C_i \oplus W_i$ is of the form $S_i||F(S_i)$ or not. That can hardly be done, because without knowing m there is no way to tell where S_i ends and $F(S_i)$ begins. In conclusion that means either data owner and data storage provider agree to certain values for n and m in advance or n and m have to be provided in addition to key and searchword at least once per ciphertext. For the rest of this thesis we use $n = 8$ and $m = 4$ as default values, as obtained in the tests presented in Appendix A.1.

4.4.2.2. SUISE

Overview. In contrast to the SWP scheme the SUISE algorithm basically works with two indexes: γ_f and γ_w. γ_f is a *forward index* storing all the unique words per document in an encrypted form. γ_w is an *inverted index.* Once a word has been searched for it stores the identifiers of all the documents where the search word occurred.

Adding documents during the *encryption phase* basically works as follows: On the client side a list of unique words $(w_1, ..., w_n)$ per document f is created. Then a PRG G creates an equal amount $(s_1, ..., s_n)$ of pseudo random values. Afterwards for every word w_i three steps are performed: (1) a search token r_w is created using a PRF F and a key k_1: $r_w = F_{k_1}(w)$, (2) if that search token has been used previously, it is added to a list x and (3) a representation $c = H_{r_w}(s)||(s)$ is created, where H is a random oracle. The server then stores all c's of f in γ_f. Additionally, the identifier of f is added to the inverted index γ_w for all entries of x.

Besides the indexes the originating documents are stored encrypted, using a block cipher (e.g. AES). Therefore different cryptographic keys should be used, otherwise the ciphertexts leak equality (which they do in the author's original publication).

Searching for documents containing a word w then works as follows: first, the search token $r_w = F_{k_1}(w)$ is computed. If there is an entry for r_w in γ_w (which means the word has been previously searched for), then the search can simply be finished by just returning the list of file identifiers that is stored in γ_w for r_w. Otherwise that list has to be created first: every c in every entry of γ_f is split into $l||r$ and then checked if $H_{r_w}(r) = l$ (note that this concept is not much different from the SWP algorithm). If so, the according file identifier is added to the result list, which is then stored in γ_w to accelerate future searches.

Practical Strengths and Weaknesses. Like for the SWP scheme the following list provides an overview of the Pros(+) and Cons (–) of the SUISE scheme from a practical perspective.

+ Searches for previously used searchwords can be done in constant time.

+ The encryption process only needs as much itereations as there are unique words per document. In contrast, the SWP scheme has to carry out the necessary encryption steps for the entire data set.

– Despite the indexes searches for previously not used search words still require linear time like in the SWP scheme.

– The index γ_f stores encrypted representations of all unique plaintext words for every document. As described above those representations $c = H_{r_w}(s)||(s)$ are in most cases much longer than the original plaintext words, which means depending on the plaintext γ_f can become quite large in terms of disk size it utilizes (actually much larger than the plaintext size itself).

– It is part of the algorithm to create a list of unique words per document, which means the algorithm can by design only deliver the information whether a searchword occurs in a document or not. This is less information than provided by the SWP scheme.

Practical Issues There are no practical issues similar to the ones described for the SWP approach, except for the problem of how to split the plaintext into searchable words. Since that is the same issue as for SWP, it is not discussed again at this point.

4.4.3. Inapplicable Schemes

Like previously done for OPE schemes, this chapter gives an idea of why other schemes for SE from Table 4.3 have not been considered practical, even though their authors usually claim that they are.

[36] proposes to use bloom filters as indexes per document. Each distinct word gets pre-processed by a PRF twice and is then inserted into the bloom filter, mapped by a number of independent hash functions. Additionally a unique file identifier is taken into account to make sure, that the same documents do not lead to the same bloom filter contents. Querying bloom filters is by design connected to a certain probability of false positives, which can be encountered by padding the plaintext words to a certain length and using very large bloom filters, both at the cost of less overall performance in practice. Furthermore most databases are not able to perform bitwise operations on bloom filters.

[25] proposed a similar approach but with using pre-build dictionaries, that store the mappings from keywords to bloom filter positions. They can be stored either on client side (which defeats the purpose of remote searchable encryption to a certain degree) or on server side. [36] (providing the new definition of “IND1-CKA” security) as well as [25] (later providing “IND2-CKA”[8]) do not guarantee that the database cannot infer information about keywords from trapdoors. Thus, their security definitions are considered weak in the context of SE and (as pointed out later by [31]) are incorrect[9].

The approach of [31] achieves sub-linear search time. Unfortunately it relies on (for a database) complicated index structures: Firstly, it requires an array, who’s elements represent a number of scrambled linked lists, each of which stores sets of document identifiers from the input document collection, containing its

[8]The main difference between IND1-CKA and IND2-CKA is the fact, that IND1-CKA does not hide the plaintext document’s size, while IND2-CKA does

[9]meaning insecure schemes can fulfill them (which is why there were not discussed in Section 2.4.2.2)

unique words. Since the elements of that array also contain encrypted pointers to other list elements and keys for decrypting them, it quickly becomes very large. Secondly, a lookup table (based on another special data structure: a FKS dictionary [33]) is required to identify the first node in the array for a particular word being searched for. The index is created per unique word from the document collection and it is linear in the number of distinct words per document. Due to the fact that all unique words of the dataset must be known in advance, updates are not supported, (at least the authors do not provide an algorithm for that). Thus, while this scheme might be suitable for "write once" databases, it is not practical for dynamic databases.

Being aware of that problem [51] proposed an extension, in which two additional data structures are introduced. Firstly, another array structure ("deletion array") helps to recover pointers to the nodes that correspond to the files being deleted. Secondly, the "free list" keeps track of what positions in the main array are free and can be filled with new elements. Furthermore, pointers are now encrypted using homomorphic encryption in order to modify them without having to decrypt them. While all these measures improve the update performance, this extension brings even more metadata that has to be maintained, than the original approach. Furthermore, it requires the server to perform homomorphic encryption. Thus, the original scheme can not be considered more practical with that extension.

Another recent extension of [31] is the approach of [47], trying to make range queries more efficient. The key idea is to connect the linked lists of "adjacent" keywords, that are scrambled hidden in the main array. That creates one large linked list with the same size like the main array. The problem is, that the trapdoor of the first word of the desired range can then be used to traverse all the way to the end of the array. Hence the list is split up in two halfs, the first of which is only traversable in forward direction, while the other one always points backwards. If the endpoint of the desired query range lies within the second half this guarantees the appropriate list traversal. Of course, this only works, if start and endpoint of a range query are located in different halves. To make sure that this is always the case, the authors need to further split the list in order to create sub-lists. To cover every possible case, l list are needed for a dataset of size 2^l. While this technique indeed speeds up range queries, the main disadvantage of the original scheme remains: is does not allow updates.

[86] proposes storing a searchable representation of each unique keyword at server side, consisting of the encrypted keyword itself, a masked version of the list of identifiers of documents that contain the keyword and a part of a secret that is necessary to unmask this list. When realizing these lists as bit vectors, updates and masking can be performed using bitwise XOR operations, which relieves the server of having to perform cryptographic computations. While this is indeed a unique feature for an index-based IND-CKA2 secure scheme, note that inserting new data items may require updating the masked lists, which

makes not only searches interactive[10], but also storage (in contrast for example to the also index-based approach of [45] that is used for this thesis). Furthermore depending on the number of unique keywords, the bit vectors can become very large.

Being aware of these problem, the authors of [97] proposed an extended version of that approach, getting rid of the interactivity and long bit vectors. Unfortunately (without going into further detail to not lose scope at this point), this requires a counter for every keyword on client side as well as computing PRFs and hash values on server side.

[10]meaning it requires mltiple rounds of communication between client and server

5 Architecture of FamilyGuard

This chapter describes the Architecture of FamilyGuard. It starts with a general overview of all participating entities, followed by a detailed view on how to store PPE-encrypted data on server side. It then discusses the management of auxiliary data in form of metadata and encryption keys, that is necessary to operate on the encrypted database content. Afterwards, the chapter presents, how interacting with the encrypted data works and introduces the concrete API of FamilyGuard.

Contents

5.1. Concepts and Overview

FamilyGuard aims for executing queries over encrypted data in WCSs. Therefore it uses the basic concept of onion layers (see Section 5.2.1). However, it does not follow the approach of using a proxy server between the application and database for re-writing queries for the encrypted database, decrypting query results, etc (like e.g. CryptDB [73] does). Instead it introduces a simple application programming interface (API) taking care of these tasks, that is used by the client application. This approach has various advantages over the proxy model:

+ A third entity besides client and server can be avoided, which results in less computation and network overhead.
+ In contrast to the SQL world, where most of the approaches described in section 3.1.1 make use of the fact, that the majority of SQL queries use a well defined (and rather small) subset of SQL commands, some NoSQL databases do not even have query languages. Thus, there would be no uniform way for a proxy to manage incoming requests. The API of FamilyGuard hides the complexity of the databases' native APIs. The user does not has to deal with it.
+ The same is true for the realization of the WCS data model (as introduced in 2.1.2). It is shared by all WCSs, but implemented differently in some aspects (see Table 2.1 and Section 5.2.5). FamilyGuard unifies the data access.
+ The user is able to configure the parameters of the used PPE schemes much more fine-grained and individually, while a proxy can only work with standard parameters.

The only disadvantage is that the client application itself has to be changed to use the methods of FamilyGuard. However, the effort for doing so can be considered to be very small (see Chapter 5.6.3).

Figure 5.1 shows the overall architecture. The client application using the API of FamilyGuard 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. Since this data has to be stored persistently, it is kept outside the application in the client machine's file system. The API manages the database connections, data transfer, encrypting and decrypting. Furthermore it keeps track of metadata and key management. FamilyGuard utilizes advanced (index-based) encryption schemes, which allow the system to scale better when datasets become large. Partly, they even provide new functionality (e.g. the ability to search for single words without the need of secondary indexes). The database server never sees any decryption keys, hence it is never able to decrypt private data. Thus, any adversaries (e.g. administrators) are not able to gain sensitive information only from read access. The database server does not require any changes in order to work with FamilyGuard.

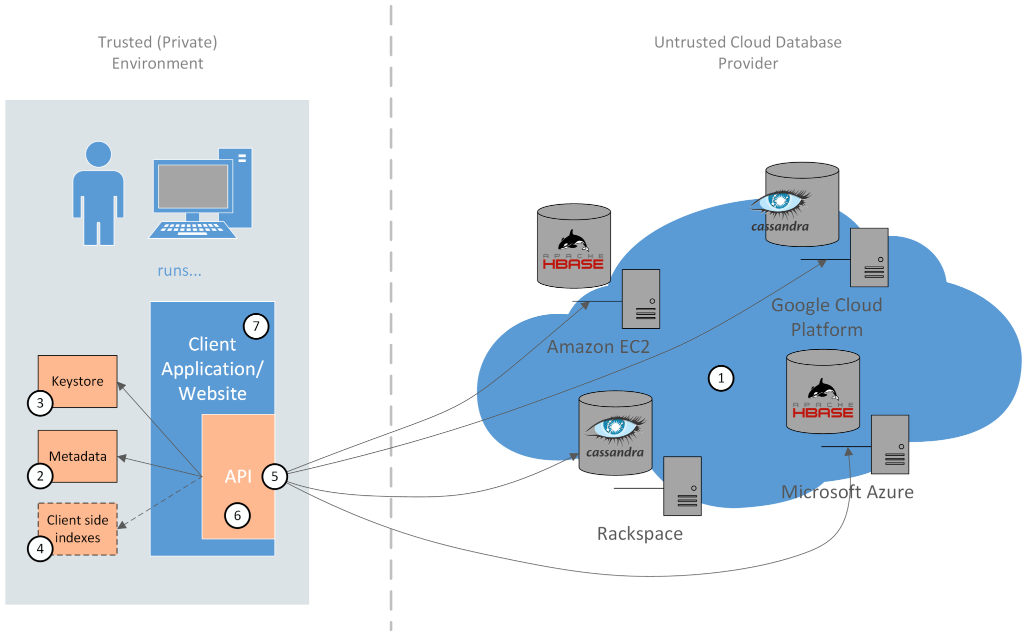


Figure 5.1.: Overall architecture of FamilyGuard

Figure 5.1 can be considered a map for this chapter with the numbers pointing to corresponding sections in the following way:

1. Section 5.2 describes how encrypted data is managed on server side. This includes the introduction of the onion layer model for the WCSs, enhancing security via further distribution of data and options for fine-tuning the encryption process.
2. Section 5.3 explains the structure of the metadata neccessary to manage the tasks described in Section 5.2.
3. Using PPE and onion lyaer encryption can require a large amount of encryption keys. Section 5.4 presents their management.
4. Explanations of the client side indexes can be found in the descriptions of the used PPE schemes, see sections 4.3 and 4.4.
5. Section 5.5 explains how PPE encryption and the onion layer model work together in the process of reading from and writing to the databases.

6. A comprehensive description of the working principles and available commandos of the API can be found in Section 5.6.

7. Finally Section 5.6.3 describes how the API can be applied in a client application.

5.2. Managing Encrypted Data on Server Side

5.2.1. Onion layers in WCS

The concept of onion layers was introduced in CryptDB by [73], also calling it *adjustable query-based encryption.* The idea is to encrypt every value with a PPE scheme that leaks just enough information to still be able to perform certain query operations on the encrypted data. In contrast to CryptDB's design for SQL-based databases, this thesis focuses on the data model and the operation set of WCS databases. While at first glance both seem very similar, they have some fundamental differences, that affect the designs of the onion layer model. Furthermore, the used PPE schemes as well have a certain impact. Hence, the influence of both aspects on the onion layer design is discussed in detail in the following sections.

5.2.1.1. Required Onions

This thesis uses four types of onion layers. Except for the random encryption layer, the purpose of the PPE schemes they are using was already described in Chapter 4. The following paragraphs describe the onion layers 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, see Chapter 2.4.2.2). Two equal plaintext values are mapped to different ciphertexts with a very high probability. This is achieved using AES (CBC) or Blowfish with randomly generated encryption keys and IVs. Every row of a table has a column for storing its own individual IV ("RND Row-IVs" in Figure 5.2) 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 5.2, see also "column key" later on in Table 5.4). 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, again see Chapter 2.4.2.2). 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 process query conditions (see Section 5.5.2).

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. As discussed in Chapter 4.2.2, 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 5.2 and Table 5.3 later on). Since the used WCSs do not support join operations, managing the same keys and IVs for multiple tables is not necessary.

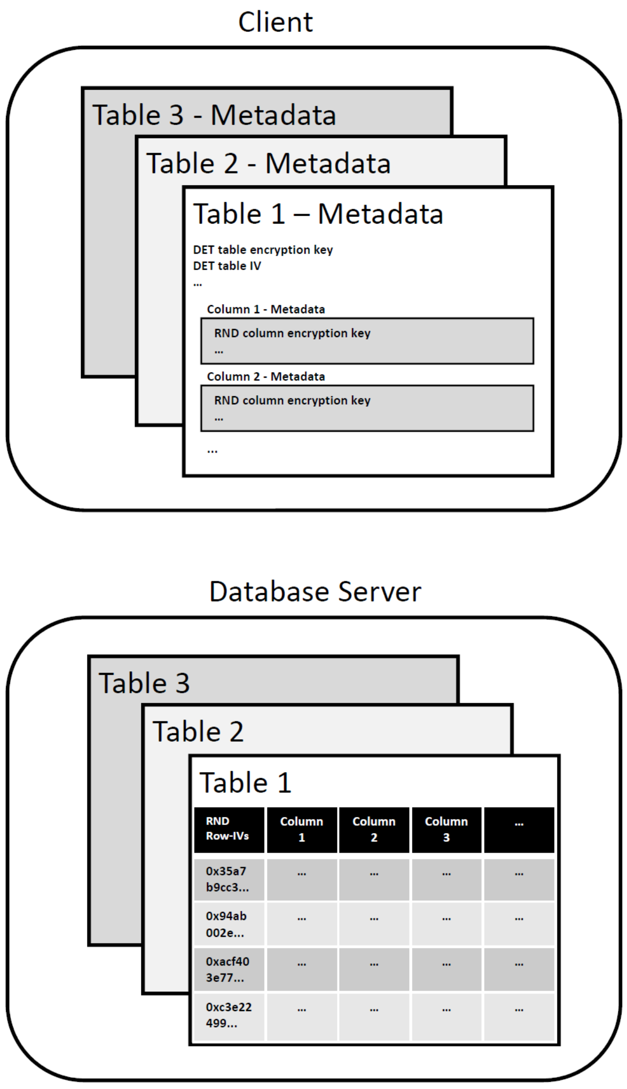


Figure 5.2.: Encryption key and IV management of the RND and DET layer

OPE - Order-Preserving Encryption The onion layer for OPE is realized using the OPE schemes introduced in Chapter 4.3.2: RSS, OACIS and mOPE. Their different strengths and weaknesses qualify them for different tasks within the encryption of a table. RSS and mOPE have to be used for the OPE layer of columns that hold a table's row identifiers (see Chapter 2.1.2, not to be confused with the column that holds the RND layer IVs), because their ciphertexts are immutable. Mutable ciphertexts as produced by OACIS are not allowed here for the reasons discussed in detail in Chapter 2.1.2 and 4.3.1-I. mOPE can be used further, if client side indexes are to be be avoided. Thus it is the most storage-efficient solution, that can be used for the OPE layers of all columns, but it is also the computationally most expensive OPE scheme. RSS is a good allrounder, as it can be also used for all columns and is faster to compute, but it requires a client side index. As mentioned before, OACIS cannot be used for row identifier columns because of its mutable ciphertexts, but as it does not even involve pseudo randomness, it has the best runtime properties of the available schemes, which makes it especially attractive for large amounts of data. Only pre-sorted input has to be avoided and a client side index is needed with OACIS (as discussed in Section 4.3.2.2). Section 5.2.4 explains how the available OPE schemes can be chosen in the encryption process.

SE - Searchable encryption The onion layer for SE is realized using the SE schemes presented in Section 4.4.2: SWP and SUISE. 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. That also means it has a search time linear to the dataset size, since it requires reading the data entirely when searching. If storage space is not an issue, that can be avoided by using SUISE for the price of a very small client side index and a rather large additional server side index, as discussed in Chapter 4.4.2.2. SUISE is especially helpful when certain queries are supposed to be executed frequently, as it can return results beginning from a second search for the same keyword in constant time. Similar as for the OPE schemes, Chapter 5.2.4 explains how the available SE schemes can be chosen in the encryption process.

These four layers result in three onions, as can be seen in Figure 5.3.

In contrast to SQL-based databases, WCSs like Cassandra and HBase do not allow join operations of any kind and have only very limited capabilities of aggregating data items by design. This simplifies the onion design in the way that other onion layers are not required, in particular:

- Onion layers for join operations: CryptDB introduced a JOIN layer for equality join operations and a JOIN-OPE layer to enable joins by order relation. WCSs do not need both of them. However the JOIN layer in CryptDB is also responsible for detecting repeating values in different

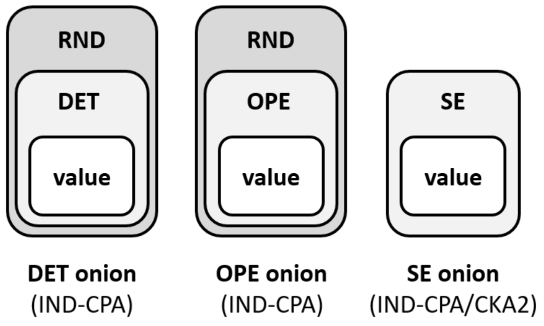


Figure 5.3.: Onions used in FamilyGuard

columns of the same table, which is a functionality that is needed in WCSs as well. This thesis solves this problem in the layer for deterministic encryption by using the same cryptographic key and IV throughout the entire table (both stored in the table's metadata at client side).

- Onion layers for homomorphic encryption: The only options provided for aggregating data (and thus, making use of homomorphic encryption) in Cassandra's CQL are `SUM` and `AVG` operators for calculating sums and averages. HBase does not even have any aggregation functionality in its native API. Furthermore, since homomorphic encryption is very costly in terms of runtime (see chapter 2.4.4.2), having an extra onion layer for it means a lot of effort for very little benefit.

5.2.1.2. 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 hinder this approach.

Limitations caused by the WCS data model While the lack of join capabilities of WCS databases seems to make the onion design easier compared to SQL-based databases, their data model comes with another challenge. As explained in Section 2.1.2 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 can not be used, since its outermost RND layer does not allow order comparisons. Using it anyway 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 therefore, they are not allowed to have a RND layer as outermost layer.

Limitations caused by query operators Depending on the type of data it makes no sense or it is not possible/too costly to encrypt and maintain all onions. For the sake of simplicity we distinguish between three types of data in this thesis: strings, numerical values and byte blobs. All other data types are strongly related to or 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 within numerical values. It is not even apparent, how searching within numerical values could be defined or what benefit it might have. Thus, SE is useless in that case. The same holds for using SE onions with byte blobs. More examples are shown below.

Limitations caused by OPE As explained earlier in Section 2.4.3 OPE schemes work by mapping values from a plaintext space ("domain") to a ciphertext space ("range"). Thereby 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 [61], 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[1]. Assuming a small maximum allowed string length of only 4 bytes "abc"[2] 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 even though the input string was very small. Storing the above number would require 31 bits[3] (which is not really surprising given an input of 4 characters, meaning $4 \cdot 8 = 32$ bits). Limiting the message space to the 96 actually allowed and printable characters of the ASCII table barely improves the situation: $97 \cdot 96^3 + 98 \cdot 96^2 + 99 \cdot 96^1 + 0 \cdot 96^0 = 86732064$ which appears to be a much smaller number in decimal notation, but still takes 27 bits when written binary[4]. However, for performance reasons it is highly desirable to operate with native data types like Java's 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. Even if `BigInteger` was less expensive in terms of memory and CPU consumption its arbitrary size for representable values would

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

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

[3]110 0001 0110 0010 0110 0011 0000 0000

[4]101 0010 1011 0110 1101 0010 0000

not be a solution, since still a maximum string length needs to be known in advance for initializing the used OPE schemes properly, which is a problem we cannot get rid of.

In order to still deliver a good compromise between string length flexibility and performance, this thesis uses the following approach. To keep a certain security level the ciphertext space should be at least twice as big in terms of bit length compared to the plaintext space (see Section 4.3.1-V), so that a 32 bit Integer should be the preferred input size for the OPE schemes, while a 64 bit Long serves as output. 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 with random bytes, 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 predefined 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 sigificant” 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 [61] or using “0000”-chunks would leak the plaintext string length.

Using this approach means the “least significant” bytes of strings longer than 32 bytes, (thus, starting from the 33rd character), are being ignored in this process, which has three implications:

- The accuracy of the order of the encrypted values involves only the first 32 bytes of the plaintext string. A set of strings with their first 32 characters being equal has a high probability of being not exactly order preservingly stored. However if absolutely necessary, the accuracy can be increased by sacrificing performance and simply taking more than eight 4-byte-chunks into account when encrypting. Alternatively one should try to select the used values in the row identifier column more carefully, since being able to store $96^{32} = 2,7 \cdot 10^{63}$ unique row identifier values should be sufficient in practice.
- If only the OPE-encrypted chain of chunks was used as input for the database, two strings with the first 32 byte being equal would produce the same encryption output and thus column input for the row identifier OPE column. Since values in row identifier columns must be unique (see Section 2.1.2) the row containing the second string as identifier would just overwrite the row containing the first string as identifier when inserting the data, which has to be strictly avoided. The user would lose data unnoticed. Thus, in order to still produce unique representations with

high probability a SHA256 hash of the entire plaintext string is appended to the byte array representation. In this way, only identical inputs produce identical outputs, which moves the problem of rows overwriting each other to the quality of the generation process of the dataset. Assuming a string consists of the characters $c_1...c_x$, the final OPE encoding then is

$$\text{OPE-Encode}(c_1..c_x) = \text{Enc}_{\text{OPE}}(c_1..c_4) \mid\mid ... \mid\mid \text{Enc}_{\text{OPE}}(c_{29}..c_{32}) \mid\mid \\ \text{SHA256}(c_1..c_x)$$

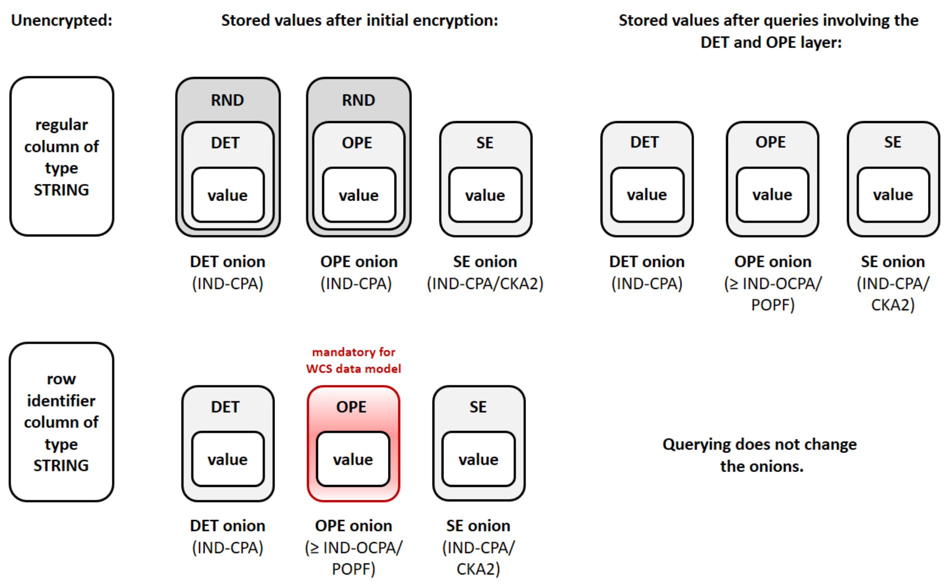
Another advantage is the constant output length of OPE-encoded strings, which is $8 \cdot 8 + 256/8 = 64 + 32 = 96$ bytes, independent from the length of the plaintext string, which means no information about the plaintext string length can be inferred.

- It is obvious, that decrypting byte arrays constructed in the above described fashion is time consuming and in case the plaintext was longer than 32 bytes even impossible. However, that is no problem, since for decryption the values of the DET onion layer can be used. Moreover, the DET onion can be used directly without the RND layer, because determinism is and has to be leaked in row key columns by the OPE onions anyway. Otherwise the database's behaviour regarding overwriting rows with the same identifier would change.

Independent from the data type of a row identifier column, there is a second requirement to consider for its OPE onion, that affects the choice of the OPE scheme to be used. Recalling the three available schemes (see Section 4.3.2, there is only one scheme, that can be used for row identifier columns. OACIS produces mutable ciphertexts, which would be a problem, since row identifiers are not allowed to change over time. The approach of mOPE involves a secret offset, that the database must not know about, but that has to be taken into account, when comparing values, in particular at data insertion time. Thus, only RSS is left, which works fine for the purpose of OPE-encrypting row identifier columns.

Mapping for atomic data types The concrete onion layout for the different data types can be seen in Figures 5.4, 5.5 and 5.6, where the red OPE onion for row identifier columns denotes the absolute minimum requirement necessary for maintaining the WCS data model, which would break, if a RND layer was wrapped around it.

String columns as shown in Figure 5.4 mark the most complex cases, mainly due to the costly transformation procedure necessary for realizing OPE column as described above. Apart from that all three onion types make sense for strings. The DET layer can be used for equality checks. The OPE layer is not only mandatory for the data model, it is also useful and desirable to be able to



Unencrypted:
Stored values after initial encryption:
Stored values after queries involving the DET and OPE layer:
regular column of type STRING
RND
DET
value
RND
OPE
value
SE
value
DET onion (IND-CPA)
OPE onion (IND-CPA)
SE onion (IND-CPA/CKA2)
DET
value
OPE
value
SE
value
DET onion (IND-CPA)
OPE onion (≥ IND-OCPA/ POPF)
SE onion (IND-CPA/ CKA2)
row identifier column of type STRING
mandatory for WCS data model
DET
value
OPE
value
SE
value
DET onion (IND-CPA)
OPE onion (≥ IND-OCPA/ POPF)
SE onion (IND-CPA/ CKA2)
Querying does not change the onions.

Figure 5.4.: Transformation of a plaintext string column into onion-layered ciphertext columns

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 in within strings.

Integer columns as shown in Figure 5.5 are a little easier to handle. Since they already contain numerical values, the costly conversion process is not needed. Furthermore the SE layer makes no sense for integers, as explained earlier. It cannot be searched within numbers and even if something like this is desired, string columns can (and should) be used. Having an integer column as row identifier column even makes the DET layer unnecessary, because since all used OPE schemes are deterministic, information regarding determinism can also be drawn from the mandatory OPE onion anyway.

Finally, byte blob columns (see Figure 5.6) are the most limited cases. Byte blobs, as the name suggests, usually represent raw binary data (e.g. images), which is not supposed to be searched, ordered by or compared to something. Thus, it is enough to have it deterministically encrypted in order to be able to recover it or perform equality checks. All other onions do not make practical sense. Especially the conversion to a numerical value for the OPE layer would be extremely expensive in terms of computation time, since compared to the process with strings much more than 32 bytes would have to be taken into account due to the usually larger sizes of byte blobs. That is why an OPE onion for row identifing byte blob columns is not efficiently computable and thus, a byte blob column is not allowed as row identifier column.

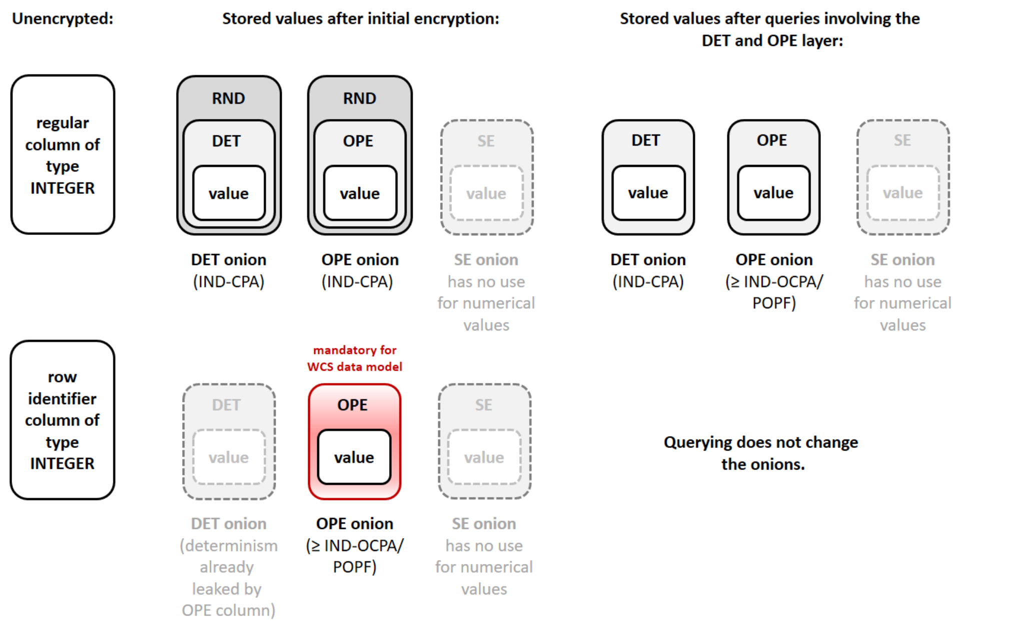

Unencrypted:
Stored values after initial encryption:
Stored values after queries involving the DET and OPE layer:
regular column of type INTEGER
RND
DET
value
RND
OPE
value
SE
value
DET
value
OPE
value
SE
value
DET onion (IND-CPA)
OPE onion (IND-CPA)
SE onion has no use for numerical values
DET onion (IND-CPA)
OPE onion (≥ IND-OCPA/ POPF)
SE onion has no use for numerical values
row identifier column of type INTEGER
mandatory for WCS data model
DET
value
OPE
value
SE
value
DET onion (determinism already leaked by OPE column)
OPE onion (≥ IND-OCPA/ POPF)
SE onion has no use for numerical values
Querying does not change the onions.

Figure 5.5.: Transformation of a plaintext integer column into onion-layered ciphertext columns

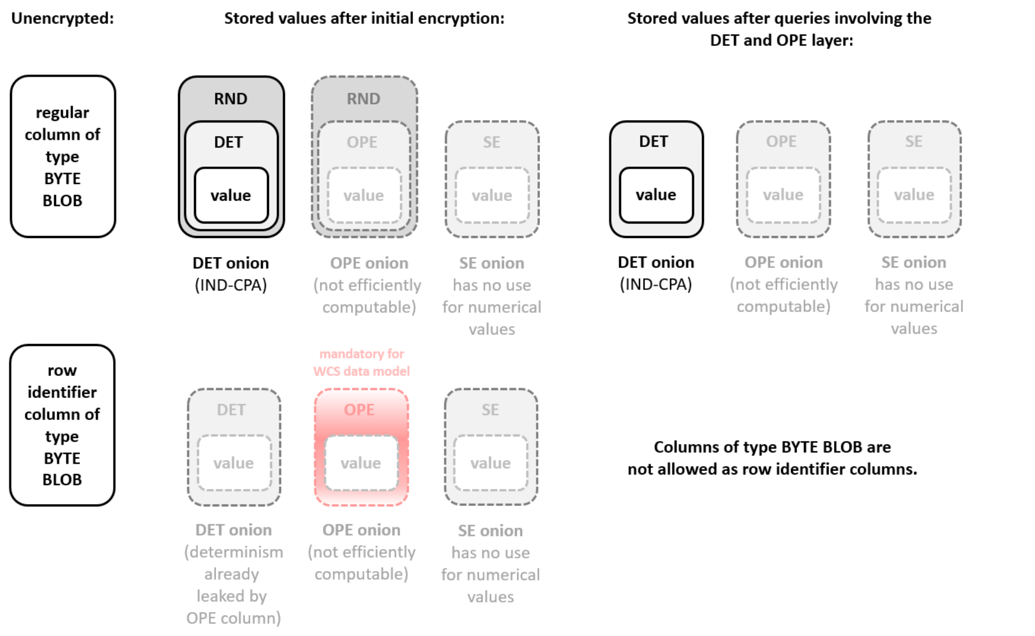
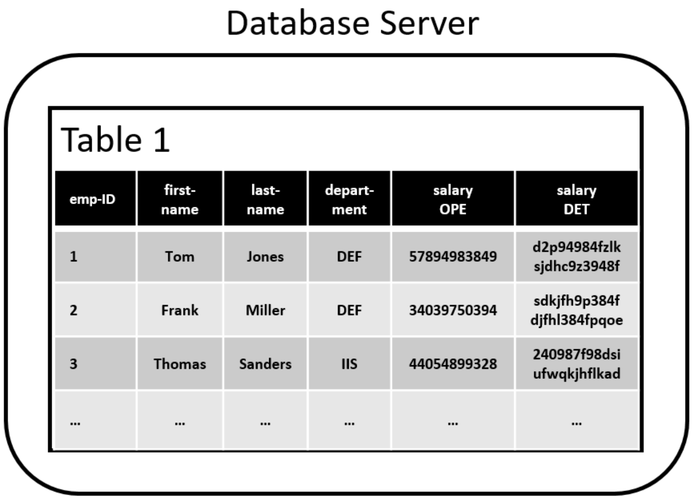

Unencrypted:
Stored values after initial encryption:
Stored values after queries involving the DET and OPE layer:
regular column of type BYTE BLOB
RND
DET
value
RND
OPE
value
SE
value
DET
value
OPE
value
SE
value
DET onion (IND-CPA)
OPE onion (not efficiently computable)
SE onion has no use for numerical values
DET onion (IND-CPA)
OPE onion (not efficiently computable)
SE onion has no use for numerical values
row identifier column of type BYTE BLOB
mandatory for WCS data model
DET
value
OPE
value
SE
value
DET onion (determinism already leaked by OPE column)
OPE onion (not efficiently computable)
SE onion has no use for numerical values
Columns of type BYTE BLOB are not allowed as row identifier columns.

Figure 5.6.: Transformation of a plaintext byte blob column into onion-layered ciphertext columns

5.2.2. Selective Encrypting

“Selective encryption” means to treat only pre-determined parts of a data set as sensitive and not to encrypt all other data. Depending on the concrete dataset it might not be necessary to encrypt every column of all tables in the database. Consider the table shown in Figure 5.7, that lists the salary of employees. Employee names and departments are no secret, but the salary information is supposed to be kept private. Thus, the salary column is encrypted according to the onion layer model like previously introduced in Section 5.2.1. Since the salary is numeric data, it results in an OPE and a DET column while all other columns remain readable without any decryption.


Database Server
Table 1

emp-ID	first-name	last-name	depart-ment	salary OPE	salary DET
1	Tom	Jones	DEF	57894983849	d2p94984fzlk sjdhc9z3948f
2	Frank	Miller	DEF	34039750394	sdkjfh9p384f djfhl384fpqoe
3	Thomas	Sanders	IIS	44054899328	240987f98dsi ufwqkjhflkad
...	...	...	...	...	...

Figure 5.7.: Human resources table of an imaginary company with secret salary information

Selective encryption helps to save computation time when reading from as well as writing to the database, because it reduces the number of required encryption and decryption operations. It also reduces storage space, because it avoids unnecessary indexes. Since in this example the employee names and department affiliations are no company secret, they do not have to be encrypted. That makes querying simpler and faster later on.

5.2.3. Separation of Duties

“Separation of Duties” describes the concept of more than one person being responsible and able to complete a task. The idea is to have an internal control mechanism, that prevents fraud and errors. The same principle can be used to take care of privacy issues that the onion layer model alone is not able to cover. Consider again the table shown in Figure 5.7. Imagine some query that involved ordering has already removed the RND layer of the salary column. The exact salary of the employees still does not leak, since it is encrypted in (most likely very large) different numerical values, but it can easily be seen, who earns the most or who earns more money than others. That might be unwanted. If more than one database instance is available (and the available databases are managed by individual and independent authorities), it makes sense to further split up the table in order to avoid that conclusions can be drawn. Therefore the principle of columnwise (vertical) partitioning (see Section 2.1.2) can be taken one step further by storing different columns on different database instances. One possible result for the example described above is shown in figure 5.8.

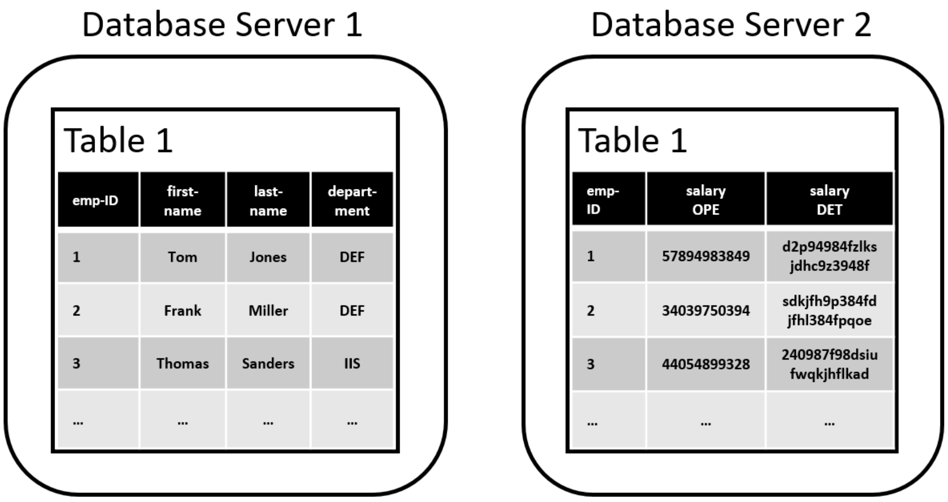


Table 1

emp-ID	first-name	last-name	depart-ment
1	Tom	Jones	DEF
2	Frank	Miller	DEF
3	Thomas	Sanders	IIS
...	...	...	...

Table 1

emp-ID	salary OPE	salary DET
1	57894983849	d2p94984fzlks jdhc9z3948f
2	34039750394	sdkjfh9p384fd jfhl384fpqoe
3	44054899328	240987f98dsiu fwqkjhflkad
...	...	...

Figure 5.8.: Human resources table of an imaginary company distributed across two independent databases

As can be seen, neither the administrator of database server 1, nor the administrator of database server 2 can infer any information about a specific salary or about who earns more than someone else, at least as long as they do not talk to each other. Hiding this information would not be possible by only using onion layered PPE. Of course at least the row identifier column (in the example the employee ID column) has to be present in all involved tables in order to link the information when queried later on. The API presented in this thesis allows splitting a table into an arbitrary number of database instances (see Section 5.6.1.4). Because of the data model unification as described in Section 5.2.5, even different database types can be arbitrarily mixed (e.g. use Cassandra as database server 1 and HBase as database server 2).

Thus, for the rest of this thesis the term *logical table* describes a table as shown in Figure 5.7. It contains all columns as desired when creating the table. In contrast, the term *physical table* refers to tables like shown in Figure 5.8.

One logical table can result in multiple physical tables, whereas a physical table always belongs to exactly one logical table. Physical tables are actually stored in databases, while logical tables represent their overall structure. If a logical table was not split up, it is equal to its physical table. As can be seen in Table 5.3, the client side metadata keeps track of all relevant data in order to map logical tables to physical tables and vice versa when querying.

This thesis proposes three strategies for splitting logical tables to physical tables:

- `Random distribution`: In this approach a logical table's columns are distributed randomly across the database instances available for the keyspace (see Section 5.6.1.2).
- `Round Robin distribution`: This approach distributes a logical table in round robin fashion across the available database instances. That means the data gets distributed as equally as possible at the point of creating a table.
- `Custom Distribution`: Using custom distribution the user actively specifies, which columns have to be stored separated from which other columns. In that way precise security enhancements can be achieved. Considering the example above the user could select the sensitive salary column to be stored separately from all other columns (this example is shown Section 5.6.1.4).

Which of the three options is used for a specific table has to be determined when creating that table (see Section 5.6.1.4). The set of available database instances is defined in the process of creating the keyspace, that is supposed to contain the table later on (see Section 5.6.1.2).

5.2.4. Table Profiles

As discussed in Chapter 4, especially the schemes for OPE and SE have their individual strengths and weaknesses. Depending on the application the user might want to exploit these strengths and minimize the impact of these weaknesses. That is why the API introduced in this thesis provides the option of specifying so called table profiles when creating tables (see Section 5.6.1.4). A table profile determines what PPE encryption schemes are actually used during data insertion. The selection of the PPE schemes it contains is based on the theoretical aspects described in Chapter 4, but also on practical performance evaluations introduced later in Sections 7.1.1. While for RND and DET layer encyption all profiles use AES, they differ regarding their OPE and SE schemes like follows:

- `OPTIMIZED READING`: This profile prioritizes schemes that have advantages for queries that involve mainly reading from the database. Thus it is the best choice for "write-once" databases. The OPE schemes best suited for fast reading are RSS and OACIS. They have the same type of index, which results in equal reading performance. However, RSS is the preferred choice for this profile, because is does not have the flaw of bad writing performance in case of a pre-sorted input (see Section 4.3.2.2). For the SE layer the SUISE scheme is used. It is faster than SWP in the process of searching, in particular for repeated queries.

- `OPTIMIZED WRITING`: This profile prioritizes schemes that have advantages for queries that involve mainly writing to the database. Thus, it should be used for scenarios, in which writing occurs more often than reading. 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. Since it does not have to maintain indexes, it can insert data faster than SUISE.
- `STORAGE-EFFICIENT`: This profile prioritizes storage needs over computation time and selects the PPE schemes, that require the least amount of storage for data and indexes, on client side as well as on server side. Thus, OPE layer columns are encrypted using mOPE, since that does not require an index at all and for the same reason SE layer columns are encrypted using SWP.

Table 5.1 gives the corresponding overview, to which there is only one exception, meaning the choice of the profile does not affect the choice of the PPE scheme: the OPE layer of row identifier columns is always encrypted using RSS for reasons explained in detail in Section 5.2.1.2. Note that SE layer encryption is only performed, if the plaintext data is a string. OPE layer encryption is only executed, if the plaintext data is a string or numerical value.

Table 5.1.: Overview about the table profiles and their corresponding PPE schemes

Profile	Data type	OPE scheme	SE scheme
Optimized Reading	String	RSS	SUISE
	Integer	RSS	-
	Byte	-	-
Optimized Writing	String	OACIS	SWP
	Integer	OACIS	-
	Byte	-	-
Storage-efficient	String	mOPE	SWP
	Integer	mOPE	-
	Byte	-	-

Independent from the used profile, every column gets its own instances of the PPE schemes they use. That means in particular, indexes are maintained per column, not per table. This allows the separation of duties as described in Section 5.2.3 and involves only the index data that is actually required for answering a query.

5.2.5. Unifying the Data Models of Cassandra and HBase

Cassandra as well as HBase follow the data model of WCSs as described in Section 2.1.2. However, 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 (see Section 6.3), but a few of them have to be dealt with in the individual process of designing a scenario's data model, if necessary. The following section provides insights into the key differences of the way both databases realize the WCS data model and helps understanding the design of the API later on.

5.2.5.1. How to Address the Row Identifier Column

The WCS data model dictates the existence of (at least) one column, that stores unique row identifiers per table: throughout this thesis that column is referenced to as row identifier column (see Section 2.1.2). Cassandra and HBase have different ways of addressing this column. Cassandra requires assigning a concrete name and data type for it in the process of creating a table. In contrast, HBase does not need any of that information, because its row identifying columns do not get a name and are always of type byte blob.

Thus, since Cassandra has to be be given the more precise definitions regarding the row identifier column when creating tables, it defines what input is necessary when creating tables in FamilyGuard. That is why defining a name and data type is mandatory for this process of creating tables using the API (see Section 5.6.1.4).

5.2.5.2. Composite Keys

Apache Cassandra has a concept of defining key values per row, similar to the way of doing so in the SQL world. It encompasses a variety of key types. First of all, there is the row identifier. It 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 also possible to combine multiple fields to create a row identifying key, which is then called a composite (or sometimes compound) 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, like for example in this table definition:

```
CREATE TABLE fancytable (
     part_key_1 text,
     part_key_2 int,
     cl_key_1 text,
     cl_key_2 int,
     cl_key_3 text,
     other_data text,
     PRIMARY KEY((part_key_1, part_key_2), cl_key_1, cl_key_2,
         cl_key_3)
  );
```

In contrast, HBase does not know the concept of multiple fields defining a row identifier. It always uses only one single-field row identifier per row. No other options are available. 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. Then, HBase's native Java API provides options for defining column prefix filters, that can be used to "simulate" compound keys, in a way that the prefix plays the role of the partition key:

```

// create HBase table object (HTable)
Configuration conf = HBaseConfiguration.create();
HTable table = new HTable(conf, "fancytable");

// manually create row key
String part_key_1 = "foo";
String part_key_2 = "bar";
byte[] rowkey = Bytes.add(Bytes.toBytes(part_key_1),
    Bytes.toBytes(Bytes.toBytes(part_key_1));

// put the data
Put p = new Put(rowKey);
p.add(Bytes.toBytes("column"), Bytes.toBytes("qualifier"),
    Bytes.toBytes("VALUE"));
table.put(p);

// get the data
Scan s = new Scan();
Filter filter = new PrefixFilter(Bytes.toBytes("foo"));
s.setFilter(filter);
ResultScanner rs = table.getScanner(s);

// do something with the result
// ...
```

There are frameworks available doing just that, for example Apache Spark[5] or Apache Phoenix[6], but since this thesis focuses on plain versions of the underlying databases, these frameworks are not considered here, which makes HBase the more restrictive database concerning the row identifier design. Thus, FamilyGuard only allows row identifiers consisting of a single field, which means if more information is supposed to be included in a row identifier, it has to be composed appropriately during the design of the dataset, for example by concatenating values in a certain fashion as described above.

5.2.5.3. 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. In this case, the value in

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

is not a single value, but a further subset of values. The single elements of these subsets can be addressed by traversal (set), specifying an index (list) or a key (map). In contrast, HBase does not support collection types, but has another mechanism for addressing elements of a row. The mapping to a concrete value in HBase looks like this:

$$(keyspace^{7}, table, row\ identifier, column, column\\ qualifier, timestamp) \rightarrow value$$

Here, the additional column qualifier can be used as an indexing mechanism within a “single” row element and thus, realize collection types. While using the API proposed in this thesis, 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 hidden. 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 FamilyGuard.

5.2.5.4. Data types

As already briefly mentioned in Section 2.1.2 Cassandra differentiates between a variety of data types[8]), whereas HBase stores everything as byte array. This

[5]`http://spark.apache.org/`

[6]`https://phoenix.apache.org/index.html`

[7]To be exact, keyspaces in HBase are called “namespaces”, but they are conceptually the same: the highest organizational level for organizing tables.

[8]for a complete list, see: `https://docs.datastax.com/en/cql/3.1/cql/cql_reference/cql_data_ types_c.html`

thesis will follow the HBase approach and store only byte arrays in encrypted columns in Cassandra as well. This has two advantages:

- 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.
- Seeing only byte blobs in the database makes it much harder for an attacker to infer information. Since column and table names are randomly generated as well, an adversary can only count the number of columns, but even that does not necessarily lead to useful information, since different plaintext data types require a different amount of ciphertext columns (see Figures 5.4, 5.5 and 5.6 and Section 7.3).

However, columns that are not supposed to be encrypted and are specified as such in the table definition (see Sections 5.6.1.4) will use appropriate data types in Cassandra.

5.3. Required Metadata Structures

Operating the DBMS with encrypted content requires keeping track of additional metadata structures, in particular to identify columns representing the onion layers and initiate PPE schemes with the right parameters. The following paragraphs and Table 5.2 - 5.4 provide an overview what metadata is required to manage keyspaces, tables and columns.

Keyspace Level

The keyspace is an object that groups all tables of a certain concept or design. It is usually the outermost container of the data in a DBMS. In some cases it is also called namespace (e.g. in HBase). Table 5.2 lists the metadata items FamilyGuard maintains for every plaintext keyspace in order to enable operations on encrypted data.

Table Level

A table groups columns that belong to the same concept or design and are likely to be queried together. Thus, a table has a similar organizational function for columns like a keyspace has for tables. As mentioned in Section 2.1.2, literature uses the term “column family” synonymously. For every physical table FamilyGuard maintains the metadata like described in Table 5.3.

Table 5.2.: List of metadata items maintained by FamilyGuard for every keyspace

plaintext name	The name of the keyspace in plaintext representation, given by the user in the process of creating the keyspace. The database never sees it.
ciphertext name	The name of the keyspace in ciphertext representation. This is a randomly generated string with a length of 8 printable characters that is used to identify the keyspace 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.
a list of available database connections	A list containing all the available database connections this keyspace is allowed to use for storing tables. Every list entry consists of the type of the database (e.g. Apache Cassandra) and the IP address that can be used to access the database. One logical table might be split and distributed in form of multiple physical tables using these connections (see Section 5.2.3).
a list of physical tables	A list of all physical tables present in this keyspace. Every table in this list is stored with the metadata as described in Table 5.3.
keystore	The keystore (see Section 5.4) that is used by PPE schemes to store keys in order to perform encryption and decryption tasks for columns of tables of the keyspace.

Column Level

A column stores key-value pairs, which are the most basic unit of information that can be stored in a WCS database (see Section 2.1.2). For every column the metadata as listed in Table 5.4 is maintained in order to enable operations over encrypted data.

5.4. Key Management

Using the framework of FamilyGuard means dealing with a number of cryptographic keys, caused by the need of multiple encryption schemes in the different encryption layers. The number of required keys per table can be calculated as follows:

- For the encryption of the RND layer it is necessary, that the same values in different columns of the same row result in different ciphertexts. Otherwise determinism would leak (at least within the row). Since the initialization

Table 5.3.: List of metadata items maintained by FamilyGuard for every physical table

plaintext name	The name of the table in plaintext representation, given by the user in the process of creating the (logical) table. Like the plaintext name of the keyspace, the database never sees it.
ciphertext name	The name of the table in ciphertext representation. Like the keyspace ciphertext name this is a randomly generated string with a length of 8 printable characters, used to identify the table 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.
initialization vector for deterministic encryption	This is the initialization vector used by the onion layer responsible for deterministic encryption. It is a byte array of length 16, randomly generated in the process of creating the table. It has to be the same for all values stored inside the table, otherwise equality checks over multiple columns would produce wrong results. Thus it has to be stored only once per table.
key for deterministic encryption	For the key the same is true as stated for the initialization vector, except for the length, which is 32 bytes. For comparable results over multiple values the same key has to be used for deterministic encryption throughout the entire table.
a list of columns that belong to this table	This is a list of all columns that are associated to the table. For every column metadata is stored as described in Table 5.4.
table profile	The table profile specifies in which way the columns of the table shall be encrypted (for details, see Section 5.2.4). For the sake of of API simplicity the same profile is used for all columns. However, there is no reason to not specify the profile for every column separately in order to achieve a better performance for each individual dataset. This can be considered an opportunity for future work (for details, see Section 8.2).
keyspace	The keyspace the table belongs to.
database connection	This specifies the database type and IP address of the database storing this table.
row identifier column	This is the column that contains the row identifiers for this particular table. According to the WCS data model (see Section 2.1.2) every table has exactly one.

IV column	This is the column that contains the initialization vectors for RND layer encryption. Each row needs an individual IV, otherwise equality could leak from the RND layer, which has to be avoided (see Section 5.2.1 for details).

vector for the used RND encryption scheme is unique only per row, the encryption key has to be different for every column in order to produce different ciphertexts for the same plaintexts in different columns. Thus, the RND layer needs one cryptographic key for every column.

- The encryption of the DET layer is required to produce the same ciphertexts for the same plaintext. Since joins are not possible in WCS databases that has to be true only for all ciphertexts per table (not the whole database). Thus, the DET layer requires one cryptographic key for every table.
- OPE encryption does not require any keys. The cryptographic secret is the index itself.
- The number of cryptographic keys required in the SE layer heavily depends on the used SE scheme. Like in the RND layer, different keys are needed for every column in order to not leak determinism. As discussed in Section 4.4.2.1 the SWP scheme needs two keys per column, one for its pre-encryption step and one for a keyed hashing computation. The SUISE scheme as described in Section 4.4.2.2 also needs two keys, one for generating search tokens and one for directly encrypting the plaintext using some IND-CPA secure encryption algorithm (which is AES for the implementation of this thesis).

As FamilyGuard can be extended to the use of other PPE schemes (see Section 6.4.2), the number of necessary cryptographic keys might grow further. However, since the database server is not allowed to possess these keys, they have to be managed and stored on the client side. Doing that manually is impractical for the user, which is why FamilyGuard 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 on single password to gain access to all keys, which massively improves the usability. In FamilyGuard this happens when the metadata is loaded (see Section 5.3 and 5.6.1.1).

Since all keys have to be available independently from each other, no key derivation algorithm is used when generating them. Instead they are created purely randomly using the random number generator of `java.security.Secure`

Table 5.4.: List of metadata items maintained by FamilyGuard for every column

plaintext name	The name of the column in plaintext representation, given by the user in the process of creating the table or (in case a column was added later to an existing table) the column itself. The database never sees it.
ciphertext name of the DET column	The onion layer model requires multiple ciphertext columns for each individual plaintext column in order to store the values resulting from the different PPE schemes. This field contains the name of the column storing deterministically encrypted values. It is a randomly generated string of 32 printable characters. Thus, it does not leak any information about the plaintext name.
ciphertext name of the OPE column	Similar to the ciphertext name of the DET column, this is the name of the corresponding ciphertext OPE column.
ciphertext name of the SE column	Similar to the ciphertext names of the DET and OPE columns, this is the name of the corresponding ciphertext SE column.
ciphertext name of the RND column	Similar to the ciphertext names of the DET, OPE and SE columns, this is the name of the corresponding ciphertext column for the RND layer. Note that this column is only required when storing sets for reasons as discussed in Chapter 5.2.1.
RND layer of DET column	This is a boolean value indicating whether the RND layer of the DET column was already stripped off in the process of performing operations over the encrypted data (which was the case if e.g. an equality check was computed).
RND layer of OPE column	This is a boolean value indicating whether the RND layer of the OPE column was already stripped off in the process of performing operations over the encrypted data (which was the case if e.g. comparisons or sorting were computed)
rowkey	This is a boolean value indicating whether the column contains rowkeys of a table or not.
column key	The column key is needed to perform the encryption of the RND layer. It has to be different for every column, otherwise information could leak about plaintexts of the same row (since the same IV and key would be used).

column type	The column type stores information about the data type that the column is housing. Supported types are strings, numbers and byte blobs. Supporting more types (like e.g. timestamps) can be considered subject of future work (see Chapter 8.2)
encrypted	Not all columns have to be encrypted and stored in onion layer fashion. If the column contains no sensitive data, this parameter is set to *false* and the content of this column is stored in plaintext (see Chapter 5.2.2)

`Random`[9].

5.5. Interacting with the Databases

Using encryption requires additional efforts, when reading from and writing to the databases. This section describes the necessary steps in detail, in which the concepts of the PPE schemes itself (Chapter 4 and Section 5.2.4) as well as of the onion layer model (Section 5.2.1), data distribution (Section 5.2.3), key management (Section 5.4) and metadata management (Section 5.3) come together.

5.5.1. Writing

5.5.1.1. Creating Keyspaces and Tables

Creating a keyspace using FamilyGuard requires the following extra steps compared to using the databases regularly:

- For hiding the plaintext name of the keyspace later on a new name for it is created, consisting of 8 randomly chosen characters (referred to as "ciphertext name" in Table 5.2).
- A metadata object representing the keyspace is created, containing the plaintext ciphertext mapping and all other relevant metadata as described in Section 5.3.
- One or multiple (see Section 5.2.3) queries are built and executed to actually create the keyspace on server side.

The process of creating a table is basically quite similar, but requires more attention when creating the individual columns:

[9]see `https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html`

- For hiding the plaintext name of the table later on a new name for it is created, consisting of 8 randomly chosen characters (referred to as "ciphertext name" in Table 5.3) like previously done for the keyspace. The same is also 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 (see Figures 5.4 - 5.6). Thus, multiple ciphertext names per column might be necessary.
- Metadata objects representing all future columns of the table and the table itself are created, as described in Section 5.3, in particular Table 5.3 and 5.4.
- Column keys for RND layer encryption as well as a table key and IV for DET layer encryption are created and associated to the table's metadata object.
- According to the distribution profile that was chosen for this table (see Section 5.2.3), it is decided how the individual columns are spread across the available database instances. 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 (see Section 5.5.2).
- One or multiple (see Section 5.2.3) queries are built and executed to actually create the (physical) table on server side.

5.5.1.2. Inserting Rows in Tables

Creating keyspaces and tables did not involve using the PPE schemes yet. That changes, if actual data is written to the database, that is supposed to be queried later on. Inserting data requires two types of information (see Section 5.6.1.6): what is supposed to be inserted (the actual row data) and where is it supposed to be inserted (keyspace, table and column names).

- 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 involved databases (type and IP address) that are responsible for storing the table. Furthermore if not done yet, the keystore associated to the keyspace is loaded.
- An IV for the RND layer encryption of the row is created, that will be stored in the row along with its other data items.
- 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 5.2.1). Thus, one plaintext data item may result in multiple ciphertext data items (each of which maybe the result of multiple encryptions).

- 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 this pre-processing steps is investigated further in Section 7.1.2.

5.5.2. Querying

On the one hand, querying encrypted data requires the database's ability to deal with the property-preserving encrypted values. This is no problem, except for the SE layer. On the other hand, it requires decrypting, before the data can be used by the application that asked for it. Both aspects are discussed in this chapter.

5.5.2.1. The Problem of Performing SE on Server Side

In order to understand why processing SE requests cannot be performed on server side like OPE or DET encryption does, it is necessary to take a closer look on how the database deals with PPE.

The DET layer is mainly responsible for equality checks. Performing such a check is a native capability of the database. No matter what format the data item has, the database can determine whether it is equal to another reference object or not. For example the database is able to check the string "BMW" for equality with every other string without being modified in any way. In particular, it can also natively handle encrypted representations of these strings (or numbers/byte blobs). The same holds true for the OPE layer, that is responsible to check for order relations. The database is able to compare data items to one another independent from their format. It can tell whether $2 < 4$, $A < Z$, etc. or not without being modified therefore in any way.

Unfortunately that does not hold true for the SE layer. As pointed out in Section 2.4.2.1, SE always relies on trapdoors that are passed to the database server, which are used to check for matches. This checking procedure requires computations, that the unmodified databases investigated for this thesis are not able to perform. Table 4.3 already gave an overview about the types of necessary computations for a variety of existing SE schemes. To recall the details for the two implemented schemes:

- For the SWP scheme (Section 4.4.2.1) to check for a certain word W, it has to be checked for every ciphertext word C whether $W \oplus C$ is of the form $S||F_k(S)$ for some S, where S is a pseudorandomly generated value and F is a pseudorandom function, that has to process an input value S and a key k. That includes two things neither Cassandra nor HBase is capable of doing being unmodified: performing bitwise XOR operations and computing pseudorandom functions (e.g. keyed hashing) on data items.

- For the SUISE scheme (Section 4.4.2.2) the concept for checking for matches is quite similar. Every encrypted representation of a word c in the index γ_f is split into $l||r$ and then checked if $H_{r_w}(r) = l$, where H is (again) a pseudorandom function and r_w the search token provided by the client (used as key for H). Again, that includes two things neither Cassandra nor HBase is capable of doing: splitting strings or byte arrays and perform pseudorandom functions on the resulting fragments.

That means SWP and SUISE cannot be implemented with the functionality to check for matches on server side. However, even though for practical reasons the scenario of this thesis aims for unmodified databases, this raises the question, whether Cassandra or HBase could be modified in some way to enable the operations described above, if a database modification was allowed (not including the obvious option to directly rewrite parts of their source code). Currently this question can be answered as follows:

- In Cassanda the only mechanism for executing custom code are so called user defined functions (UDFs), that can be specified in a CQL query. While this would allow bitwise XOR operations and splitting data items, it does not enable computing pseudorandom functions, because there is no mechanism to import the necessary cryptographic libraries. The reason for that is the template all custom code gets inserted into[10].

- The equivalent to UDFs in Cassandra are so called co-processors in HBase. Using them means implementing some given interfaces[11] or extending given classes[12] and making them available to the HRegionServers (see Section 2.1.2.2) by distributing all custom code in form of a single jar file. This file may include cryptographic libraries as well, which would allow HBase (in contrast to Cassandra) the execution of SE. However, a major disadvantage of this method is the fact, that the distribution of the jar file requires direct write access to the HDFS HBase is running on. The vast majority of today's IaaS/PaaS providers (see Section 2.2) does not offer an option for doing so. Instead, for the sake of simplicity everything is managed via a web interface, that does not let the user set up the database cluster in the required detail.

For the scenario of this thesis, which forbids modifying the databases, that means a performance drawback that we investigate further in Section 7.1.2. The client has to perform computations that otherwise could be done by a database

[10] which allows only access to `java.nio.ByteBuffer`, `java.util.*`, `org.apache.cassandra.cql3.functions.JavaUDF` and `com.datastax.driver.core.TypeCodec`. For more details, see: `http://batey.info/cassandra-udfs.html`

[11] `org.apache.hadoop.hbase.coprocessor` and `org.apache.hadoop.hbase.coprocessorService`

[12] `org.apache.hadoop.hbase.coprocessor.BaseRegionObserver`

(cluster). However, one has to keep in mind, that the SE-powered functionality to query for single words within a value field is a functionality, that the database otherwise either would not have at all (in case of Cassandra) or could only be realized with filters for substrings or regular expressions (in case of HBase).

In the implementation of this thesis, query conditions requiring processing SE are always evaluated at last, after as much rows as possible are filtered out through other query conditions involving the DET and/or OPE layer. This way as few as possible SE is involved. The following section explains this issue in the overall query processing.

5.5.2.2. The Query Process

From the client application's query against the encrypted data until the decrypted results are available for it to work with, the following steps are necessary:

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. In order 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. Therefore 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 read from the database(s). This set consists of two subsets, that might overlap. The first subset consists of all columns that are involved in query conditions, as identified in the first step. Depending on the type of condition, the appropriate ciphertext column is selected (e.g. the column representing the OPE onion, if the condition it is involved in is an order comparison). The second subset consists of all columns, that were selected by the user (and thus, are not necessarily involved in any query conditions). In this case the DET onion representing column gets chosen, because it is the fastest one to be decrypted later on.

4. Furthermore, there are two columns, that are always read from the database, independent from 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 (maybe 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 names are replaced by their ciphertexts counterparts, as well as concrete terms in conditions are replaced by their PPE encrypted equivalents. In the end no query contains any plaintext information anymore and can be executed. For reasons discussed in Section 5.5.2.1 all SE involving conditions are left out for the moment.

6. After all database instances have sent back their results to the client, the remaining rows are checked for whether they fulfill conditions involving SE (see Chapter 5.5.2.1), which leads to the final result set.

7. 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.

8. As a last step the final result set is decrypted using the DET onion representing column, as discussed in Step 3.

5.6. The API

5.6.1. API Methods for Database Interactions

This section introduces the methods provided by the API developed for this thesis. To allow a faster understanding, every method is introduced in the following way. Firstly, a CQL statement demonstrates a certain functionality. Since CQL is very similar to the well known SQL, a CQL query's intention is easy to grasp[13]. Secondly, the general usage of the API method is explained, that can be used to achieve the same functionality. Finally an API example is given, that corresponds to the introducing CQL statement.

The API provides methods for:

- Initializing itself by loading keyspace metadata from an XML file
- Creating and deleting keyspaces
- Creating and deleting tables
- Writing PPE-encrypted or unencrypted data to these tables, either by writing whole rows or updating only single values
- Reading/querying tables

As the implementation has a proof-of-concept character, there is of course room for more functionality, which can be considered options for future work. Details can be found in Section 8.2.

[13]In contrast, one has to become familiar with the native HBase Java API first to understand its working principles. Therefore, no examples in form of HBase API calls are shown at this point.

5.6.1.1. Initializing/Closing

In order to be able to interact with the databases the API has to be aware of the existing database instances as well as of the data inside them. Thus, it has to load the available keyspace metadata first and initialize the necessary database connections. All of that is taken care of by the API's constructor.

```
public API(String path, String password, boolean silent);
```

- `path`: The file path of the XML metadata file.
- `password`: The password that is needed in order to access the keystores managed by this API instance.
- `silent`: If set to `false`, status output and error messages will be printed to the console. While this helps to see what is going on, it can be a performance bottleneck. If set to `true`, no console output will occur.

Assuming the XML metadata file is located in /home/user/mydb.xml and no console output is wanted, the corresponding constructor call would be:

```
API api = new API("/home/user/mydb.xml", "mypassword", true);
```

Analogous to the initialization, a closing process is required to save the current keyspace metadata state back to the XML file and save client side indexes for future use. All is done by a close method.

```
public void close();
```

Thus, after all database interactions are done, the necessary corresponding call is

```
api.close();
```

5.6.1.2. Creating Keyspaces/Namespaces

Keyspaces[14] are the highest level of data organisation within the databases. At least one keyspace has to be created in order to house tables. Hence, that is also the first thing that has to be done using the API introduced in this thesis. Consider the following CQL statement:

[14]Whenever keyspaces are mentioned in this chapter, everything that is referred to is also true for HBase's namespaces.

```
CREATE KEYSPACE ksn
WITH REPLICATION = {
   'class' : 'SimpleStrategy',
   'replication_factor' : 1
};
```

As can be seen, it creates a keyspace called "ksn" (short for keyspace name) with the keyspace to be created having certain parameters. When using FamilyGuard, not only the keyspace name is important, but also what database instances are available to store the tables of the keyspace in the future. One database is mandatory, but arbitrarily more are possible (see Section 5.2.3). Thus, the API method for creating keyspaces looks like follows:

```
public void addKeyspace(String keyspaceName, String[] dbs,
    HashMap<String, String> params, String password);
```

- `keyspaceName`: the plaintext name of the new keyspace to be created. The API will replace that name with a randomly generated string in every interaction with the database. Thus it will not leak at any point in time.
- `dbs`: The database instances available for storing tables of this keyspace. Every string in this array has to be of the form `"<DatabaseType>-><IPAddress>"`, e.g. `Cassandra->192.168.2.101`.
- `params`: Additional parameters that specify, how the new keyspace is handled locally by the database instances. Supported parameters are `replication_class` and `replication_factor`. If not specified FamilyGuard will use the defaults `replication_class = SimpleStrategy` and `replication_factor = 1`[15].
- `password`: The password that is needed to access the JCEKS keystore which is used to manage all cryptographic keys required by the PPE schemes that are applied to tables and columns in this keyspace (see also Section 5.4).

The following example shows, how the keyspace "ksn" from the example above could be created, assuming there is an instance of Cassandra and an instance of HBase available for storing table data of this keyspace later on.

[15]Note that this parameters only have an impact in Cassandra (see `https://docs.datastax.com/en/cql/3.1/cql/cql_reference/create_keyspace_r.html`). HBase does not allow specifying such parameters for certain namespaces individually.

```
api.addKeyspace("ksn",
           new String[]{"Cassandra->192.168.0.1" ,
               "HBase->192.168.0.2"},
           new HashMap<String, String>() {{
              put("replication_class", "SimpleStrategy");
              put("replication_factor", "1");
           }}
           "mypassword");
```

5.6.1.3. Deleting Keyspaces

Assuming the keyspace created above should be deleted, the necessary CQL query would look like this:

```
DROP KEYSPACE IF EXISTS ksn;
```

The corresponding API method for deleting tables is:

```
public void dropKeyspace(String keyspaceName);
```

- `keyspaceName`: The name of the keyspace to be deleted. Note that all tables existing in this keyspace will be dropped as well.

To delete the keyspace created above, the following API call would be sufficient:

```
api.dropKeyspace("ksn");
```

5.6.1.4. Creating Tables

After a keyspace has been created, it can be filled with tables. Consider the following CQL query:

```
CREATE TABLE ksn.cars (
   id int PRIMARY KEY,
   model text
);
```

It creates a very simple table named “cars” within the previously mentioned keyspace “ksn”. It has two columns: an integer column “id”, which is also the primary key (thus, the row identifier) and a text column “model”. To do the same in FamilyGuard, the following method has to be used:

```
public int addTable(String keyspace, String tablename, TableProfile
    profile, DistributionProfile distribution, String[] columns);
```

- `keyspace`: The keyspace in which the new table is to be created.
- `tablename`: The plaintext name of the new table. Similar to the keyspace name, it will be replaced by a randomly generated string during every interaction with the database. Thus, it will not leak either.
- `profile`: The profile that determines which PPE schemes are used for encrypting the content of the columns in the new table. The available profiles are `Table Profile.FAST`, `TableProfile.ALLROUND` and `TableProfile.STORAGEEFFICIENT`. For details see Section 5.2.4.
- `distribution`: The algorithm, that determines how the table's columns are distributed across the available database instances. Note that the selection of an algorithm only makes any difference, if more than one database instance was provided during the process of creating the table's keyspace. Otherwise no distribution is possible, since only one database is available. The possible options are `DistributionProfile.RANDOM`, `DistributionProfile.ROUNDROBIN` and `DistributionProfile.CUSTOM`. For details see Section 5.2.3.
- `columns`: An array, that contains a string for every column of the new table. Each of those strings has to have the following format:

  `[<x>->][un]encrypted-><type>-><name>[->rowid]`

  - `x`: A numerical value in the interval [1 ... number of available database instances]. The column is stored in the xth available database. Specifying this value is only allowed (and makes sense), if `DistributionProfile.CUSTOM` was used and more than one database instance was specified when the table's keyspace was created. Assuming the distribution from the example shown in Figure 5.8 from Section 5.2.3 should be achieved, one can seperate the salary column from all the other columns by using 1 for the firstname, lastname and department column and 2 for the salary column (see the example below).
  - `[un]encrypted`: By specifying a column as `unencrypted` its contents will be stored without any encryption. In contrast, using the keyword `encrypted` enables the complete onion layer encryption with the PPE schemes used as described in Section 5.2.4.
  - `type`: The data type of the column's contents. The available options are "string", "string_set", "integer", "integer_set", "byte" and "byte_set" for text, numerical values and byte blobs and corresponding sets of these types.
  - `name`: The plaintext name of the column.

- `rowid`: The `rowid` attribute must be set to exactly one column of the table definition in order to specify the row identifier column. If no or more than one column is set to be the row identifier column, the creation of the table will fail. For details about why that is necessary see Section 2.1.2 and 5.2.5. Note that the row identifier column is always stored on every database instance. Otherwise there would be no way to link data stored on different database instances in a query process.

While this seems complicated at first glance, it is quite intuitive in practice, as the following two examples will show. In order to create a completely encrypted "cars" table as in the CQL example above, the necessary API call would be:

```
api.addTable("ksn", "cars", TableProfile.ALLROUND,
    DistributionProfile.RANDOM, new String[] {
   "encrypted->integer->id->rowid",
   "encrypted->string->model"
});
```

To illustrate the creation of a more advanced and actually distributed table, the API call for a table as shown in Figure 5.7/5.8 looks like:

```
api.addTable("company", "employees", TableProfile.ALLROUND,
    DistributionProfile.CUSTOM,
   new String[] {
      "1->unencrypted->integer->emp-ID->rowid",
      "1->unencrypted->string->firstname",
      "1->unencrypted->string->lastname",
      "1->unencrypted->string->department",
      "2->encrypted->integer->salary"
   }
);
```

As can be seen, the all columns containing information, that is not sensitive, are stored in database 1. Only the salary is stored encrypted in database 2. Note that the row identifier column `emp-ID` is stored in database 2 automatically as well. Row identifier columns always have to be present in all databases in order to be able to identify (parts of) rows across multiple databases (as described in Section 5.2.3).

5.6.1.5. Deleting Tables

Deleting tables is very similar to deleting keyspaces. Assuming the cars table created above is supposed to be dropped, that could be done in CQL issuing:

```
DROP TABLE IF EXISTS ksn.cars;
```

The corresponding API method is:

```
public void dropTable(String keyspaceName, String tableName);
```

- `keyspaceName`: The keyspace that contains the table to be deleted.
- `tableName`: The table to be deleted.

In order to drop the cars table from the example above, one can use:

```
api.dropTable("ksn", "cars");
```

5.6.1.6. Writing Data

Continuing the example of the cars table introduced above, a CQL statement supposed to fill this table with actual data could look like follows:

```
INSERT INTO ksn.cars (id, model)
VALUES (12, 'Audi');
```

It creates a new row inside the table with the id (and row identifier) 12 and the text "Audi" in the model column. The API method for inserting a row in FamilyGuard is:

```
public void insertRow(String keyspaceName, String tableName,
        HashMap<String, String> stringData, // "regular" values
        HashMap<String, Long> intData,
        HashMap<String, byte[]> byteData,
        HashMap<String, HashSet<String>> stringSetData, // collection
            types
        HashMap<String, HashSet<Long>> intSetData,
        HashMap<String, HashSet<ByteBuffer>> byteSetData)
```

- `keyspaceName`: The keyspace of the table, that the new row is written to.
- `tableName`: The name of the table, that the new row is written to.
- `stringData/intData/byteData`: Maps, that contain the actual values, that are written into the new row, one for each possible data type. The key of the map always contains the name of the column in which the new value is supposed to be written, whereas the value of the map entry

contains the actual value. Thus for example: if the numerical value 12 shall be written in the column named `id`, the map `intData` has to contain the key-value-pair $<"id", 12>$. The API then uses the available metadata to find out to which database instance and to what columns the new values have to be written. Note that if columns are encrypted to multiple onion layer columns (see Section 5.2.1), one key-value-pair can result in multiple columns.

- `stringSetData, intSetData, byteSetData`: The equivalent to stringData/intData/byteData for collection types as explained in Section 5.2.5.3. Instead of having one single element in a map's value field, sets of values can be inserted at once. Note that sets can only be inserted into columns, that where specified as collection type columns while creating the table previously.

Usually one will not have to insert values of every available type. If a type is not needed, one can use `null`, instead of passing an empty map, which is the normal case in practice. The example CQL-query from above can be translated into the following API call:

```
api.insertRow("ksn", "cars",
        new HashMap<String, String>(){{ //stringData
           put("model", "Audi");
        }},
        new HashMap<String, Long>(){{ //intData
           put("id", 12);
        }},
        null, //byteData
        null, //stringSetData
        null, //intSetData
        null //byteSetData
     );
```

5.6.1.7. Reading/Querying Data

The API methods introduced so far involved only writing or deleting data, which are operations, that do not return any interesting results. However, a fundamental purpose of databases is of course reading data and thus, getting back exactly specified information. This specification usually comes by executing queries like:

```
SELECT id, model
FROM ksn.cars
WHERE ps>100 AND model='BMW';
```

This example is supposed to return all cars with more than 100 PS that where manufactured by BMW. To achieve the same in FamilyGuard the API method `query` has to be used, which comes with the following signature:

```java
public DecryptedResults query(String[] columns, String keyspace, String
    table, String[] conditions)
```

In contrast to the previously discussed API methods it returns an instance of the class `DecryptedResults` instead of a primitive data type or void. This class is introduced in detail in Section 5.6.2. The remaining parameters are as follows:

- `columns`: The columns that are supposed to be part of the result set. In general, all columns that one would write into the `SELECT` clause of an CQL/SQL query should appear here. The column that contains a table's row identifier is always automatically added.
- `keyspace`: The plaintext name of keyspace of the table, that the query is executed against.
- `table`: The plaintext name of the table, that the query is executed against. To continue the analogy to CQL/SQL queries: keyspace and table names would appear in the `FROM` clause.
- `conditions`: A set of conditions that the resulting rows are supposed to meet. Each element of this set is a string representing a condition in the form

  `<columnname><operator><term>`

  - `columnname`: The plaintext name of the column that is involved in this condition.
  - `operator`: The operator used to define the condition. The following operators are allowed:

=	equal (makes use of the DET layer)
>	greater than (makes use of the OPE layer)
>=	greater than or equal (makes use of the OPE layer)
<	less than (makes use of the OPE layer)
<=	less than or equal (makes use of the OPE layer)
#	includes (makes use of the SE layer, only for text columns)

    Note the difference between = and # when it comes to text values. While = checks for equality of complete strings, # can be used to search for single words within these strings. Details on what is considered to be a word can be found in Section 4.4.2.1. For example working with the condition "model=BMW" would return only rows, where model exactly matches the string "BMW", whereas using the "#" operator would also return rows like "BMW 320d", "new great BMW car", etc. Thus the # operator works similar to SQL's `LIKE %term%` operator.

Note further, that the # operator is only available in encrypted text columns, since it realizes its functionality utilizing the capabilities of SE.

– `term`: The term used to define the condition.

Thus, the call for the above presented example looks as follows:

```
DecryptedResults results = api.query
   (new String[]{"id", "model", "ps"},   // SELECT
   "ksn", "cars",                        // FROM
   new String[]{"ps>100", "model=BMW"}); // WHERE, also # possible
                                         // for getting everything
                                         // that includes "BMW"
```

More examples for queries can be found in Appendix B.

5.6.2. API Methods for Decrypted Result Sets

When a query is issued using the provided API method as described in Section 5.6.1.7, an instance of the class `DecryptedResults` is returned. If the (logical) table, that the query was executed against, was spread across multiple databases as described in Section 5.2.3, it joins the individual column data from all of the corresponding (physical) tables (only from rows that fulfill the query's conditions, of course). It also provides access to columns that have been stored originally in unencrypted form.

Depending on the expected result type the user can access individual decrypted values by calling one of these methods:

- `public String getStringValue(byte[] id, String column);`
- `public int getIntValue(byte[] id, String column);`
- `public byte[] getByteValue(byte[] id, String column);`
- `public Set<String> getStringSetValue(byte[] id, String column);`
- `public Set<Integer> getIntSetValue(byte[] id, String column);`
- `public Set<byte[]> getByteSetValue(byte[] id, String column);`

In all of these methods the parameter `id` is the row identifier of the row that is about to be accessed, whereas `column` is the name of the column that is about to be accessed. Of course, only columns that were specified in the `columns`-parameter of the corresponding query call (see Section 5.6.1.7) are available here, because only those have been selected to be in the result set in the first place.

The `DecryptedResults` class also provides two more auxiliary methods for handling a query's result set:

- `public int getSize();`
- `public void print(int numberOfRowsToPrint);`

As their names suggest `getSize()` delivers the total numbers of rows of the result set and `print(int numberOfRowsToPrint)` prints the specified number of rows to the standard console.

5.6.3. API usage

This section describes the simple rules one has to follow when using the API provided by FamilyGuard.

- Before database interactions can be performed, the current metadata always has to be loaded upfront by calling the constructor method, described in Section 5.6.1.1. In order to keep the metadata and the database contents consistent, the `close` method has to be called after all database related tasks are finished. Doing so saves the current table metadata as well as client side indexes of the used PPE schemes, if required.

  ```
  API api = new API("pathtometadata", "mypassword", true);

  ... // interactions with the database(s)

  api.close();
  ```

- FamilyGuard can only manipulate tables, that have been created using it. Otherwise there would be no metadata available to describe the data structures necessary for the layered encryption, possible data distribution across multiple database instances, etc.
- In particular, writing as well as querying only works for keyspaces and tables, that have been created using the methods `addKeyspace` and `addTable`, described in Section 5.6.1.2 and 5.6.1.4.

No more rules need to be followed. The user is free to combine arbitrary interactions with the database as he would do without FamilyGuard.

6

Implementation

This chapter describes the implementation of the PPE schemes as well as of the API that uses them. It starts with an overview on some basic design aspects, followed by a closer look at the cryptographic primitives used for this work. Afterwards it presents the data flow from the client application to the database and back. Finally, the chapter is concluded by a description of how this work can be extended to support other databases and PPE schemes.

Contents

6.1. Overview

All implementations were entirely done in Java 8 in about 10.000 lines of code, including all PPEschemes, API methods and database communication. The complete source code is available online[1]. It was entirely written by the author of this thesis, except for the implementation of the OPE scheme of [14], as described in Section 4.3.2.3, which was implemented by Daniel Homann [2].

Package Organization

The source code is divided in five packages.

- The `crypto` package bundles implementations of all used PPE schemes and their indexes. Furthermore it contains classes for extending the functionality of the JCEKS and implementing the `javax.crypto.SecretKey` interface.
- The `databases` package contains classes for managing the keyspace, table and column metadata as described in Section 5.3 and communicating with the supported databases.
- The `enums` package bundles all enumeration types.
- The `interfaces` package bundles all interfaces.
- The `misc` package contains auxiliary functionality like writing serialized data structures to the file system, converting data types, etc.

Furthermore the project root contains the `API.java`, which bundles all API methods described in Section 5.6 end exposes them for external usage.

Indexes of PPE Schemes

Since disk access and memory management in WCSs are performed at column level, the indexes of all PPE schemes are designed the same way. That means every column that uses a PPE scheme relying on an index, gets its own index. Thus, 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.

[1]see `https://gitlab.gwdg.de/twaage1/FamilyGuard`

[2]Research Group for Knowledge Engineering, Institute of Computer Science, University of Goettingen

Interacting with the Databases

The Cassandra Java Driver 3.1[3] is used to interact with Apache Cassandra, which allows utilizing the current version 3 of the Cassandra Query Language (CQL). CQLv3 is the first version of CQL, that explicitly supports collections, as described in Section 5.2.5.3. 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[4]. All operations have to be performed creating put-, get-, or delete-objects first, equip them with appropriate row filters that correspond to query conditions (if desired) and hand them over in form of "scanners" to objects, that represent tables and keyspaces.

Client Side Storage

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 and thus, in a structured and easily accessible manner. For that purpose, every class, that represents a metadata object, implements the interface `interfaces.SaveableInXMLElement`.

6.2. Cryptographic Primitives

The Java Cryptography API is the foundation for using cryptographic algorithms in Java. It specifies interfaces, that can be used further by so called crypto providers. This thesis uses two of them:

- The Java Cryptography Extension (JCE) has been Java's built-in solution since JDK 1.4. It is a framework for various cryptographic tasks like encryption, authentication or key management.
- The "Legion of the Bouncy Castle" package (BC) is an open source alternative to the JCE, which is a little faster in many cases, but sometimes does not offer the same comprehensive functionality.

If the desired functionality can be realized with both crypto providers, this thesis always uses the one with the faster implementation. Table 6.1 gives an overview about what cryptographic primitive was used for which purpose.

[3]see `https://github.com/datastax/java-driver`
[4]`http://hbase.apache.org/apidocs/`

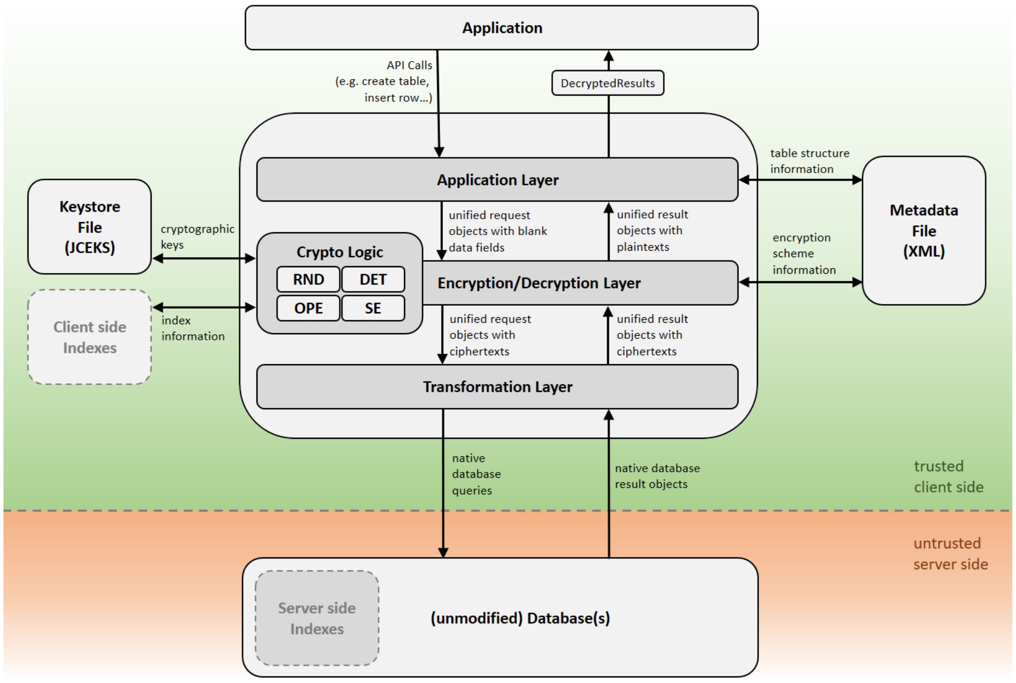
Table 6.1.: Cryptographic primitives used in this thesis

Purpose	Layer	Scheme	Crypto Provider
Encrypt plaintexts	RND	AES	BC (AES CBC)
Encrypt plaintexts	DET	AES	BC (AES CBC)
part of lazy-sampling range gaps	OPE	mOPE	JCE (HMAC-SHA256)
part of sampling the final range point to a domain value	OPE	mOPE	BC (AES CBC)
Randomly initialize the index, randomly split domain and range	OPE	RSS	JCE (SecureRandom, SHA1PRNG)
Pre-encrypt searchwords $E(W)$	SE	SWP	BC (AES CBC)
Compute the key for "controlled searching"	SE	SWP	JCE (HMAC-SHA1)
Padding plaintexts to length n	SE	SWP	BC (PKCS7 Padding)
Random numbergenerator G	SE	SWP	JCE (SecureRandom, SHA1PRNG)
Compute $F(S)$	SE	SWP	JCE (HMAC-MD5)
Encrypt plaintexts	SE	SUISE	BC (AES CBC)
Compute the search token r_w	SE	SUISE	JCE (HMAC-SHA1)
Random oracle H	SE	SUISE	JCE (HMAC-SHA1)
Random numbergenerator G	SE	SUISE	JCE (SecureRandom, SHA1PRNG)

As previously mentioned in Section 5.4, apart from the usage of cryptographic primitives in the implementation of the various PPE schemes, this thesis uses the keystore implementation of the JCE for managing and storing all necessary cryptographic keys.

6.3. Data Flow

This section discusses the data flow and involved components of FamilyGuard's architecture in detail. As can be seen in Figure 6.1 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.



Application
API Calls (e.g. create table, insert row...)
DecryptedResults
Application Layer
table structure information
Metadata File (XML)
Keystore File (JCEKS)
cryptographic keys
unified request objects with blank data fields
unified result objects with plaintexts
Crypto Logic
RND
DET
OPE
SE
Encryption/Decryption Layer
encryption scheme information
Client side Indexes
index information
unified request objects with ciphertexts
unified result objects with ciphertexts
Transformation Layer
native database queries
native database result objects
trusted client side
untrusted server side
Server side Indexes
(unmodified) Database(s)

Figure 6.1.: Architecture of FamilyGuard

6.3.1. 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 as described in Section 5.6.1. 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 "unified request objects" (UROs) for every interaction with the database. These UROs describe exactly, what is supposed to happen (using their `Type` filed, 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`)[5]. As the name suggests, `DBLoc` is a data structure, that precisely describes a location within the database, consisting of a keyspace/namespace, table name and row/column restricting patterns. For example, when creating keyspaces only the keyspace name

[5]To be precise: a URO has one more field, that stores a reference to the row identifier column, which is important for some query types. However, for the sake of simplicity of the explanations in this chapter, this is only mentioned when important in the following sections.

is of interest and all other values can be set to `null`, while querying the database is the more challenging case for DBLocation. Then keyspace/-namespace 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.

Consider for instance the examples from Section 5.6.1.6 and 5.6.1.7 (reading and writing data). The corresponding UROs can be seen in Figure 6.2. Fields not necessary for a specific type of opration are set to `null`. Note that plaintext values only appear in UROs, if a column was marked to be stored unencrypted (see Section 5.2.2). Otherwise the corresponding fields in the `String/Int/Byte_Args` section are empty when passed to the encryption layer and filled there.

TYPE: Read
DBLoc: ksn.cars.ps>100&model=BMW.[id,model]
ROW_KEY_COLUMN: id
STRING_ARGS: null
INT_ARGS: null
BYTE_ARGS: null
STRING_SET_ARGS: null
INT_SET_ARGS: null
BYTE_SET_ARGS: null

TYPE: Write
DBLoc: ksn.cars.null.null
ROW_KEY_COLUMN: id
STRING_ARGS: (model, \`Audi´)
INT_ARGS: (id, 12)
BYTE_ARGS: null
STRING_SET_ARGS: null
INT_SET_ARGS: null
BYTE_SET_ARGS: null

Figure 6.2.: URO examples for reading (left) and writing (right) data

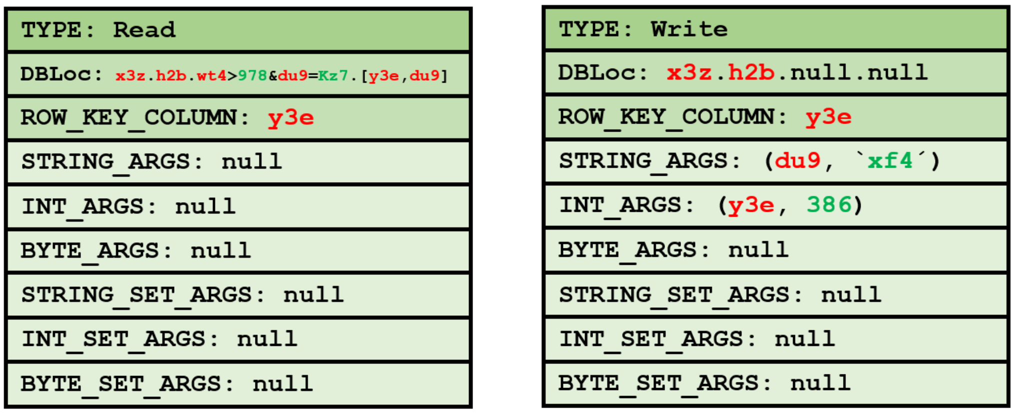
- It takes care of the client side metadata, which consists of two subtasks. Firstly, it always synchronizes the metadata. If new keyspaces, tables or only columns are created or deleted, the metadata has to be updated. This is important for knowing the location of actual data and for translating plaintext 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 collects and accumulates the results from the individual sources and creates an instance of `Decrypted Results`, which it then passes to the application in order to provide an easy and unified access to the result set of a query.

6.3.2. Encryption Layer

The UROs outputted 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 sub tasks:

- 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.



TYPE: Read
DBLoc: x3z.h2b.wt4>978&du9=Kz7.[y3e,du9]
ROW_KEY_COLUMN: y3e
STRING_ARGS: null
INT_ARGS: null
BYTE_ARGS: null
STRING_SET_ARGS: null
INT_SET_ARGS: null
BYTE_SET_ARGS: null

TYPE: Write
DBLoc: x3z.h2b.null.null
ROW_KEY_COLUMN: y3e
STRING_ARGS: (du9, `xf4´)
INT_ARGS: (y3e, 386)
BYTE_ARGS: null
STRING_SET_ARGS: null
INT_SET_ARGS: null
BYTE_SET_ARGS: null

Figure 6.3.: URO examples from Figure 6.2 after passing the encryption layer

The results can be seen in Figure 6.3. Ciphertexts generated from and maintained in the client side metadata are shown in red, whereas ciphertexts produced by PPE schemes for onion layers are shown in green. In the opposite case when data is returned from the database, the encryption layer decrypts these ciphertexts.

6.3.3. 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 database individual details. Thus, it is also the only part, that has to be changed, if there is the desire to support other databases in the future. Therefore only two Java classes have to be implemented (see Section 6.4.1).

6.4. Extensibility

6.4.1. Other databases

Supporting other WCSs can be achieved easily by extending only two classes of the `databases`-package: `DBClient.java` implements methods for interacting with the database (e.g. managing connections, inserting data and querying), while `Result.java` implements methods for accessing the databases' individual resultset objects. The corresponding implementations for Cassandra and HBase are about 400 lines of code each.

6.4.2. Other PPE schemes

Supporting other PPE schemes works similar. Depending on the layer in which the new scheme is supposed to operate, either the abstract class `RNDScheme`, `DETScheme`, `OPEScheme` or `SEScheme` of the `crypto` package has to be implemented. If the scheme requires an index, the `Index` class has to be extended as well. The only requirement for new PPE schemes is, that they have to be dynamic (as discussed in Section 4.3.1-IV-c for OPE and Section 4.4.1-II for SE). Otherwise the database could only be used in a "write-once" fashion.

7 Evaluation

This chapter evaluates the approach of this thesis as described in Chapter 4 and 5 focussing on three aspects. The first one is performance. Cryptography always comes with the cost of additional runtime. This performance loss is quantified in Section 7.1. The second aspect is functionality, because having to deal with ciphertexts limits the databases' native capabilities. Details can be found in in Section 7.2. Finally, the third aspect is security. The question of what information about the plaintext data can still be inferred from the ciphertext data is answered in Section 7.3.

Contents

7.1. Performance

This section evaluates the performance of running Cassandra and HBase using PPE encryption. It starts with focus on the pure PPE schemes to explore their individual strengths and weaknesses in terms of runtime (Section 7.1.1). However, using the database for real world queries requires more effort (keeping track of encryption metadata, encryption keys, managing the onion layers and indexes etc). Thus, the corresponding overhead is taken into account afterwards in Chapter 7.1.2 to get a more realistic picture.

7.1.1. PPE Schemes

All experiments in this section were run on an Intel Core i7-4600U CPU @ 2.10GHz, 8GB RAM, a Samsung PM851 256GB SSD using Ubuntu 16.04. Since these tests are supposed to evaluate the pure performance of the schemes and databases, the overhead caused by the onion layer model and this thesis' API (as introduced in Section 5.6) is avoided by using Cassandra's Java Driver and HBase's native Java API only. Furthermore, in order to avoid measuring network effects local installations of the databases were used, as only the computation time of the schemes in combination with the insertion speed of the databases was of interest. Cassandra was used in version 3.9, HBase was used in version 1.3.

7.1.1.1. Random and Deterministic Encryption

In the implementation of this thesis AES and Blowfish are used to realize the RND and DET layer encryption (see Section 4.2). Both are well known cryptographic algorithms with hardware support on many platforms, for which plenty of performance evaluations can be found in literature (for example [28, 92, 82]). Thus, there is no need to perform another analysis at this point.

7.1.1.2. Order-Preserving Encryption

Since the authors of the OPE schemes used in this thesis did not benchmark their work[1], the individual results as well as a comparison between them is of interest.

Encrypting Thus, for the measurement of this thesis up to 20000 uniformly distributed and randomly created 32-bit integer values were inserted into Cassandra and HBase using the three OPE schemes as described in Section 4.3.2.1

[1]Except for [54]. However, in their work only small numbers up to 16 bit length were encrypted and the insertion order was always random, which does not allow a comprehensive measurement, because it does not reflect the fact that insertion order matters.

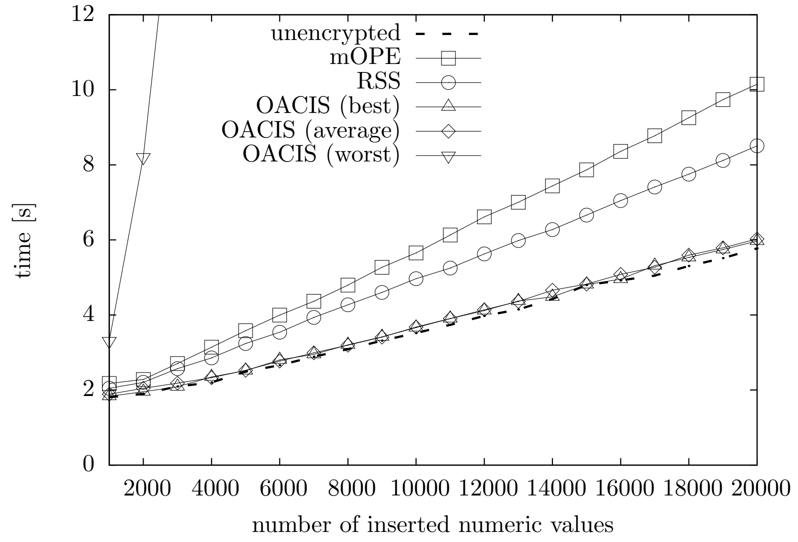


Figure 7.1.: Time needed for encryption with increasing data set size in Cassandra

- 4.3.2.3. A value is always encrypted and inserted, before the next one is encrypted and inserted. For the sake of a fair comparison between the databases no bulk loading features are used. While for mOPE [14] and RSS [100] the order of insertion does not matter, for OACIS [54] the three cases as described in Section 4.3.2.2 have been tested.

Figure 7.1 and 7.2 present the results, showing the average of ten measurements. Except for the worst case scenario of OACIS, which runs in $O(\frac{n^2}{logn})$, all other test cases require $O(n)$ with only slightly different constant factors. Even though they have an index to maintain the approaches of RSS and OACIS are generally faster than the stateless mOPE scheme. The only exception is using OACIS with pre-sorted input, which is prohibitively slow and should be avoided. The best combination of OPE scheme and WCS is OACIS in its best and average case with Cassandra, where the encryption step causes a performance loss of only 3%.

Cassandra is generally ca. 40% faster than HBase, mainly because RSS and OACIS are so fast that the database system's mere insertion time requires a significant share in the overall process of encrypting and inserting. With Cassandra being optimized for writes it takes advantage of this. An exception is the worst case usage of OACIS, because its re-balancing phase requires also reading performance. In this case HBase is always 12-15% faster than Cassandra, which seems to reflect the fact, that while Cassandra is optimized for writes, HBase is optimized for reads.

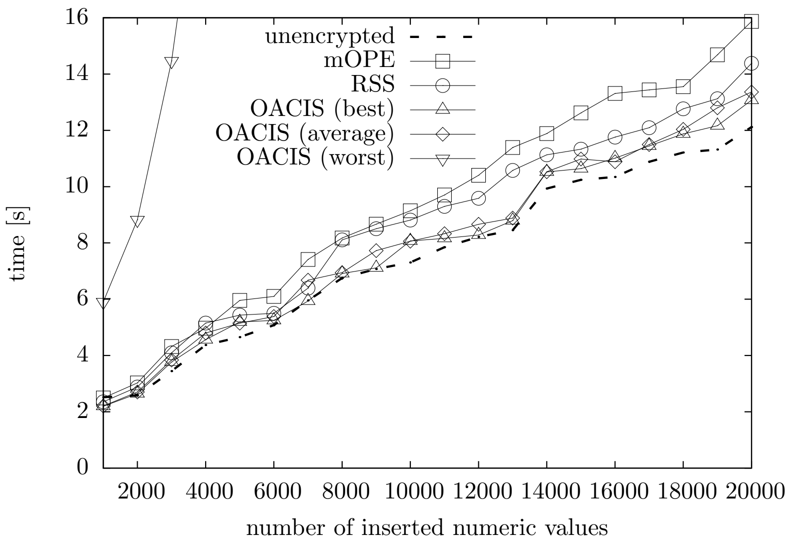


Figure 7.2.: Time needed for encryption with increasing data set size in HBase

Decrypting Because decrypting is very simple, there is no need for an elaboration in the same level of detail as for the encryption process. In RSS and OACIS it is just a trivial lookup in the index which takes less than 1 ms. In contrast, mOPE is computationally more expensive, since it has no index for lookups. Instead it has to compute the decryption. Hence it requires up to 4 ms for a single value.

Summary The results show that OPE can be used in WCSs efficiently. However, when choosing an OPE scheme it makes sense to think about the future use of the database. If insertion speed matters and there is a low probability of pre-sorted inputs, the scheme of [54] is advisable, which is used in the table profile `OPTIMIZED WRITING` (see Table 5.1). If an index should be avoided and ciphertexts are required to be immutable mOPE is the way to go (table profile `STORAGE-EFFICIENT`). RSS is a compromise between both. It delivers immutable ciphertexts almost as fast as OACIS. Since it has an index, that provides short lookup runtimes, it can be used by choosing the table profile `OPTIMIZED READING`.

Of course in practical applications a combination of the different OPE schemes is possible. There is only one use case in which RSS is without an alternative: row identifier columns are always (and only can be) encrypted with RSS, since OACIS is not qualified for that task due to its mutable ciphertexts and mOPE needs a secret offset that the database is not allowed to know about (see Section 5.2.4).

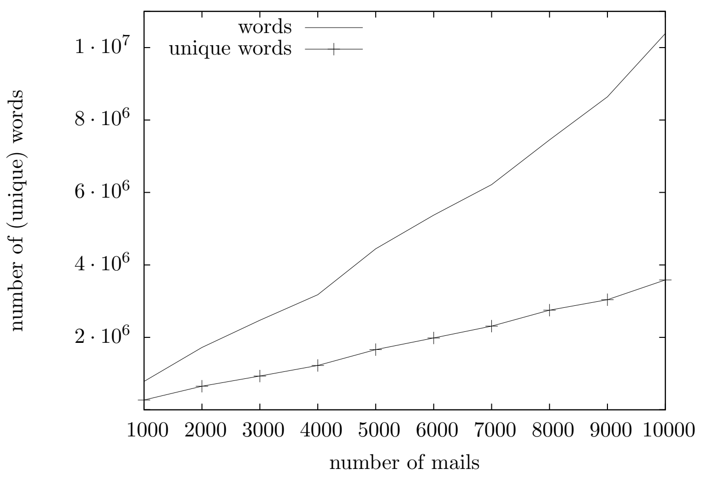


Figure 7.3.: Number of (unique) words with increasing size of the data set

7.1.1.3. Searchable Encryption

Like the previous section did for OPE, this section quantifies the performance of both of the SE schemes that have been implemented for this thesis: SWP (see Section 4.4.2.1) and SUISE (see Section 4.4.2.2). The hardware setup remains the same as for the OPE benchmarks.

The Dataset. When benchmarking OPE encrypting randomly chosen numerical values is sufficient. If a value shows up again, it can either be looked up in the index (RSS, OACIS) or encrypting it again takes exactly the same amount of resources as it took for encrypting it the first time. The situation for SE is different. Text length and word distributions have an impact on how long encryption and decryption takes. For example, the search time of SWP depends on where exactly in the text (if at all) the search word occurs. The speed of encryption and decryption in SUISE depends on the size of the index so far and whether the searchword was searched for before or not. Thus, instead of testing the SE schemes with just randomly chosen words, a real dataset is needed. This thesis uses a subset of the Enron email dataset[2], which is a set of about 500,000 e-mails of any size. It reflects the practical scenario of using SE for protecting sensitive mailbox data. For the benchmarks it is assumed that an average mailbox may have a size from 1,000 up to 10,000 emails. Hence, the measurements are started with 1,000 randomly chosen emails of the corpus. That number is increased up to 10,000 emails in order to see how the schemes and databases scale.

Figure 7.3 presents the key characteristics of the dataset. As can be seen

[2]available at `https://www.cs.cmu.edu/~./enron/`

the total number of words in the dataset increases from roughly 780,000 in 1000 mails to over 10 million words in 10,000 mails. In order to evaluate the measurements of the SUISE scheme, it is important to know how many unique words are contained in this total number of words, as that leads to the number of entries in its indexes. In the sense of the SUISE scheme a word counts as a "unique" word, if it appears once per text[3]. The same word counts again, if it appears in another text. Hence, the number of unique words is not necessarily the same as the number of different words (which is most likely much lower). Figure 1 shows that the amount of unique words compared to the total number of words is always around 30% as the dataset grows.

Encrypting The first test measures the time taken by the encryption processes of SWP and SUISE. This includes all steps necessary for the encryption itself as well as the time needed for outputting the results to the databases. In case of the SWP scheme the output only consist of the encrypted texts, whereas in case of the SUISE scheme it consists of the encrypted texts and the indexes γ_f and γ_w (see Section 4.4.2. The word length for SWP is set to $n = 8$ for reasons described further below (see Appendix A.1).

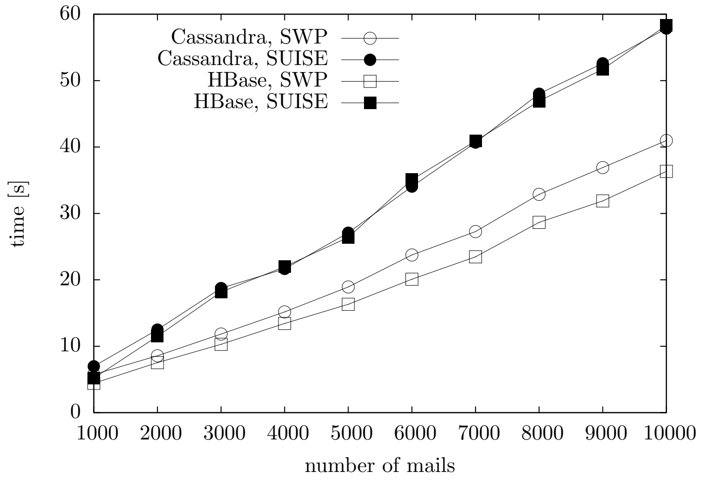


Figure 7.4.: Time needed for encrypting the data set with increasing size

Figure 7.4 illustrates the results. Since both algorithms have to iterate once over the complete plaintext input, it is no surprise that they require $O(n)$ with SUISE having a factor of 1.5 compared to SWP.

Therefore, the SWP scheme always finishes the encryption process faster than the SUISE scheme, no matter which underlying database is used. Even though SUISE only iterates over unique words and thus only needs around 30% of the

[3]In this context a text is the content of one row item ("table cell") within the database.

iterations compared to SWP (see Figure 7.3), this is not enough to compensate for the extra effort of maintaining its two indexes. Thus, SWP's overall encryption performance is better. Of course, this result depends on the given dataset. With a dataset having less than 30% unique words, SUISE might be faster than SWP. Concerning the databases it can be observed, that Cassandra and HBase perform nearly at the same level with HBase being slightly, but not significantly, faster.

Searching The second test measures the time taken by the search process. For allowing a fairer comparison between the two schemes the SWP scheme is slightly modified. In order for it to deliver the same kind of results as the SUISE scheme does, we allow the SWP scheme to abort the search as soon as it finds the first match within a text. That is sufficient to identify matches and include the affected row(s) to the result set when answering queries. Otherwise the SWP scheme could even tell the number and exact positions of matches within a text, but that functionality is not needed in the context of this thesis. SUISE cannot offer that information by design (like the majority of index-based SE schemes). After that modification of SWP both schemes deliver the same information, namely whether a document (table cell) contains the search word or not.

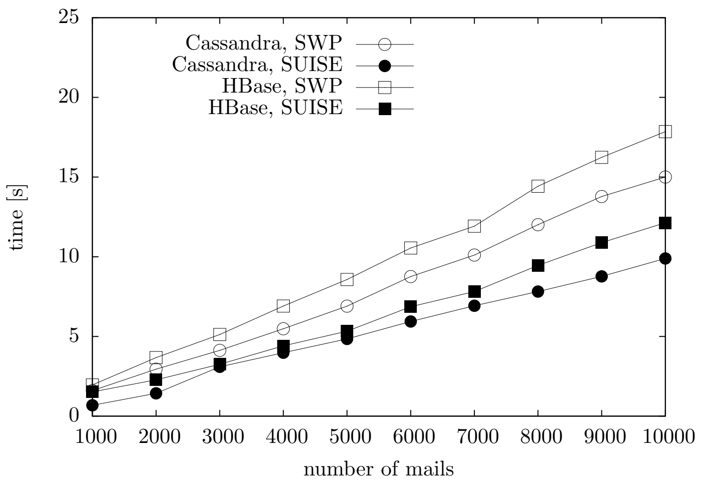


Figure 7.5.: Time needed for searching the data set with increasing size

Figure 7.5 shows the results. While SWP has to itereate over the ciphertext and SUISE has to iterate over its index γ_f both schemes require $O(n)$. Note that the measurements presented here for the SUISE scheme were taken with the search word being searched for for the *first* time, which results in a search time linearly growing with the data set, like it is the case for SWP. SUISE is also able to provide constant search time, when words are being searched for

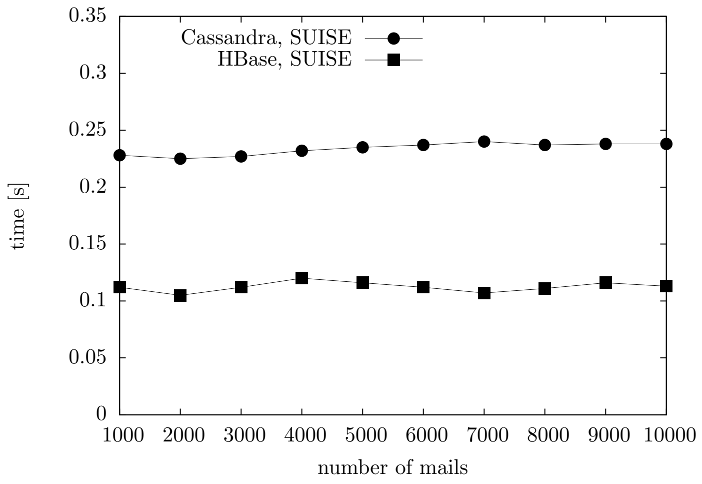


Figure 7.6.: Time needed for the searching previously searched words in SUISE

for a *second* time, thanks to its second index γ_w, that stores search results (see Figure 7.6).

Even with the SWP scheme being allowed to abort the search process after finding the search word once (which of course only helps, if it occurs at all) it is not faster than SUISE. Concerning the databases it can be stated, that Cassandra is faster, no matter which scheme is used. The difference is always about 20%. Note that there was not much of a difference during the encryption process (see previous test).

As mentioned before Figure 7.6 presents the results for a *second* search of the same word using the SUISE scheme. Using its second index it can deliver constant search time ($O(1)$). As the SWP scheme does not have a comparable feature, nothing can be presented on its behalf here. Since the second search is just a lookup in γ_w both databases are very fast (note the scale of time, compared to the previous tests). HBase outperforms Cassandra when it comes to a second search. It is roughly twice as fast here, which is exactly the opposite picture compared to a first time search.

Note that for SUISE γ_w fulfils the functionality of a cache that stores search results. There is no reason why a similar index would not work for SWP as well. However, the authors do not mention anything in that direction and an we do not implement a cache in this thesis either, because that would defeat the main advantage of SWP: no need for auxiliary data structures of any kind.

Summary Concerning the tested schemes for SE the following recommendations can be given:

- The SUISE scheme should be preferred if search performance is a primary concern. In this discipline it is faster than the SWP scheme, roughly by factor 1.3 to 1.5. Especially in scenarios, where it is very likely for certain words to be searched for more than once, SUISE is clearly the way to go, since it delivers outstanding constant performance in that case.

- The SWP scheme should be preferred if storage space and encryption performance is of primary concern, because SUISE advantage in searching comes with the price of maintaining two indexes, at least one of which is very large by design. Thus, in environments with smaller data sets, where search performance is not the key requirement, multiple searches for the same words are unlikely and disk usage matters, the usage of SWP is advisable. Its much smaller demand for disk space is not its only advantage. Since it is very simple in its design without indexes, it can be used for typical client server scenarios (like the scenario of this thesis) with much less effort. There is no need to implement and carry out index maintenance and index querying, because the search process is barely more than one simple iteration through the ciphertext.

Appendix A explores options for finetuning both schemes.

Concerning the databases it can be stated, that there is no clear winner. HBase is slightly faster during the encryption in SWP and a second time search in SUISE. Cassandra is faster in performing the search process.

7.1.2. API overhead

The benchmarks conducted in the previous section gave an insight into the PPE schemes used in this thesis in order to reveal their individual strengths and weaknesses. However, they can only be a hint (and serve as basis for the table profiles as introduced in Section 5.2.4) on what to expect in a real world scenario, because executing an actual query against encrypted data requires more than the previously benchmarked encryption/search steps. Using PPE in databases with arbitrary table structures additionally needs amongst other things (explained in detail in Section 5.5):

- translating queries to make them work with the encrypted tables on server side (see Section 6.3)
- managing metadata (see Section 5.3) in a consistent and persistent way to keep track of the mapping from plaintext to ciphertext data structures
- managing encryption keys (see Section 5.4) as well as RND and DET layer IVs (see Section 5.2.1.1)
- creating and maintaining column level PPE indexes on client and server side (if needed)

- decrypting the column set that the client application has asked for
- removing the RND layer from DET and OPE columns, if necessary

As can be seen from this (non-exhaustive) list, real world PPE usage in combination with the onion layer model requires some extra efforts. The following benchmarks take all of this additional overhead into account. Therefore the dataset as described in Section 7.1.1.3 is used again, but compared to the previous benchmarks the following tests differ in two key aspects. Firstly, while in the previous benchmarks for SE the individual mails have been written to the database in their raw format to evaluate the schemes throughput, they are now parsed by their data fields[4]. This allows querying for concrete information (like the sender, subject, timestamp, etc.) in the following benchmarks. All fields are encrypted/decrypted. No column remains in plaintext (compare Section 5.2.2). Secondly, in constrast to the tests conducted in Section 7.1.1, all database communication in the following benchmarks is done using the API as introduced in Section 5.6.1. That means the data flows are as described in Section 6.3 and in particular as shown in Figure 6.1. Thus, the encryption is no longer limited to a certain PPE scheme, but follows the onion layer model. That means every plaintext value is encrypted four times, if it is a string value (RND, DET, OPE and SE), three times, if it is a numerical value (RND, DET and OPE) or two times, if it is a byte blob (RND and DET) as described in Section 5.2.1. The software and hardware setup, as well as the dataset, remains the same as for the previous benchmarks in Section 7.1.1.

Encryption The first test measures the time for encrypting and inserting up to 10,000 mails ($1,03 \cdot 10^7$ words) using the available table profiles as introduced in Section 5.2.4.

Figure 7.7 presents the results for Cassandra, Figure 7.8 for HBase. Since all involved PPE schemes require $O(n)$ runtimes, it is no surprise, that the overall process does as well. The measurements shown in theses figures take everything into account as listed above, except for the parsing of the mails, as this a dataset related issue and has nothing to do with the PPE encryption or the onion layer model. As the number of encryptions caused by the onion layer model is high and there is no hardware support for OPE and SE, it is no surprise that a certain price in terms of runtime has to be paid for encryption. The results are very different for the available profiles. Since encrypting and inserting data requires mainly writes to the database, the profile for `OPTIMIZED WRITING` performs best, increasing the average insertion time compared to writing plaintext data to the database by a factor of 5.92 using Cassandra and 5.62 using HBase. It does not require any server side index maintenance, which is why encryption and insertion

[4]which are the following: ID, from, to, cc, bcc, subject, body, x-cc, x-bcc, x-folder, x-origin, x-filename, x-to, x-from, mime version, content transfer encoding, content type and timestamp. Additionally the size of a mail is stored as well.

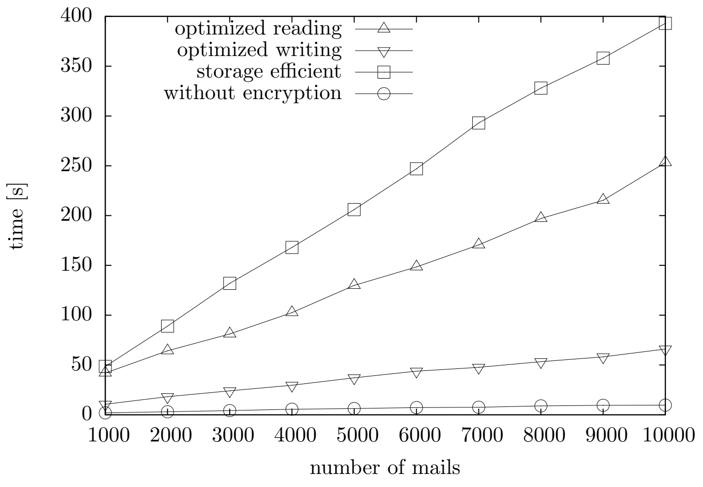


Figure 7.7.: Time needed for onion layer encryption in Apache Cassandra

into the database can be done a lot faster compared to the `OPTIMIZED READING`-profile, where these factors are 26.1 and 22.9. The `STORAGE EFFICIENT`-profile has the same advantage, but it suffers from the slow mOPE scheme in the OPE layer, which results in overall encryption factors of 40.4 and 34.5. Thus, the best performance in write-heavy scenarios, where querying is a rather rare event, is delivered by the profile for `OPTIMIZED WRITING`. By increasing the runtime by a factor of roughly 6 it is expensive, but in relation to the security that can be gained, it can still be a justifiable option that is feasible in most scenarios.

Concerning the databases one can observe, that there are only non significant differences. HBase is faster than Cassandra for the `OPTIMIZED WRITING`-profile by 5% and `STORAGE EFFICIENT`-profile by 9%. All other measurements are nearly identical.

Querying Querying is more complicated than encrypting. The runtime of a query depends on many aspects, for example the query type, the state of the onion layers and the PPE schemes used. These aspects are considered during the benchmarks in the following ways:

- First of all, five queries are tested, that involve dealing with different combinations of PPE schemes and are based on real world use cases:
    1. Q1 (DET): This query asks for all emails by a certain sender. Therefore it involves one equality check on the sender field.
    2. Q2 (OPE): This query asks for all emails larger than a certain size. That involves one order comparison on the size field.
    3. Q3 (SE): This query asks for all emails with a certain word in their body, which involves performing searchable encryption in the body

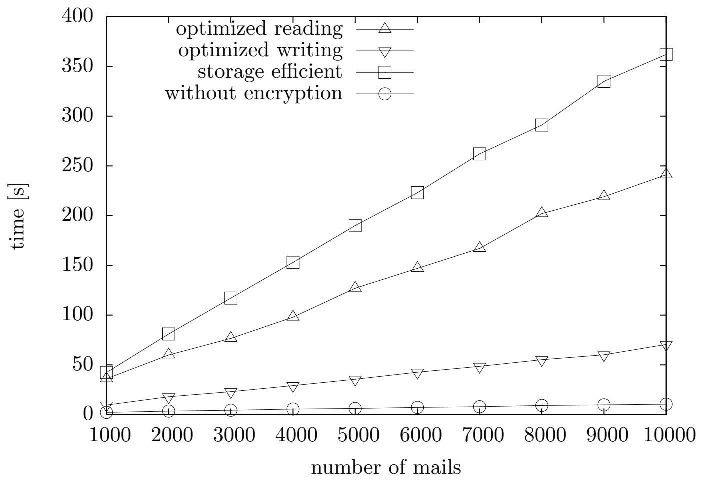


Figure 7.8.: Time needed for onion layer encryption in Apache HBase

field. Note that this is the worst case for SE in this scenario, since the body field is by far the largest. It comprises about 90% of the entire dataset.

4. Q4 (DET + OPE + SE): This query combines the filters of the previous queries Q1-Q3 and asks for all emails by a certain sender (equality check) with a certain minimum size (order comparision) and a certain word, which has to appear in the mail's body (word search).
5. Q5 (DET + 2x OPE + 2x SE): This is a more complicated query, that is supposed to evaluate more complex filters. It asks for all emails by a certain sender (equality check), that have been written in a certain time interval (two order comparisons with the start and endpoint of this interval) and contain certain words in their subject and body (requiring two applications of searchable encryption)

The concrete API calls that have been used to execute these queries for the following benchmarks can be found in Appendix B.

- When performing a query, it makes a significant difference in terms of runtime whether the same or a similar query has already been performed before or not, which the following two reasons are responsible. Firstly, if a column is involved in an equality check for the first time, the RND layer has to be removed from that column (see Section 5.2.1.1 and 5.2.1.2), which requires a certain amount of time. The column has to be read, decrypted and written back to the database. The same is true for columns involved in order comparisons. If a column is involved in equality checks or order comparisons in the future, this effort is not necessary again as the RND layer has already been removed. Secondly, when the SUISE scheme is involved in SE more than once, it might already have the search results in its index γ_w, which also can improve the query runtime significantly. For these two reasons every query is executed twice with Qx.1 indicating the first execution of the query Qx after encryption and Qx.x indicating the runtime, that can be expected from all future executions of Qx.

- Before an application can use the results of a query, it's resultset has to be decrypted again. Decryption is always done using the AES encrypted ciphertexts of the DET column. The decryption time of a result set heavily depends on its size. This size does not only depend on the query's filter conditions, but also on the columns that have been selected in the query. However, since the decryption speed of AES is not subject of this benchmark, every query only asks for the content of the ID column, which is enough to identify the set of mails that fulfill the query's conditions. That means, there is only one "word" to decrypt per mail in the resultset. This keeps the time needed for decrypting resultsets under 5ms for all queries and thus eliminates the decryption time as an disruptive factor, which the following benchmarks are not aiming for[5].
- Of course, another important runtime factor is the choice of PPE schemes, that were used for encryption in the first place. For this reason all queries are executed with all table profiles as introduced in Section 5.2.4.

All benchmarks have been conducted with the largest dataset that has been used in the previous encryption benchmarks, having a volume of $1.03 \cdot 10^7$ words in 10.000 emails.

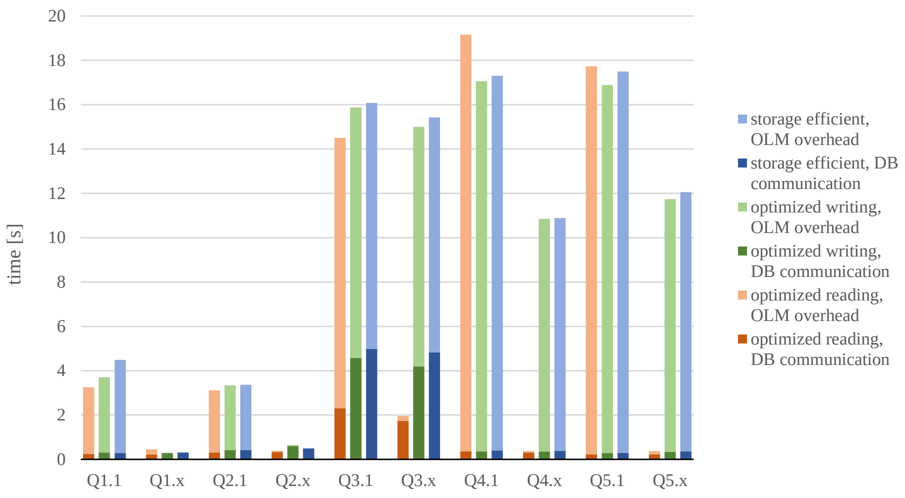


Figure 7.9.: Query runtime with Apache Cassandra

[5]Note that decryption does not cause a RND layer removal.

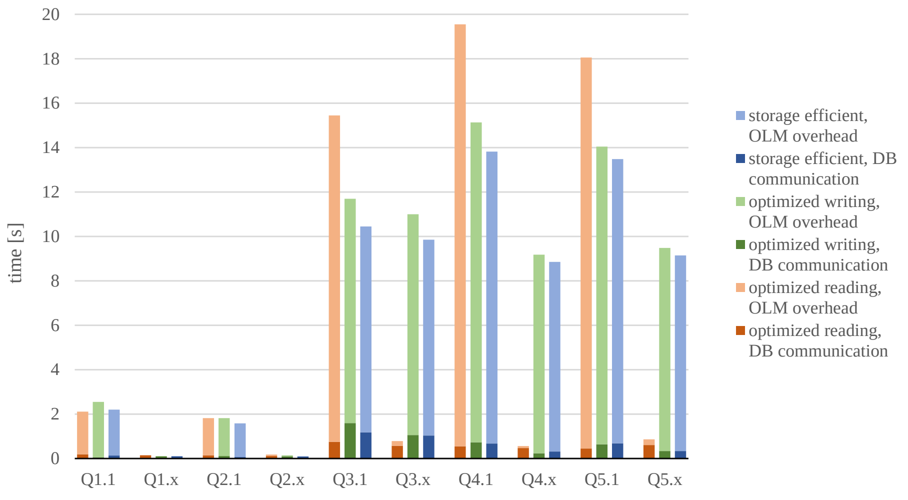


Figure 7.10.: Query runtime with Apache HBase

The results are presented in Figure 7.9 for Cassandra and Figure 7.10 for HBase. In these figures "DB communication" denotes the pure runtime of the databases' communication mechanisms (which is the execution time of driver calls in case of Cassandra and the execution time of HBase's native API calls) and "OLM[6] overhead" denotes everything that is a direct or indirect consequence of the onion layer model, for example query rewriting, RND layer removal or the SE processing as described in Section 5.5.2.1. The following observations can be made:

- If encryption was done using the table profile for `OPTIMIZED READING`, all Qx.x queries perform well under one second, except for Q3.x in combination with Cassandra. This can be considered practically feasible performance (compare [87]).
- Qx.x queries are always faster than Qx.1 queries. That means the performance always improves, if similar queries are executed.
- Performing SE is very expensive compared to requiring DET or OPE functionality only. This could have been expected for the reasons discussed in Section 5.5.2.1.
- HBase seems to have a slight overall performance advantage. A possible explanation for this might be the RND layer removal, which HBase is doing almost twice as fast as Cassandra. This can be seen in Q1 and Q2, in which most of the time is need for the RND layer removal.

[6]short for onion layer model

- Performing SE on small fields of data has barely a performance impact (compare Q4 and Q5, even though Q5 adds the subject field to SE, it is still faster than Q4 due to the smaller intermediate results after processing the DET and OPE conditions), but can slow down querying significantly, if done on a large subset of the data (compare Q1 and Q2 against Q3).

Table 7.1 investigates the concrete performance loss caused by PPE and the onion layer model. It compares the runtime for executing the queries Q1.x - Q5.x over unencrypted data and the runtime for the same queries executed over data that has been encrypted using the profile for optimized reading (see Section 5.2.4). This comparison is reasonable, because one will choose this profile if query runtime matters and most applications run similar queries mutliple times. It can be observed, that as long as SE is not involved (Q1 and Q2) queries take ca. twice as long in Cassandra. In HBase there is almost no performance loss on database side; the extra time can be explained by the decryption process. However, as soon as SE is involved, query runtimes increase roughly by factor 3-5. Note that in this case no measurements can be shown on behalf of Cassandra, because it does not support querying for single words within strings (see Section 5.5.2.1).

Table 7.1.: Comparison of overall query runtimes for Q1.x - Q5.x in seconds, the profile for optimized reading vs. unencrypted storage

Database		Q1.x	Q2.x	Q3.x	Q4.x	Q5.x
Cassandra	Opt. Reading	0.46	0.44	1.97	0.36	0.42
	none (unencrypted)	0.22	0.19	n/a	n/a	n/a
	factor of performance loss	2.09	2.32	n/a	n/a	n/a
HBase	Opt. Reading	0.19	0.21	0.79	0.57	0.88
	none (unencrypted)	0.17	0.16	0.15	0.17	0.16
	factor of performace loss	1.12	1.31	5.26	3.35	5.5

Summary NoSQL WCSs are by design databases with very limited query capabilities compared to standard SQL. They heavily profit from a table design that is tailored to the queries, that are to be performed later on. The same is true for using these databases with encrypted content. Knowing the scenario upfront (in particular the relation between reading from and writing to the database) and choosing the PPE schemes (via the given table profiles) appropriately has a crucial impact on the performance and can decide whether the encrypted database is practically feasible or not.

7.2. Functionality

Since the supported databases differ in functionality, it has to be evaluated for every supported database individually what functionality is still available and what functionality is lost due to the architecture of this thesis. Thus, this section also answers the question of what would be possible, if only one database was supported. That means, after the onion-layered PPE was applied, it has to be differentiated between functionality that is

- still possible: the functionality is available in the same manner as with unencrypted data (eventually with a non-significant overhead). However, some of it might not yet be implemented and thus can be considered subject of future work (see Section 8.2).
- theoretically feasible for the individual database: the functionality is possible, but not supported due to the unification of datamodels (see Section 5.2.5) or because it cannot be mapped to all available databases. It could be handled by the individual database, but cannot be realized in all databases (at least not without prohibitive overhead). Hence, this section describes the price to be paid for supporting multiple databases simultaneously.
- not possible: functionality that is possible with unencrypted data, but is impossible or involves a disproportionate effort when realizing it with PPE encrypted onion layer data.

Note that in order to not lose scope the lists in the following sections only present the most important aspects and are not meant to be exhaustive. Going into more detail is not expedient at this point.

7.2.1. Apache Cassandra

7.2.1.1. Still Possible

The following functionality is still available in Apache Cassandra:

- Almost all data types[7] can be used with PPE encryption, except for the ones mentioned in Section 7.2.1.3.
- The general CQL limitations[8] (for example regarding the size of partitions, lengths of keys or keyspace/table names, number of query parameters, maximum blob size, etc.) remain unaffected.

[7]see `https://docs.datastax.com/en/cql/3.1/cql/cql_reference/cql_data_types_c.html`. The support for booleans, frozens, lists and maps is not yet implemented, but possible.
[8]see `https://docs.datastax.com/en/cql/3.1/cql/cql_reference/refLimits.html`

- The same is true for the creation of keyspaces, in which Cassandra specific arguments[9] (for example the cluster internal data distribution strategy) can still be used.
- Altering keyspaces also works like with unencrypted data (for which the metadata does not even have to be changed).

Since the implementation of this thesis is of prototypical character, the following functionality is possible, but has not (yet) been implemented:

- Table properties[10] could be set like the keyspace properties mentioned above.
- Altering tables can be done like follows
  - Adding a column can be done trivially by creating and adding the corresponding metadata and the new columns inside the database, that represent the necessary onions.
  - Dropping a column can be done trivially as well by removing the column from the metadata and dropping the corresponding onion columns in the database.
  - Renaming a column can be done in two ways, depending on the desired result. Either the column is only renamed in the metadata (thus, the new name can already be used for queries) or the entire plaintext ciphertext mapping can be changed, which means the column is renamed in the database as well. However there is the question of why this would make sense, since the ciphertext name of a table already is a randomly generated string with no leakage, that (using the framework of this thesis) would be replaced by another randomly generated string causing no leakage.
  - Changing a column's data type is still possible, but very costly, since it may require completely new onions and corresponding columns[11].
- Expiring data with a time-to-live (TTL) works like with unencrypted data.
- The same is true for limiting the fetch size of a query.
- Granting or revoking user permissions on keyspace or table level is also still possible, but limited to permission types, that can be mapped to HBase's permission types. Thus, the available types would be `ALTER`, `CREATE`, `DROP`, `SELECT` and `MODIFY`.

[9]see `https://docs.datastax.com/en/cql/3.1/cql/cql_reference/create_keyspace_r.html`
[10]see `https://docs.datastax.com/en/cql/3.1/cql/cql_reference/create_table_r.html`
[11]Note that changing the column's datatype is already very restricted even without using encryption. When it changes, the bytes stored in values for that column remain unchanged. If the data cannot be deserialized to conform to the new datatype, the conversion fails, regardless of the encryption

7.2.1.2. Theoretically Feasible

The following functionality is theoretically feasible with PPE encrypted and onion layered data in Cassandra, but cannot be mapped to HBase.

- In the framework provided by this thesis creating tables can only be done with simple primary keys, even though Cassandra has very sophisticated mechanism for designing individual row identifiers, since HBase does not allow anything else. This problem is discussed in detail in Section 5.2.5.2.
- HBase also does not know the concept of triggers, which is why they are not supported by FamilyGuard. However, depending on the concrete trigger functionality, one could use HBase's coprocessors, but that could cause the need for further modifications of the database, which are not allowed in the scenario investigated by this thesis.
- Ordering result sets (like with `ORDER BY` in a CQL query) depends on compound keys in Cassandra, which cannot be supported for the reasons discussed in Section 5.2.5.2. However, sorting independently on client side is doable, if required.
- Secondary indexes could be used like with unencrypted data. However, HBase does not have a mechanism like that. Secondary indexes would have to be created and maintained manually. For reasons of programmatic complexity this is not supported in FamilyGuard.
- Batch operations consisting of `INSERT`, `UPDATE` and `DELETE` statements are generally possible with encrypted data in Cassandra and HBase. Unfortunately both databases treat batch operations differently. While Cassandra guarantees the execution order of the individual statements, HBase does not. That means with HBase it is mere coincidence whether values can be read that were written into the database within the same batch operation or not. Thus batch operations cannot be used in both databases consistently, which is why FamilyGuard does not support them.
- Creating user defined types could be realized by using PPE and the corresponding onion layer columns for every field individually. However, HBase does not know this concept. A work-around could be to use different column qualifiers to address different data fields.

7.2.1.3. Not Possible

While the onion layer model does not cause any restrictions, PPE comes with certain limits that have an impact on Cassandra's functionality.

- OPE encryrption requires to map values from a plaintext space to a much larger ciphertext space. For security reasons (see Section 7.3.1.3) the data type chosen for the ciphertext space should be the largest one available. In Cassandra this is `bigint`[12], which is equal to a 64 bit signed long in Java. The plaintext input should not be larger than half of this size in bits, which would be a 32 bit signed integer (the `int` type in Cassandra). Thus, this is the upper bound for numerical plaintext values, that can be stored, even though 64 bit ciphertexts are used to represent them in the database after encryption. However, unencrypted columns are still allowed to contain values larger than 32 bit.

- Floating point values cannot be stored (at least not OPE encrypted), because at this point in time no OPE scheme exists that is able to encrypt and decrypt floating point values with the precision necessary to cover the widely used IEEE-754 standard. This forbids the usage of Cassandra's `float` and `double` type. However it is not a problem to use them in unencrypted columns.
- Booleans cannot be stored in a property-preserving way, that Cassandra is able to handle natively. The same challenges as described in Section 5.5.2.1 arise.

A work-around for the problems described above could be storing all ciphertexts in form of byte blobs. While this would enable at least encrypted storage, only DET layer functionality would be available for reasons described in Section 5.2.1.2.

- CQL's commands for calculating the sum (`SUM`) and average (`AVG`) of a column's values are not available in encrypted columns for reasons already mentioned in Section 5.2.1.1. An extra onion for homomorphic encryption would be necessary, which is not only costly in terms of runtime, but would also require either to change the source code of the database to modify the way these calculations are done or to introduce cryptographic primitives into Cassandra UDFs (which is not possible with the current version[13]).

7.2.2. Apache HBase

As some aspects are very similar to the ones discussed for Cassandra in the previous Section, these points are listed in following section as well, but not discussed in the same level of detail.

7.2.2.1. Still Possible

The following functionality is still available in HBase:

[12]not to be confused with Java's `BigInteger` type that can store arbitrarily large numbers
[13]which is Cassandra 3.10

- `PUT`, `GET` and `DELETE` requests can be used with PPE ciphertexts like with plaintexts.
- Altering tables can be done like in Cassandra, except for the renaming process. HBase does not support renaming tables natively, but a work-around is cloning snapshots using a new table name and then delete the old table[14]. However, when using the API provided by FamilyGuard it would be sufficient to change the table name in the metadata to enable usage of the new table name in queries.
- Limiting the fetch size of a query is not affected by the onion layer model or PPE.
- Expiring data with a TTL also works like with unencrypted data.
- The same is true for granting and revoking user permissions.
- All Hbase-individual parameters for creating tables (e.g. blocksize, TTL, etc.) can be used like with unencrypted data[15].

7.2.2.2. Theoretically Feasible

The following functionality is theoretically still feasible with PPE encrypted and onion layered data in HBase, but cannot be mapped to Cassandra.

- In contrast to Cassandra, HBase allows versioning not only via a specified TTL, but also via a pre-defined maximum number of versions that are stored per table cell, before inserting new data causes deletions of old data.
- HBase supports additional operations, that could be used in batch operations: `increment` and `append`, both not being available in Cassandra.
- HBase allows a more fine-grained adjustment for user permissions. Granting and revoking is available also on column and even cell level, whereas Cassandra only supports user permissions on keyspace (namespace) and table level.

7.2.2.3. Not Possible

The problem of storing very large numerical values, floating point and boolean values as described previously in Section 7.2.1.3 is caused by the concept of PPE and hence is also a problem in HBase.

[14]see `http://hbase.apache.org/book.html#table.rename`
[15]see `https://hbase.apache.org/book.html#schema`

Another loss of functionality occurs by using SE. HBase's Java API features computationally expensive, but also powerful filters for querying texts, e.g. scanning for prefixes or matching regular expressions. SE does not allow such things. Due to the working principles of SE (see Section 2.4.2.1) everything that can be matched by an SE scheme's search algorithm has to be predefined in advance and can only be matched in that exact form. In practise those are extracted words, usually separated by white space characters[16].

7.3. Security

A formal security analysis of OPE and SE was already given in Section 2.4. The following section provides a security analysis from a more practical point of view. It reviews security aspects of the individual PPE schemes used in this thesis and discusses their concrete security leaks, that have been introduced formally in Section 2.4.

7.3.1. PPE related issues

7.3.1.1. Deterministic Encryption

As described in Section 4.2.2, both schemes used for the DET layer in this thesis, AES and Blowfish, leak the equality of plaintexts, if used with the same encryption key and IV for all plaintexts. Besides this intentional leakage, there are no known attacks of practical relevance.

7.3.1.2. Searchable Encryption

SWP

The SWP scheme (see Section 4.4.2.1) does not need an index and thus does not leak any index information (see Section 2.4.2.2) like for example the number of words or documents. By using a fixed word length for its ciphertext output it also hides the true length of plaintext words. However it leaks search patterns, since the pre-encrypted search words are deterministically encrypted, which allows linking them to actual plain words. SWP also leaks access patterns, which is unavoidable in the client server scenario, in which a request (pre-encrypted searchword) can always be linked to the corresponding answer (the result set).

[16]There are SE schemes that support at least encrypted fuzzy search, meaning words with a certain distance to the searchword can be found, but that helps only very little when doing prefix search and not at all when trying to check for matches with regular expressions. Furthermore fuzzy search is too computationally expensive for being used in the context of this thesis

SUISE

The SUISE scheme (see Section 4.4.2.2) uses two indexes and thus, leaks index information. The number of encrypted items per column is equal to the number of rows in the index γ_f. While this is an information the database is able to see anyway, the number of unique words per document is equal to the set size in its corresponding row in γ_f, which might be a more valuable information for an attacker. Since search tokens are generated deterministically (like the pre-encrypted searchwords in SWP) the search pattern leaks, because that enables an attacker to link search tokens to plain words, although the plain words remain hidden. Even if that was not the case the database server could learn, if a word has been searched for before, because if not, both indexes have to be accessed.

Access patterns leak for the same reason as described previously for SWP. Since trapdoors have a constant length and the information outside the indexes is encrypted using a block cipher (AES) the length of plaintext words is hidden.

7.3.1.3. Order Preserving Encryption

In contrast to SE the working principles of almost all OPE schemes are the same: when a range query has to be performed, the plaintext boundaries in the query are replaced by the encrypted boundaries of the search term. The database performs a regular range query as it would do on plaintext data using these encrypted boundaries, since the order of the plaintexts is preserved in the ciphertexts. Because this procedure is the same for all OPE schemes used in this thesis, their leakage is quite similar. As already mentioned in Section 2.4.3, specifying the theoretical leakage of OPE schemes is still an open problem, but some observations can be made in the practical context.

OPE schemes cannot leak any index information, because the index (if exists) resides on client side. Search patterns can leak, because OPE schemes produce deterministic ciphertexts[17] that are used for queries and can be tracked. Access patterns leak for the same reasons as explained earlier for SE.

Furthermore, the security of all OPE schemes heavily depends on the relation between the sizes of plaintext space and ciphertext space: the more ciphertexts potentially available per plaintext, the better.

Another security risk for OPE is having only very few or many values of a domain actually being encrypted. This can be explained very easily considering the extreme examples: On the one hand, if only two values of a domain p_1 and p_2 are encrypted, they can easily be mapped to their corresponding ciphertexts c_1 and c_2. Obviously the smaller p value is encrypted in the smaller c value and the larger p value is encrypted in the larger c value. There is no other option. On the other hand, it is equally severe if all values of a specific domain are to be encrypted. The ordered ciphertexts can simply be matched to the ordered plaintexts. [64] showed, that under certain conditions this can even be achieved

[17] with only very few exceptions, for example [49] and [53]

for schemes like mOPE (see Section 4.3.2.3), that try to hinder such an attack by adding a secret offset. Note that both problems also occur in non-deterministic OPE schemes. That means it makes sense to think about how to store data in certain scenarios. For example it is much more secure to store a date in form of a unix timestamp, which results for every date in a different 32 bit integer value, compared to storing it split in individual characteristics like day (domain size only 31), month (domain size only 12) and year (reasonable domain size depending on the scenario).

7.3.2. Onion-Layer-Model Related Issues

The onion layer model as described in Section 5.2.1 is able to hide the structure of the plaintext table up to a certain degree, even if the attacker knows about it. Every plaintext column results in one, two or three ciphertext columns representing the onions depending on its data type. Using the API proposed in this thesis, every one of these ciphertext columns is of data type byte blob, no matter what type of content it represents. As soon as there are more than two columns in an encrypted table (which is the absolute minimum[18] for the model as introduced in Section 5.2.1) the attacker is not able to make reasonable assumptions about the structure of the originating plaintext table. For example two (non-IV) ciphertext columns can originate from two byte blob columns or one integer type column, three ciphertext columns can originate from one text column, one integer and one byte blob column or three byte blob columns, etc. The more ciphertext columns a table consists of, the more originating combinations are possible. Since the column names of the ciphertext columns (and tables) are also generated randomly, the attacker does not get any clues from them either.

Thus, (initially) there is no way for an adversary to figure out to which onion or plaintext data a certain ciphertext column belongs. The plaintext table scheme remains hidden. However, if the attacker is familiar with the used onion layer model, they could recognize the event of removing the RND layer from a DET or OPE onion representing column, because this requires reading from and updating every value of this column. In connection with the query that caused the removal (equality check or order comparison), it is possible to determine the onion type that the column is responsible for (DET onion or OPE onion). However, that still does not help to identify neither the data type nor the name of the originating plaintext column. The only information that can actually be inferred is the following: if the attacker was able to identify an OPE onion column, that means the plaintext table must contain either an integer or a text column, because plaintext byte blob columns do not result in OPE columns. If however reading and updating all values in a column is an event that occurs

[18]The smallest possible table consists of the IV column and the OPE encrypted row identifier column.

on a regular basis in the domain of the database's content, RND layer removals might not even attract the attention of a possible attacker.

If the attacker knows about the existence of OPE columns (and the onion layer design of this thesis), they might be able to identify them based on the fact, that all of their contents have the same length[19]. This can be a security risk in particular if other columns of the table were stored unencrypted in the first place. Remember for instance the example shown in Figure 5.7. Knowing the concept of OPE and the corresponding column, the attacker would be able to identify the person with the highest salary[20]. However, distributing tables across independent database instances as described in Section 5.2.2 is an effective countermeasure.

If the attacker gets access to the metadata file, that will help them to completely reveal the plaintext table structure, but it still does not enable her to decrypt the actual contents. Therefore the password of the JCE keystore is required.

[19]To be exact: this is also true for the column that stores the IVs. However these are only of 16 byte length, which does not qualify for OPE, which the attacker might know about.

[20]but still not the exact amount

8 Conclusion

This final chapter of the thesis summarizes findings and contributions. Furthermore, an outlook on future work in this research direction is given.

Contents

8.1. Discussion

In this section strengths and weaknesses/limitations of the approach as proposed in this thesis are pointed out.

8.1.1. Strengths

The approach of this thesis as introduced in the Chapters 4-7 has the following advantages that are unique in this combination, compared to related work

- By utilizing SUISE ([45], see Section 4.4.2.2) for SE as well as OACIS ([54], see Section 4.3.2.2) and RSS ([100], see Section 4.3.2.1) for OPE the architecture proposed in this thesis takes advantage of index-based PPE schemes, that lead to a better performance for querying larger datasets.
- No proxy-client or any other additional architectural component is required. The client application talks directly to the database server(s). This avoids extra network traffic and reduces latencies. Thus, it leads to a better query performance.
- The database server(s) remain unmodified. That means the approach of this thesis can be extended to databases, that are not open source. Furthermore no specialized cryptographic hardware is necessary. Hence, offers from today's real-world cloud database providers can easily be used.
- Very different native query mechanisms (CQL of Cassandra, the native Java API of HBase) are unified in an easy to use API. Others can be integrated as well by implementing a single abstract Java class (see Section 6.4.1), most likely resulting in no more than ca. 300 lines of code.
- The same is true for the corresponding resultset objects. 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 name (which is not possible in the native resultset objects of Cassanda and HBase), no matter what underlying database was used.
- The user can optimize the performance by selecting PPE schemes based on profiles for certain use cases (see Section 5.2.4) or exclude non-sensitive data columns from encryption (see Section 5.2.2).
- 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.
- Simple interfaces can be implemented to support further databases and PPE schemes (see Section 6.4).

8.1.2. Limitations/Weaknesses

The limitations of this thesis' approach are mainly the limitations of PPE:

- OPE schemes do not support encrypting floating point values. Hence, they cannot be stored in FamilyGuard.
- For some datatypes there are no PPE schemes available. A possible solution is a representation in another format. For example a boolean value can be represented as numerical value (true = 1, false = 0) or as string (true = "true", false = "false"), a date/time type can be represented as unix timestamp or split up in multiple numerical values for day, month, hour, minute, etc.
- PPE encrypted ciphertexts are usually longer than their corresponding plaintexts. OPE-encrypted numerical values are at least twice as long in bit length compared to the corresponding plaintexts and SE-encrypted words are much longer than the average length of a word in the English (or almost any other) language (see Appendix A).
- The effect of this problem is amplified by the fact, that the onion layer model causes the storage of every plaintext multiple times using various PPE schemes of different categories. However, in cloud scenarios storage space is usually not a problem.

Besides the limitations caused by PPE a weakness of the architrecture as introduced in Chapter 5 is the fact that applications have to be changed. However, as only the original database interactions have to be replaced by the corresponding API calls (see Section 5.6), these changes do not cause much effort.

Nevertheless, as the data model and working principles of WCSs remain unchanged and are not affected by the architecture proposed in this thesis, the "right" way to use these databases remains unchanged, too. With or without PPE they heavily profit from table layouts, that are tailored to the queries appearing later on (as described in Section 2.1.2).

8.2. Future Work

The area of processing queries over encrypted data as well as this thesis itself leaves room for further research. A few directions are pointed out as follows.

- While the concepts in this thesis are designed for WCS, they are not limited to them. The data model of WCSs can be mapped easily to key-value stores (in which the key might be composed as `table:column:qualifier:timestamp`) or document stores (where rows can be mapped to documents and columns can be mapped to fields of a document). The general idea of doing so pops up in literature every now and then (e.g. [8, 102]) and Section 6.4.1 discusses the effort that would be necessary in case of extending the implementation of this thesis.

- The API can be integrated into a proxy client between application and database server. In this way, no modifications to the application would be necessary, but an additional architectural component is introduced (as done e.g. the approach of "CryptDB", see Section 3.1.1). A step further would be the integration of the onion layer model and PPE schemes into the database drivers/native APIs, combining the architectural simplicity of the approach of this thesis with the opportunity to leave the client application as well as the database server unchanged. Of course this solution would be very database specific and the option to transparently use different databases would not be available.
- SE schemes for multi-keyword-search (sometimes also called conjunctive keyword search) (e.g. [16, 90, 98]) allow searching for multiple keywords simultaneously. Even though their encryption step is usually more complex, they might significantly speed up queries involving corresponding searches. Thus, practical evaluations are needed, maybe using an extra onion.
- The table profiles as introduced in Section 5.2.4 can be used more fine-grained by applying them to individual columns instead of whole tables. Depending on the concrete scenario this enables further performance improvements, if read/write ratios differ significantly for different columns of the same table. However, this would make the table creation process (see Section 5.6.1.4) more complex.
- Besides the differentiation of onions based on text, numerical values and byte blobs, more onions are possible, "specialized" on certain data types (e.g. booleans, timestamps, etc.).
- The API as introduced in Section 5.6 can be extended to support simple aggregations on client side, like calculation of sums or averages. Thus, no extra onion for homomorphic encryption has to be kept on client side for that purpose.

8.3. Summary

In Chapter 2 the foundation of this thesis was laid by identifying the characteristics of NoSQL WCSs data model that are relevant for this work, most important the role of row identifiers. Afterwards Apache Cassanda and HBase were presented as the basis of the architecture, implementations and experiments later on. Both databases were discussed in detail and compared regarding their fundamental working principles. It followed a firm description of cloud computing as the definition of this term is often very blurry in literature, including a description of how Cassandra and HBase work in this context. After having completed the technical aspects, Chapter 2 moved on to the description of the "honest-but-curious" adversary scenario as it is the basis of all further considerations in this work. The Chapter is completed by a comprehensive introduction into property-preserving encryption, explaining its basic building blocks as well as its general working principles and security definitions.

Chapter 3 presented an overview on related work. Since the research field of enabling databases to process queries over encrypted data is quite large, the discussions in this chapter are limited to approaches that are also designed for the "honest-but-curious" adversary as introduced in Chapter 2, compute over encrypted data and rely on encryption as the only mechanism to provide data confidentiality[1]. Since the number of these approaches is still quite high they were split into further sub-categories, to which the following observations could be made. Approaches that rely on either relational databases often suffer from scalability issues and limited query expressiveness. Approaches that rely on non-relational databases (like this thesis) have the problem of not being practical due to significant storage inefficiency or computation overhead. Approaches based on homomorphic encryption are not (yet) practically computable either. Finally, approaches relying on specialized hardware are expensive and thus not adopted by real world cloud database providers.

Chapter 4 paved the way for designing an own architecture. It explained the purpose of PPE schemes in the context of processing queries over encrypted data. It grouped the schemes into approaches for deterministic, order-preserving and searchable encryption. For each of these categories the following contributions were made. Firstly, the exact requirements were identified that schemes of their category are supposed to meet in order to be useful in a WCS database scenario as described in Chapter 2. Secondly, various schemes were surveyed and evaluated based on these requirements. Applicable schemes were identified and selected for implementation (Chapter 6) and experiments (Chapter 7) later on. For some of them modifications were proposed and implemented to improve their practicability. Finally, schemes not fulfilling the requirements as introduced above were discussed to point out the reasons for their inapplicability.

Having laid the theoretical foundations in the previous chapters, Chapter 5 proposed the first architecture for a practical usage of PPE schemes in WCSs, starting with the organization of data on server side. Therefore an adapted version of [73]'s onion layer model was proposed, as well as optional table fragmentation across multiple independent database instances. The PPE schemes selected in Chapter 4 were grouped into profiles for use cases with certain characteristics (write-heavy, read/query-heavy, storage limited) to ease their practical

[1]Note that the mechanisms for data fragmentation over multiple database instances for improving security as described in Chapter 5.2.3 is optional, not mandatory

application. Afterwards the metadata on clientside, that is necessary to keep track of the encrypted data structures on serverside, was identified, . Since some of the PPE schemes require multiple cryptographic keys, a solution for their management on clientside was discussed as well. Afterwards it was shown, how metadata and cryptographic keys must be incorporated when pre-processing queries. Therefore the steps necessary for read and write operations were presented in detail. Finally an API for applying the PPE schemes in the above proposed onion layer model was introduced, that hides the complexity of the PPE schemes and onion layer model from the user and provides methods for the main tasks that occurring in a database scenario (like e.g. creating tables, inserting rows, querying). The proposed API works independent from whether Cassandra or HBase is used as underlying technology and can be easily extended to support other database systems and PPE schemes as well. Different databases can even be used for columns of the same table, which is a unique feature of this thesis' approach.

Chapter 6 presented the building blocks of the API implementation. Thus it discusses aspects like communication to the databases, cryptographic primitives and libraries as well as management of information that is stored on clientside. Furthermore it showed, how other PPE schemes or databases can be integrated easily in the future.

An evaluation of the approach of this thesis as described in Chapter 4 and 5 was presented in Chapter 7, focussing on three aspects. The first one is performance. Thus, for individual PPE schemes selected in 4 as well as for using the API as introduced in Chapter 5 detailed measurements were provided, showing the time necessary to insert data in encrypted form and query against it. Using a real world data set with realistic queries, the performance loss caused by the encryption steps was quantified. It could be observed that selecting the PPE schemes carefully according to the later database usage can lead to feasible response times. The second aspect is functionality. Therefore detailed descriptions were given of what functionality is affected by using PPE schemes in Cassandra and HBase. It was shown that the databases keep the majority of their functionality. However some functionality cannot be maintained due to the lack of support by the PPE schemes (e.g. floating point numbers cannot be encrypted using OPE). The third aspect is security. PPE schemes as well as the onion layer model leak information unintentionally. Hence, Chapter 7 concluded with a characterization of this leakage.

Chapter 8 gave this summary, discussed the strengths and weaknesses of the approach presented in this thesis, and pointed out directions for future extensions of this work.

A

Appendix - Towards Optimizing Searchable Encryption

The schemes for SE introduced in Chapter 4.4.2.1 and 4.4.2.2 can be finetuned in terms of speed and storage efficiency.

A.1. Optimizing SWP

In the SWP scheme the common word length n is the most important parameter. As described in Chapter 4.4.2.1 words smaller than n have to be padded. Words larger than n have to be split (multiple times eventually). Padding leads to more bytes in the ciphertext. Splitting words leads to more words in the ciphertext, and thus more iterations during the encryption and search process. That means in practice: a small n can cause a slower performance, but saves disk space. A larger n leads to more consumption of disk space due to large ciphertext words, but it can accelerate the encryption/search process, since less iterations are required. In order to examine the actual impact of changing the value of the word length n we ran a test starting from $n = 4$ up to $n = 9$, assuming the average word length of the data set is within that range.

Figure A.1 presents the results. As the ciphertext increases linearly with n growing, the performance does not get significantly better after $n = 8$, which is precisely the reason, why the previous tests were conducted with $n = 8$ as parameter for the word length in SWP. It represents the best compromise between performance and disk usage. Strictly speaking, this examination should be performed for every data set individually and as data sets in practice tend to grow over time, it might be difficult to predict an appropriate word length n, since it cannot be changed during future use without re-encrypting the whole data set.

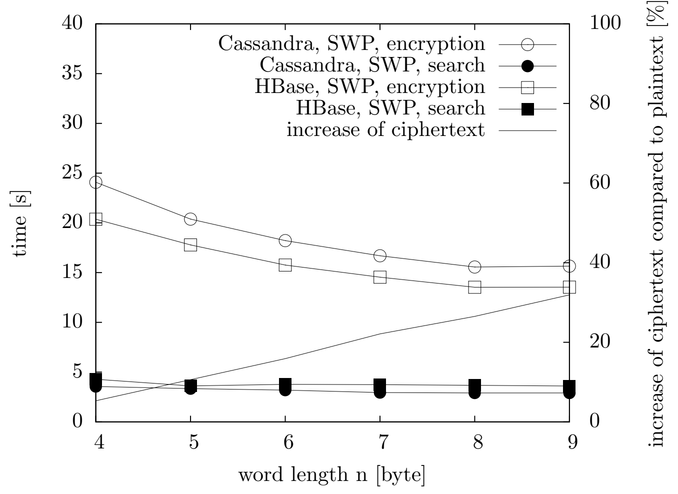


Figure A.1.: performance and disk usage of SWP with increasing n

Another option for optimizing SWP in the context of this thesis is given by the onion layer model as follows. The original work of the authors introduced the SWP algorithm in 4 tiers, the last of which is an extension that enables the user to decrypt the ciphertexts produced by their algorithm (referred to as “final scheme” in Chapter 4.4.2.1). The onion layer model already uses the ciphertexts of the DET column for decryption, since with AES that can be done much faster. That means tier 3 of the SWP scheme (“hidden searches”) is sufficient for providing all functionality and security needed for this thesis. It avoids a number of computationally expensive steps of the final scheme, which speeds up encryption as well as searching.

A.2. Optimizing SUISE

SUISE does not leave room for improvements regarding the encryption and decryption speed like SWP. Thus the only options for improvement concern its storage efficiency. SUISE has to maintain two indexes, which can become quite large. While it is not possible to make general assumptions regarding the size of γ_w, as its size highly depends on the search patterns, it can be stated, that γ_f grows fast in relation to the size of the plaintext data. This can easily be explained pointing out the fact, that every plaintext word, no matter how long it is, gets stored in γ_f in form of $H_{r_w}(s)||(s)$ (see Chapter 4.4.2.2). Using HMAC-SHA-1 for H, as suggested by the authors [45], that leads to 20 + x bytes, depending on the length x chosen for the pseudorandom values s (= output length of G). The authors used 160 bits as output size of G, which results in another 20 bytes. In other words: every unique word of the plaintext will consume 40 bytes in γ_f, even if that word originally consisted of only one

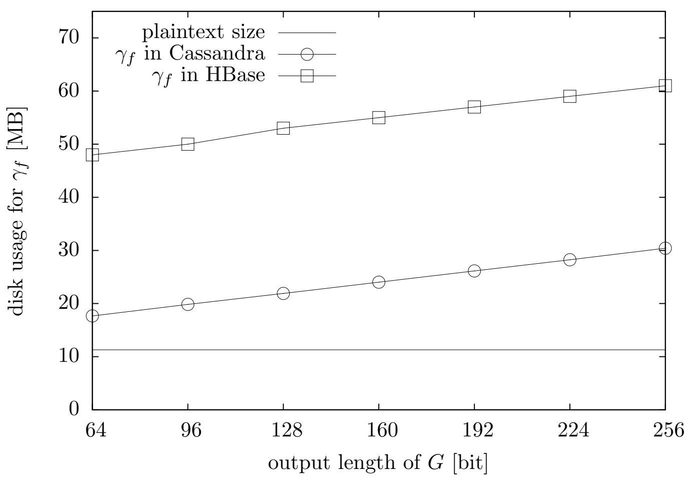


Figure A.2.: size of γ_f with growing length of s

byte. That means even though only unique words are stored in γ_f (which make just 40% of the total number of words, as we showed earlier), it can be expected that γ_f is a multiple in size compared to the plaintext. In practice this situation can get even worse, depending on how the databases store the index internally.

Figure A.2 presents the actual size of γ_f when stored in Cassandra and HBase. Input for this examination were 2000 randomly chosen emails from the dataset. As can be seen the size of γ_f grows linearly with the length of the output s of the pseudorandom generator G used by SUISE. By following the authors' value of 160 bit we already get an index more than twice as big as the plaintext data set. Note that this is pure index size. It does not include the actual encrypted files. Using HBase this issue becomes quite severe. Figure A.2 also shows, that using smaller output lengths for G does not improve this situation very much[1]. Summing up, the index size of γ_f can be reduced a little if saving storage space is important. However, in most practical cloud computing scenarios storage space is not an issue.

[1]The performance with different output lengths of G was also tested with no evidence for a significant gain or loss

B

Appendix - Benchmark Queries

Q1

A query for the IDs of all mails of a certain sender (sender=eric.bass@enron.com).

```
api.query(new String[]{"id"},
        "enron", "mail",
        new String[]{"sender=eric.bass@enron.com"});
```

Q2

A query for the IDs of all mails that are larger than 5000 bytes.

```
api.query(new String[]{"id"},
        "enron", "mail",
        new String[]{"size<5000"});
```

Q3

A query for the IDs of all mails that have the word “party” in their body.

```
api.query(new String[]{"id"},
        "enron", "mail",
        new String[]{"body#party"});
```

Q4

A query for the IDs of all mails with eric.bass@enron.com as sender, that are larger than 5000 bytes and have the word “party” in their body. Thus, it combines the filter conditions of Q1, Q2 and Q3.

```
api.query(new String[]{"id"},
        "enron", "mail",
        new String[]{"sender=eric.bass@enron.com", "size<5000",
            "body#party"});
```

Q5

A query for the IDs of all mails, that came from eric.bass@enron.com, written between the 1st[1] and 31st[2] January 2002 with the word “Bachelor” in the subject and the word “party” in the body.

```
api.query(new String[]{"id"},
           "enron", "mail",
           new String[]{"sender=eric.bass@enron.com",
                    "timestamp>1009843200",
                    "timestamp<1012435200",
                    "subject#Bachelor",
                    "body#party"});
```

[1]The unix timestamp of 01/01/2002 00:00am is 1009843200.
[2]The unix timestamp of 31/01/2002 00:00am is 1012435200.

Bibliography

[1] Gagan Aggarwal, Mayank Bawa, Prasanna Ganesan, Hector Garcia-Molina, Krishnaram Kenthapadi, Rajeev Motwani, Utkarsh Srivastava, Dilys Thomas, and Ying Xu. Two can keep a secret: A distributed architecture for secure database services. *CIDR 2005*, 2005.

[2] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 563–574. ACM, 2004.

[3] Ihsan H Akin and Berk Sunar. On the difficulty of securing web applications using CryptDB. In *Big Data and Cloud Computing (BdCloud), 2014 IEEE Fourth International Conference on*, pages 745–752. IEEE, 2014.

[4] Halderman Alex, Seth Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph Calandrino, Ariel Feldman, Jacob Appelbaum, and Edward Felten. Lest we forget - cold boot attacks on encryption keys. *Proceedings of the 17th Usenix Security Symposium*, 2008.

[5] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide*. O'Reilly Media, Inc., 2010.

[6] Ross Anderson, Eli Biham, and Lars Knudsen. Serpent: A proposal for the advanced encryption standard. *NIST AES Proposal*, 174:1–23, 1998.

[7] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. Orthogonal security with Cipherbase. In *CIDR*, 2013.

[8] Paolo Atzeni, Francesca Bugiotti, and Luca Rossi. SOS (Save Our Systems): A uniform programming interface for non-relational systems. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 582–585. ACM, 2012.

[9] Sumit Bajaj and Radu Sion. TrustedDB: A trusted hardware-based database with privacy and data confidentiality. *Knowledge and Data Engineering, IEEE Transactions on*, 26(3):752–765, 2014.

[10] Josh Benaloh. Dense probabilistic encryption. In *Proceedings of the workshop on selected areas of cryptography*, pages 120–128, 1994.

[11] Eli Biham and Orr Dunkelman. Cryptanalysis of the A5/1 GSM stream cipher. In *Progress in Cryptology*, pages 43–51. Springer, 2000.

[12] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. Biclique cryptanalysis of the full AES. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 344–371. Springer, 2011.

[13] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam Oneill. Order-preserving symmetric encryption. In *Advances in Cryptology-EUROCRYPT 2009*, pages 224–241. Springer, 2009.

[14] Alexandra Boldyreva, Nathan Chenette, and Adam ONeill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Advances in Cryptology–CRYPTO 2011*, pages 578–595. Springer, 2011.

[15] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-DNF formulas on ciphertexts. In *Theory of Cryptography Conference*, pages 325–341. Springer, 2005.

[16] Dan Boneh and Brent Waters. Conjunctive, subset, and range queries on encrypted data. In *Theory of Cryptography Conference*, pages 535–554. Springer, 2007.

[17] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, et al. Apache Hadoop goes realtime at Facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1071–1080. ACM, 2011.

[18] Christoph Bösch, Pieter Hartel, Willem Jonker, and Andreas Peter. A survey of provably secure searchable encryption. *ACM Computing Surveys (CSUR)*, 47(2):18, 2015.

[19] Zvika Brakerski and Renen Perlman. Lattice-based fully dynamic multikey fhe with short ciphertexts. CRYPTO, 2016.

[20] Eric Brewer. A certain freedom: thoughts on the CAP theorem. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 335–335. ACM, 2010.

[21] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, 2000.

[22] Josiah L Carlson. *Redis in Action*. Manning Publications Co., 2013.

[23] Rick Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4):12–27, 2011.

[24] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[25] Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Applied Cryptography and Network Security*, pages 442–455. Springer, 2005.

[26] Nathan Chenette, Kevin Lewi, Stephen A Weis, and David J Wu. Practical order-revealing encryption with limited leakage. 2015.

[27] Kristina Chodorow. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 2013.

[28] Pawel R Chodowiec. *Comparison of the hardware performance of the AES candidates using reconfigurable hardware*. PhD thesis, George Mason University, 2002.

[29] Sherman SM Chow, Jie-Han Lee, and Lakshminarayanan Subramanian. Two-Party Computation Model for Privacy-Preserving Queries over Distributed Databases. In *NDSS*, 2009.

[30] Valentina Ciriani, Sabrina De Capitani Di Vimercati, Sara Foresti, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. Combining fragmentation and encryption to protect privacy in data storage. *ACM Transactions on Information and System Security (TISSEC)*, 13(3):22, 2010.

[31] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 79–88. ACM, 2006.

[32] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Workshop on the Theory and Application of Cryptographic Techniques*, pages 10–18. Springer, 1984.

[33] Michael L Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *Journal of the ACM (JACM)*, 31(3):538–544, 1984.

[34] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[35] Craig Gentry, Shai Halevi, and Nigel P Smart. Homomorphic evaluation of the aes circuit. In *Advances in Cryptology–CRYPTO 2012*, pages 850–867. Springer, 2012.

[36] Eu-Jin Goh et al. Secure indexes. *IACR Cryptology ePrint Archive*, 2003:216, 2003.

[37] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. Sirius: Securing remote untrusted storage. In *NDSS*, volume 3, pages 131–145, 2003.

[38] Oded Goldreich. Foundations of cryptography - fragments of a book, 1995.

[39] Oded Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge university press, 2004.

[40] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. On the cryptographic applications of random functions. In *Advances in Cryptology*, pages 276–288. Springer, 1985.

[41] Shafi Goldwasser and Silvio Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 365–377. ACM, 1982.

[42] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[43] Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 216–227. ACM, 2002.

[44] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

[45] Florian Hahn and Florian Kerschbaum. Searchable encryption with secure and efficient updates. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 310–320. ACM, 2014.

[46] Jing Han, E Haihong, Guan Le, and Jian Du. Survey on NoSQL databases. In *Pervasive Computing and Applications (ICPCA), 2011 6th International Conference on*, pages 363–366. IEEE, 2011.

[47] Nam-Su Jho, Ku-Young Chang, Dowon Hong, and Changho Seo. Symmetric searchable encryption with efficient range query using multi-layered linked chains. *The Journal of Supercomputing*, pages 1–14, 2015.

[48] Christine Jost, Ha Lam, Alexander Maximov, and Ben JM Smeets. Encryption performance improvements of the paillier cryptosystem. *IACR Cryptology ePrint Archive*, 2015:864, 2015.

[49] Hasan Kadhem, Toshiyuki Amagasa, and Hiroyuki Kitagawa. MV-OPES: multivalued-order preserving encryption scheme: A novel scheme for encrypting integer value to many different values. *IEICE TRANSACTIONS on Information and Systems*, 93(9):2520–2533, 2010.

[50] Hasan Kadhem, Toshiyuki Amagasa, and Hiroyuki Kitagawa. A secure and efficient order preserving encryption scheme for relational databases. In *KMIS*, pages 25–35, 2010.

[51] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 965–976. ACM, 2012.

[52] A Kasten, A Scherp, F Armknecht, and M Krause. Towards search on encrypted graph data. *Proceedings of PrivOn*, 2013.

[53] Florian Kerschbaum. Frequency-hiding order-preserving encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 656–667. ACM, 2015.

[54] Florian Kerschbaum and Axel Schröpfer. Optimal average-complexity ideal-security order-preserving encryption. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 275–286. ACM, 2014.

[55] Ankur Khetrapal and Vinay Ganesh. HBase and Hypertable for large scale distributed storage systems. *Dept. of Computer Science, Purdue University*, pages 22–28, 2006.

[56] Xuejia Lai. *On the design and security of block ciphers*. PhD thesis, Diss. Techn. Wiss ETH Zürich, Nr. 9752, 1992. Ref.: JL Massey; Korref.: H. Bühlmann, 1992.

[57] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[58] Jin Li, Zheli Liu, Xiaofeng Chen, Fatos Xhafa, Xiao Tan, and Duncan S Wong. L-EncDB: a lightweight framework for privacy-preserving data queries in cloud computing. *Knowledge-Based Systems*, 79:18–26, 2015.

[59] Jinyuan Li, Maxwell N Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (sundr). In *OSDI*, volume 4, pages 9–9, 2004.

[60] Ming Li and Paul Vitányi. *An introduction to Kolmogorov complexity and its applications*. Springer Science & Business Media, 2013.

[61] Dongxi Liu and Shenlu Wang. Programmable order-preserving secure index for encrypted database query. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 502–509. IEEE, 2012.

[62] Zheli Liu, Xiaofeng Chen, Jun Yang, Chunfu Jia, and Ilsun You. New order preserving encryption model for outsourced databases in cloud environments. *Journal of Network and Computer Applications*, 2014.

[63] Tal Malkin, Isamu Teranishi, and Moti Yung. Order-preserving encryption secure beyond one-wayness. *IACR Cryptology ePrint Archive*, 2013:409, 2013.

[64] Charalampos Mavroforakis, Nathan Chenette, Adam O'Neill, George Kollios, and Ran Canetti. Modular order-preserving encryption, revisited. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 763–777. ACM, 2015.

[65] Peter Mell and Tim Grance. The NIST definition of cloud computing. 2011.

[66] Viet Hung Nguyen, Tran Khanh Dang, Nguyen Thanh Son, and Josef Kung. Query assurance verification for dynamic outsourced xml databases. In *Availability, Reliability and Security, 2007. ARES 2007. The Second International Conference on*, pages 689–696. IEEE, 2007.

[67] Tim O'Reilly. What is Web 2.0: Design patterns and business models for the next generation of software. *Communications and Strategies*, 65(1):17–37, 2007.

[68] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 223–238. Springer, 1999.

[69] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. Blind seer: A scalable private DBMS. In *2014 IEEE Symposium on Security and Privacy*, pages 359–374. IEEE, 2014.

[70] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: A strongly encrypted database system. *IACR Cryptology ePrint Archive*, 2016:591, 2016.

[71] Raluca Ada Popa, Frank H Li, and Nickolai Zeldovich. An ideal-security protocol for order-preserving encoding. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 463–477. IEEE, 2013.

[72] Raluca Ada Popa, Jacob R Lorch, David Molnar, Helen J Wang, and Li Zhuang. Enabling security in cloud storage slas with cloudproof. In *USENIX Annual Technical Conference*, volume 242, 2011.

[73] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100. ACM, 2011.

[74] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: processing queries on an encrypted database. *Communications of the ACM*, 55(9):103–111, 2012.

[75] Bart Preneel and Hongjun Wu. Cryptanalysis and design of stream ciphers. 2008.

[76] Vincent Rijmen and Joan Daemen. Advanced encryption standard. *Proceedings of Federal Information Processing Standards Publications, National Institute of Standards and Technology*, pages 19–22, 2001.

[77] R Rivest and A Shamir. Data Encryption Standard (DES). Federal Information Processing Standards Publications (FIPS PUBS) n^o 46-3, 1999.

[78] Ronald L Rivest, MJB Robshaw, Ray Sidney, and Yiqun Lisa Yin. The rc6tm block cipher. In *First Advanced Encryption Standard (AES) Conference*, 1998.

[79] Ronald L Rivest and Jacob CN Schuldt. Spritz - A spongy RC4-like stream cipher and hash function, 2014.

[80] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[81] Daniel Roche, Daniel Apon, Seung Geol Choi, and Arkady Yerukhimov. POPE: partial order-preserving encoding. Technical report, Cryptology ePrint Arch. 2015/1106, 2015.

[82] Giacinto Paolo Saggese, Antonino Mazzeo, Nicola Mazzocca, and Antonio GM Strollo. An FPGA-based performance analysis of the unrolling, tiling, and pipelining of the AES algorithm. In *International Conference on Field Programmable Logic and Applications*, pages 292–302. Springer, 2003.

[83] Scott M Sawyer, B David O'Gwynn, An Tran, and Tao Yu. Understanding query performance in Accumulo. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6. IEEE, 2013.

[84] Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (Blowfish). In *Fast Software Encryption*, pages 191–204. Springer, 1994.

[85] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. Twofish: a 128-bit block cipher. *NIST AES Proposal*, 15, 1998.

[86] Saeed Sedghi, Peter Van Liesdonk, Jeroen M Doumen, Pieter H Hartel, and Willem Jonker. Adaptively secure computationally efficient searchable symmetric encryption. 2009.

[87] Faisal Shahzad, Waheed Iqbal, and Fawaz 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)*, pages 120–125. IEEE, 2015.

[88] Nigel P Smart and Frederik Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *International Workshop on Public Key Cryptography*, pages 420–443. Springer, 2010.

[89] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*, pages 44–55. IEEE, 2000.

[90] Wenhai Sun, Bing Wang, Ning Cao, Ming Li, Wenjing Lou, Y Thomas Hou, and Hui Li. Privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking. In *Proceedings of the 8th ACM SIGSAC symposium on Information, Computer and Communications Security*, pages 71–82. ACM, 2013.

[91] Sai Deep Tetali, Mohsen Lesani, Rupak Majumdar, and Todd Millstein. Mrcrypt: Static analysis for secure cloud computations. *ACM SIGPLAN Notices*, 48(10):271–286, 2013.

[92] Jawahar Thakur and Nagesh Kumar. DES, AES and Blowfish: Symmetric key cryptography algorithms simulation based performance analysis. *International journal of emerging technology and advanced engineering*, 1(2):6–12, 2011.

[93] Brian Thompson, Stuart Haber, William G Horne, Tomas Sander, and Danfeng Yao. Privacy-preserving computation and verification of aggregate queries on outsourced databases. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 185–201. Springer, 2009.

[94] John J Trinckes Jr. *The Definitive Guide to Complying with the HIPAA/HITECH Privacy and Security Rules.* CRC Press, 2012.

[95] Stephen Tu, M Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. In *Proceedings of the VLDB Endowment*, volume 6, pages 289–300. VLDB Endowment, 2013.

[96] Bogdan George Tudorica and Cristian Bucur. A comparison between several NoSQL databases with comments and notes. In *Roedunet International Conference (RoEduNet), 2011 10th*, pages 1–5. IEEE, 2011.

[97] Peter Van Liesdonk, Saeed Sedghi, Jeroen Doumen, Pieter Hartel, and Willem Jonker. Computationally efficient searchable symmetric encryption. In *Workshop on Secure Data Management*, pages 87–100. Springer, 2010.

[98] Bing Wang, Shucheng Yu, Wenjing Lou, and Y Thomas Hou. Privacy-preserving multi-keyword fuzzy search over encrypted data in the cloud. In *INFOCOM, 2014 Proceedings IEEE*, pages 2112–2120. IEEE, 2014.

[99] Bill Wilder. *Cloud architecture patterns: using Microsoft Azure.* O'Reilly Media, Inc., 2012.

[100] Sander Wozniak, Michael Rossberg, Sascha Grau, Ali Alshawish, and Guenter Schaefer. Beyond the ideal object: towards disclosure-resilient order-preserving encryption schemes. In *Proceedings of the 2013 ACM workshop on cloud computing*, pages 89–100. ACM, 2013.

[101] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 162–167. IEEE, 1986.

[102] Xingliang Yuan, Xinyu Wang, Cong Wang, Chen Qian, and Jianxiong Lin. Building an encrypted, distributed, and searchable key-value store. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 547–558. ACM, 2016.

Tim Waage

Curriculum Vitae

Personal Data

Date of birth	11/06/1983
Nationality	German
Civil status	married

Professional Experience

Georg-August-University Göttingen

since 2014 **Research assistant / PhD student**, *Institute of Computer Science, Researchgroup Knowlegde Engineering*.
- PhD Topic "Property-preserving encryption in NoSQL Wide Column Stores"
- Supervision of lectures and seminars, partly in cooperation with the Otto-Friedrich-University, Bamberg
- Techincal conception and coordination of workshops
- Expertise/Research interests: Security in NoSQL-Databases, cloud computing platforms

2010 **Student assistant**, *Kooperationsstelle Hochschulen und Gewerkschaften*.
- Conception and initial maintenance of a project-related website

2009 - 2010 **Student assistant**, *Department of Developmental Psychology*.
- Conception and programming of a PHP/MySQL database for gathering personal data of clinical trials
- Technical supervision of an audio/video setup with multiple cameras and microphones to conduct studies with infants
- Website maintenance (GCMS)
- Administration of active-directory-based computers

2006 – 2010 **Student assistant**, *Institute of Computer Science*.
- Assistant for nine lectures, incl. weekly seminars and conception of lecture accompanying content
- Correction of homework and exams
- Teaching support in block courses

2008 **Student assistant**, *Department of Economic and Social Psychology*.
- Programming of applications for computer-based tests
- Maintenance of the department website and other project-related websites

University Medical Center Göttingen (UMG)

2011 – 2014 **IT-Coordinator**, *Department of Nephrology and Rheumatology*.
- Supervision of department-specific software and medical equipment
- Coordination of department-specific needs and UMG/external service providers
- Administration of active-directory-based computers
- Daily IT-support
- Conception and maintencance of several websites

Malamut Team Catalyst GmbH

2010 – 2012 **Webdeveloper**, Göttingen.
- Realization of an online testcenter for determining social competences in work life

Miscellaneous

since 2010 **Freelance photographer**.
- Weddings, Product-/Eventphotography, Travel-/Portraitphotography

Education

2007 – 2010 **Master of Science**, *Georg-August-University, Institute of Computer Science*, Göttingen.
- Thesis Topic: Implementation of a Tool for Determining the Nominal Capacity of Wireless Mesh Networks
- Advisor: Prof. Dr. Dieter Hogrefe
- Thesis Grade: 1.0
- Degree Grade: 1.3 **(graduated with honors)**

2004 – 2007 **Bachelor of Science**, *Georg-August-University, Institute of Computer Science in cooperation with the Faculty of Law*, Göttingen.
- Thesis Topic: Softwarepatente in Deutschland, Europa und den Vereinigten Staaten von Amerika
- Advisor: Prof. Dr. Gerald Spindler
- Thesis Grade: 1.7
- Degree Grade: 1.5, Specialization: Law of Informatics

Conference Publications

2017
- Christian Göge, **Tim Waage**, Daniel Homann, Lena Wiese. Improving Fuzzy Searchable Encryptionwith Direct Bigram Embedding (TRUSTBUS 2017), to appear in the conference proceedings in August 2017

2016
- **Tim Waage**, Daniel Homann, Lena Wiese. Practical Application of Order-Preserving Encryption in Wide Column Stores (SECRYPT2016), Proceedings of the 13th International Joint Conference on e-Business and Telecommunications, pages 352-359, Scitepress, 2016.
- **Tim Waage**, Lena Wiese. Ordnungserhaltende Verschlüsselung in Cloud-Datenbanken (DACH Security 2016), Proceedings of DACH2016, pages 75-86, syssec, 2016.
- **Tim Waage**. Order Preserving Encryption for Wide Column Stores (SICHERHEIT2016), GI Lecture Notes in Informatics volume 256, pages 209-216, Köllen Druck+Verlag, 2016

2015
- Lena Wiese and **Tim Waage**. Benutzerfreundliche Verschlüsselung für Cloud-Datenbanken (DACH Security 2015), IT Security & IT Management, pages 12-23. syssec, 2015.
- **Tim Waage**. Durchsuchbare Verschlüsselung in NoSQL Datenbanken (INFORMATIK2015), Lecture Notes in Informatics volume 246, pages 1747-1758, Bonner Köllen Verlag, 2015
- **Tim Waage**, Ramaninder Singh Jhajj, Lena Wiese. Searchable Encryption in Apache Cassandra. In Foundations and Practice of Security - 8th International Symposium (FPS2015), Lecture Notes in Computer Science volume 9482. Springer, 2015

2014
- **Tim Waage** and Lena Wiese. Benchmarking encrypted data storage in HBase and Cassandra with YCSB. In Foundations and Practice of Security - 7th International Symposium (FPS2014), Lecture Notes in Computer Science volume 8930, pages 311-325. Springer, 2014.

2010
- Roman Seibel, Nils-Hendrik Klann, **Tim Waage**, Dieter Hogrefe. Wireless Mesh Networks for Infrastructure Deficient Areas (WCC2010). Communications: Wireless in Developing Countries and Networks of the Future volume 327, pages 26-38. Springer Berlin Heidelberg, 2010

Other Publications

2016
- **Tim Waage**, Lena Wiese. Implementierung von kryptographischen Sicherheitsverfahren für Apache Cassandra und Apache HBase, HMD Praxis in der Wirtschaftsinformatik 53.4, pages 499-513, Springer, 2016
- Lena Wiese, **Tim Waage**. A Fragmentation and Replication Scheme for Flexible Query Answering, The Computer Journal 60.3, pages 308-321, Oxford University Press, 2016.

Research Visits

06/2014	Institute Mihajlo Pupin, Belgrade, Serbia
12/2014	Faculty of Mathematics and Computer Science, Charles University, Prague, Czech Republic

Peer Review Service

2016	DPM
2015	FPS, DPM

Teaching Experience

WS14/15	NoSQL Databases Execise
WS14/15	Intelligent Information Systems Seminar
SS15	NoSQL Databases Execise
WS15/16	NoSQL Databases Execise
WS15/16	Intelligent Information Systems Seminar
SS16	Advanced Data Management Practical Course
WS16/17	Anwendungsentwicklung mit NoSQL Datenbanken Praktikum

Student exchange

2009	**Study abroad**, *University of Wollongong*, New South Wales, Australia.

Qualifications

2016	IREB Certified Professional for Requirements Engineering Foundation Level
2017	ISTQB Certified Tester Foundation Level

Languages

German	native speaker
English	fluent
Russian, Latin	basic knowledge

Göttingen, 19/05/2017 [illegible]