Privacy By Encrypted Databases

Patrick Grofig, Isabelle Hang, Martin Härterich, Florian Kerschbaum, Mathias Kohler, Andreas Schaad, Axel Schröpfer, Walter Tighzert

SAP
Karlsruhe, Germany
firstname.lastname@sap.com

Abstract. There are a few reliable privacy mechanisms for cloud applications. Data usually needs to be decrypted in order to be processed by the cloud service provider. In this paper we explore how an encrypted database can (technically) ensure privacy. We study the use case of a mobile personalized healthcare app. We show that an encrypted database can ensure data protection against a cloud service provider. Furthermore we show that if privacy is considered in application design, higher protection levels can be achieved, although encrypted database are a transparent privacy and security mechanism.

Keywords: Cryptography, Encrypted Databases, Healthcare, Privacy by Design

1 Introduction

Following the "privacy by design" principle privacy decisions should be taken into account when designing applications. This implies using the available means, such as anonymization, encryption and access control, to implement protection of necessary and minimized personal identifiable data. An unsolved challenge arises in the currently prevalent design of cloud or hybrid cloud, e.g. mobile and cloud, applications. In order to process personal identifiable data in the cloud, it needs to be in the clear. This availability of the cleartext enables all kinds of misuse by the cloud service provider, originally intended only to provide IT services.

This problem of trust in the cloud service provider has generated a public debate. It is not clear whether current data protection regulations allow storing personal identifiable data of European Union citizens on cloud servers not hosted in the European Union. In this paper we explore the use of a technical means in order to securely store (and process) data in the cloud. In particular, we investigate the use of an encrypted database. In our database, all data is encrypted at the client which also retains the key. Therefore the cloud service provider has only access to ciphertexts.

For all fairness, the legal debate whether encrypted personal identifiable data (without the key) is still personal identifiable data has also not been settled. In this paper we will nevertheless not explore the legal dimension. For a technically

educated person, it seems convincing that encryption solves the problem. In this paper we will show that when using a transparent encryption mechanism such as an encrypted database precautions also in the design of the application should be taken.

We use the example of a mobile personalized health care app that stores data in the cloud. Clearly, personal health records are sensitive personal identifiable data and subject to the strongest data protection regulations. We will show how to design such an app, so that it ensures confidentiality of this data against a cloud service provider using an encrypted database.

The remainder of the paper is structured as follows. In the next Section we describe related work and how we implement our encrypted database using it. In Section 3 we describe how this encrypted data works and is used. Then, in Section 4 we describe our use case of a mobile personalized healthcare app. And, finally, in Section 5 we present our conclusions for ensuring privacy by an encrypted database.

2 Processing Queries on Encrypted Data

We briefly describe how we implement our encrypted database using adjustable encryption. Security is a major concern for outsourced databases [1, 8, 13]. In the database-as-a-service model [13] an independent service provider offers its database to clients. The clients need to entrust their data to the cloud service provider without having control over unwanted disclosures, e.g., to insiders or hackers.

The solution to this outsourced security problem is to encrypt data before sending it to the cloud. Of course, the decryption key needs to remain only at the client. This is easy to implement for simple storage, but the clients must remain able to query the database. Therefore the service provider has to solve the complicated task of querying on the encrypted data. There are several proposals for processing many SQL queries on this encrypted data [2, 11, 12, 29, 31]. We implement a slightly modified version of Popa et al.'s adjustable encryption [29].

Order-preserving encryption (OPE) [2, 4, 5, 28], deterministic encryption (DET) [3, 27] and (additively) homomorphic encryption (HOM) [25] offer a (partial) solution to the encrypted data querying problem. These different encryption schemes have different algebraic properties. Let $c = E_T(x)$ denote the encryption of plaintext x in encryption type $T \in \{OPE, DET, HOM\}$. We denote $D_T(c)$ the corresponding decryption. Order-preserving encryption has the property that it preserves the order of plaintexts, i.e.

$$x \leq y \Longleftrightarrow E_{OPE}(x) \leq E_{OPE}(y)$$

Deterministic encryption preserves the equality of plaintexts, i.e.

$$x = y \Longleftrightarrow E_{DET}(x) = E_{DET}(y)$$

In (additively) homomorphic encryption multiplication of ciphertexts (modulo a key-dependent constant) maps to addition of the plaintexts, i.e.

$$D_{HOM}(E_{HOM}(x) \cdot E_{HOM}(y)) = x + y$$

Using these algebraic properties we can implement the relational operators for most SQL queries on the ciphertext, i.e. without decrypting the data. Consider a table scan with equality or range selection conditions. One can implement these on order-preservingly or even deterministically encrypted data. Similarly, one can implement join operators using these conditions – assuming the columns use the same key. Also, grouping (group by clause) can operate on deterministically encrypted data. Even, some data functions, such as minimum, maximum or counting, work on ciphertexts only. Note that for these operations it is not necessary to modify the relational operator implementation compared to a regular, non-encrypted database implementation. The operators perform the same computation on the ciphertexts as they would on the plaintexts[1].

For aggregation – sum or average functions – we can use homomorphic encryption. If the database multiplies the (selected) ciphertexts, it obtains a ciphertext of the aggregate. This requires only a small change to the operator implementation which one can implement by user-defined functions.

This leads to a convincing result: Using the appropriate encryption type a large subset of SQL queries can be implemented on encrypted data. Still, one has to choose the appropriate encryption type for one's data. This choice is important, because the encryption types have different security levels and may be incompatible.

Incompatible encryption makes executing a query impossible. For example, consider data encrypted using homomorphic encryption and performing a range query on it. This is impossible. Such combinations of encryption schemes may be even required by specific queries. Consider the query

```
SELECT x FROM T GROUP BY y HAVING SUM(z) > 100.
```

The sum function requires homomorphic encryption and the greater-than comparison requires order-preserving encryption. Such queries simply cannot be executed on encrypted data in the server's database, since no appropriate encryption scheme exists.

The different encryption types also have different security levels. Homomorphic encryption, such as Paillier's encryption scheme [25], is semantically secure. Semantic security means that it is computationally impossible to distinguish two ciphertexts, even if the adversary may choose their plaintexts. Semantic security implies that ciphertexts are randomized, i.e., equality is not preserved under encryption.

Deterministic encryption leaks this equality and is therefore considered less secure. Security guarantees have been established under the assumption that the

[1] Although the data type of the ciphertexts may be different.

plaintexts have high entropy [3]. Order-preserving encryption is not only deterministic, but also leaks the order of the plaintexts (and is therefore less secure). The security of order-preserving encryption is still under debate. Boldyreva et al. proposed an order-preserving encryption that has the best security possible assuming it is non-modifiable and stateless [4]. They later showed that it leaks the upper half of the bits of the plaintext [5]. Popa et al. proposed a modifiable, stateful scheme that has ideal-security, i.e., it leaks only the order [28].

In summary we have: homomorphic (or standard) encryption is more secure than deterministic encryption which is more secure than order-preserving encryption. This observation implies that the client should carefully choose its encryption types for data outsourcing. It should only use order-preserving or deterministic encryption if necessary to enable its queries in order to achieve the highest security level. Yet, the set of executed queries may be unknown at design time making this choice undecidable.

Popa et al. offer an intriguing solution to the encryption type selection problem [29]. First, they introduce a further encryption type RND for standard, randomized encryption. This encryption type only allows retrieval, but no queries. Note that order-preserving encryption enables a proper superset of queries to deterministic encryption. They therefore compose a layered ciphertext called onion. For each data item x they compute the following sequence of encryptions

$$E_{RND}(E_{DET}(E_{OPE}(x)))$$

This onion at first only allows retrieval – due to the randomized encryption. When the client encounters a query that requires deterministic encryption, e.g., a selection using equality, then it updates the database. It sends the key $D_{RND}()$ for decrypting the randomized encryption to the database. The database uses a user-defined function to perform the update, such that now the database stores $E_{DET}(E_{OPE}(x))$. This enables the new query to be executed. The same procedure occurs in case of a query that requires order-preserving encryption to execute.

Homomorphic encryption is handled slightly differently and stored in a separate column. The separate column also enables aggregation operations, but does not harm security, since homomorphic encryption is semantically secure. A layering is not possible, since homomorphic encryption needs to encrypt the plaintext x for the correct result in aggregations.

This algorithm represents an adjustment mechanism of the database to the series of executed queries. It enables to dynamically adjust the encryption types, i.e., without knowing all queries in advance. We call such a database *adjustably encrypted*. Furthermore, the adjustment is unidirectional. Once decrypted to deterministic or order-preserving encryption, it is never necessary to return to a higher encryption level to enable a subsequent query. Yet, security against the cloud service provider has already been weakened, because the less secure ciphertext has been revealed at least once and can therefore be used in cryptanalysis.

3 Database Adjustment

As described before the database encryption level is adjusted to the queries performed. In this section we further deepen into the SQL features and their corresponding adjustment. We then also present how an application can use our encrypted database. It is particularly important that its use is transparent to the application, i.e. an application developer usually does not have to worry about the encryption. In this paper we argue, however, that better privacy results can be obtained, if the developer carefully designs his application and queries.

We consider the database operations of selection, grouping, joins, sorting, aggregation and further functions. For each operation we identify the corresponding encryption level. The adjustment algorithm decrypts the database to this level (or below). We do not explain the semantics of the operations, but refer the reader to an SQL introduction.

Selection: There are two types of selection criteria: equality and range query. An example of an equality selection is

```
SELECT x FROM T WHERE y = 100
```

After this query the column x is either encrypted using deterministic or order-preserving encryption. An example of a range selection is

```
SELECT x FROM T WHERE y > 100
```

In this case the column x is afterwards encrypted using order-preserving encryption.

Grouping: Grouping is a common operation particularly in analytical queries. It combines groups of row with a common, equal attribute(s) value(s). An (relatively stupid) example of grouping is

```
SELECT x FROM T GROUP BY x
```

Afterwards, the column x is encrypted using deterministic encryption. Note that grouping can be combined with selection using the `HAVING` clause. An example leading to order-preserving encryption would be

```
SELECT x FROM T GROUP BY x HAVING x > 100
```

Joins: Joins can be algebraically represented by cross-products, i.e. the all-pairs combination of the relations. This implies that joins per se are neutral for the encryption. A cross-product can always be built, even on randomized encryption. An example is

```
SELECT x, y FROM T1, T2
```

Still, cross-products significantly extend the size of the results table and joins are usually combined with a selection. Using the selection criterion, joins can be implemented using significantly faster algorithms, e.g. hash joins or sort-and-merge joins. The selection criterion can be again equality or range as before and has the same impact on the encryption level. In the following example column z is encrypted using deterministic encryption.

```
SELECT x, y FROM T1, T2 WHERE T1.z = T2.z
```

Note that for cross-column selection conditions both columns need to be encrypted using the same key. We adjust the encryption key before the query as well. Proxy re-encryption offers to change the encryption key without intermediate decryption. We follow the strategy outlined by Kerschbaum et al. [20].

Sorting: Values can be returned sorted from a query. An example is

```
SELECT x FROM T ORDER BY x
```

Afterwards, the column x is encrypted using order-preserving encryption. Also rank-based statistical function such as maximum (`MAX`) and minimum (`MIN`) are based on order-preserving encryption.

```
SELECT MIN(x), MAX(x) FROM T
```

Aggregation: We consider three aggregation functions: `SUM`, `AVG`, and `COUNT`.

```
SELECT SUM(x), AVG(x), COUNT(x) FROM T
```

Summation is performed using homomorphic encryption. This requires the operator to be modified. This can be implemented using a user-defined function or by modifying the database. A decryption to an onion level is not necessary, since homomorphic encryption is stored in parallel to the leveled onion. The mean is computed as `SUM(x)/COUNT(x)` where the division is performed on the client. Summation is implemented as before and counting can be implemented using any encryption. The count operator can perform on any encryption level, again even randomized encryption. Note that the count operator requires to implement `NULL` (in addition to 0) values. While this has been criticized to lead to incomplete logic, it is commonly supported in off-the-shelf databases.

Functions: SQL offers a wide variety of further functions, e.g. for string manipulation. We do not provide specific support for these, but implement them on the client on the decrypted cleartexts.

We have shown that there is a wide spectrum of SQL functions and corresponding encryption levels. Some operations, such as equality selection, even operate on multiple encryption levels. We therefore have carefully select the encryption level (and onion) to operate on when there are multiple options and intertwined conditions from the query. The problem is complicated, if the user can configure the available encryption options. We follow again the algorithm by Kerschbaum et al. [21].

An obvious question to ask is whether adjustable encryption actually provides more security. Given an infinitely long sequence of random queries, one would expect all columns to be decrypted to order-preserving encryption. Fortunately, real sequences are not infinite. We have performed a number of experiments in order to study the security provided by adjustable encryption. We executed the sequences of TPC-H, TPC-C and of a live SAP system on our adjustable encryption scheme. The TPC-H benchmark simulates analytical queries whereas

Table 1. Encryption State of Exemplary System

	TPC-H	TPC-C	Live SAP System
Total Queries	22	20	406
Total Tables	8	9	2
Total Columns	61	71	248
RND (columns / %)	17 / 27.9%	49 / 69,0%	157 / 63.3%
DET (columns / %)	24 / 39.3%	17 / 23,9%	74 / 29.8%
OPE (columns / %)	20 / 32.8%	5 / 7,0%	17 / 7.9%

the TPC-C benchmark and the SAP system perform transactional workloads. We have summarized our findings in Table 1.

In order to judge the benefit of encrypted databases for privacy it is important to understand their use. An application developer implementing its application on top of an adjustably encrypted database needs to make few modifications. All he needs is the driver for the encrypted database. In our case this is a standard JDBC driver for JAVA applications. He can then use the encrypted database similarly to any other, non-encrypted database, i.e. he executes SQL queries and retrieves the results. The driver interface is unchanged and all encryption and decryption operation occur inside the driver. The driver only needs to be given a reference to the database instance (and key store if it is not the default). Figure 1 shows some example Java code for using our database.

```
Class.forName("com.sap.research.seeed.jdbc.SEEEDDriver");

Properties props = new Properties();
props.put(SEEEDDriver.extDriverURLPropKey, ServerInformation.HANA_Uri);

Connection conn = DriverManager.getConnection(
                      "jdbc:sap-research:seeed", props);

Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM \"MyTable\"");
```

Fig. 1. Java code for using encrypted database

The encryption of the database is *transparent* to the application developer. He does not need to make any major modifications when operating on an encrypted database. On the hand, this is a significant advantage, since integration is easy and privacy and security benefit. On the other hand, this may lead the application developer to unsafe choices and in this paper we highlight that the application developer can further enhance privacy when paying attention to his design.

Secure outsourcing could also be solved by other techniques, such as secure multi-party computation [10, 15, 18, 19, 30] which has been used for supply chain management [7, 17, 22, 23, 26] and statistical benchmarking [6, 16, 24], but this requires a more significant redesign of the application and the user and developer experience.

4 Personalized Healthcare App

4.1 Basic Functions

Next, we briefly describe the design of our healthcare app. Personalized healthcare is a current trend enabled by ubiquitous availability of computing devices, such as smartphones. These can monitor health indicators, such as blood pressure, blood sugar, heart rate, temperature, etc., or simply record mood and feelings. The collected set of measurements can be used in treatment of illnesses. It can even be used in order to identify preventive treatment when early symptoms arise.

Each measurement is a tuple. Let *id* be the identifier of the person. Let *time* be the time of measurement. Let *loc* be the location of the measurement. Let *val* be the data value of the location (with an implicit type). The tuple then consists of

$$\langle id, time, loc, val \rangle$$

This tuple also represent the start of a schema for our personalized healthcare app. All basic data, i.e. measurements, are collected in this form and stored in relations. We can simplify the tuple by removing the identifier and using specialized forms of multi-tenancy. Each person using its app has its own user identifier in the database and can share the same schema. Since user need not and should not share data, this form of multi-tenancy simplifies the tuple to the relation

$$\langle time, loc, val \rangle$$

Our healthcare app has a very simple user interface for entering such tuple, if they are not collected automatically. A person incrementally builds a record of measurements. Let *PlainData* be the name of the table. Each time the app performs the exemplary following query

```
(I) INSERT time = '16.12.2013 23:59', loc = 'Karlsruhe',
          val = '120' INTO PlainData
```

This collection of data would be mostly used, if it were not used for some analysis. A typical form of analysis are histograms, i.e. we group the data by categories, e.g. time slots or locations, and count the number of occurrence, e.g. of unhealthy events. From such a histogram one can, for example, see the time of the day with the highest blood pressure. Our app offers a simple user interface for

retrieving the most common and useful such histograms. Each time it performs the following exemplary query

(II) `SELECT loc, COUNT(loc)`
`FROM PlainData GROUP BY loc`
`WHERE val > 100 ORDER BY COUNT(loc)`

4.2 Integration with Encrypted Database

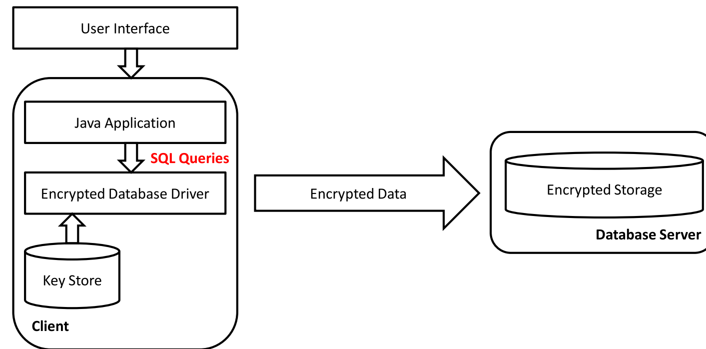

Fig. 2. Basic app architecture

We propose to use an adjustably encrypted database as described in Section 2 in order to protect the data against the service provider. Figure 2 shows the basic architecture of our app with our encrypted database. We highlight in red the interface between application code and database driver, namely SQL queries. As we have already described in detail in Section 3 the encryption state of the database adjusts to the queries performs. So, we need to investigate how this adjustment is done for our personalized healthcare app.

Initially the database starts with all onions encrypted using randomized encryption, i.e. each column is encrypted using a semantically secure encryption scheme, such as AES128-CBC. When performing query I (insert of plain values), each value will be encrypted to this onion level. As a result all personal identifiable data is encrypted using a standard, commonly recommended encryption scheme.

When performing query II (computation of histograms), several selection criteria are given. First all data is grouped by its location (or time slot). Hence, the location column is decrypted to deterministic encryption. Furthermore, all values greater than 100 are selected (because they likely indicate an unhealthy measurement). Hence, the value column is decrypted to order-preserving encryption.

If we assume that also histograms for time slots are computed, all columns will be decrypted: time and value to order-preserving encryption and location to

deterministic. As Islam et al. point out, already searchable encryption which is still stronger than deterministic encryption allows frequency analysis when side information is available [14]. In our example it would probably be easy to determine the plaintexts of most ciphertexts if someone trailed a person for some time. The observer obtains a sample of locations which is likely similarly distributed to the ones stored in the database. The security of order-preserving encryption is less investigated, but even more debated. As a result our healthcare while using an encrypted database would offer little protection of personal identifiable data.

4.3 Better Solution

Fortunately, there is a much better solution if the app designer pays a little attention. Instead of computing the histograms on the fly, the designer can pre-select the most important ones. He does that anyway, since few app users are skilled enough to select a sensible analysis by themselves. The designer thereby provides an additional service of offering the most useful analyses to his customers.

As an example, we consider the histogram query from above. Instead of executing query II the app can incrementally compute the statistics. The can simply store the aggregations in the following relation

$$\langle time, loc, count \rangle$$

Let $AggregateData$ be the name of the corresponding database table It then inserts new values as follows

```
(III) UPDATE count = (SELECT count
WHERE time = '16.12.2013 23:59' AND loc = 'Karlsruhe') + 1
INTO AggregateData
WHERE time = '16.12.2013 23:59' AND loc = 'Karlsruhe'
```

In order to retrieve a histogram it simply executes

```
(IV) SELECT * from FROM AggregateData
```

Note that sorting of query IV can be performed locally without any additional cost, since all data values need to be transferred anyway. Yet, if we consider the encryption state in our encrypted database, we see a significant improvement. Query (III) performs selection with equality conditions for the time and location column. Therefore these columns also need to be encrypted using deterministic encryption. Nevertheless, there is a significant improvement in security. Each ciphertext is unique. Hence, frequency analysis attacks as by Islam et al. are thwarted.

This is similarly to the index built for searchable encryption by Curtmola et al. [9]. They first encrypt each keyword using deterministic encryption and then build an encrypted list of documents. This significantly speeds up search time which is also useful in our application, but they also show that it maintains

security. We omit a formal analysis here, but it is easy to see that all ciphertexts can be simulated by random numbers if the key is unknown.

The count column is encrypted using homomorphic encryption. Homomorphic encryption, such as Paillier's [25] is also semantically secure. Using homomorphic encryption query III can be performed entirely on the database, i.e. without performing the sub-select and addition locally first.

Clearly, this design improves the privacy of the app user and may even allow storing this personal identifiable data in the cloud. All, data is encrypted using provably secure encryption schemes and known attacks are avoided. Still, the mobile healthcare app is still able to function as before. We have achieve a major step forward in securing privacy in a hybrid cloud application as it is prevalent in current software development.

4.4 Sharing The Data

Sometimes it is desirable to share one's data. Such sharing should, of course, only be initiated on the user's request and with his informed consent. Unintentional sharing is likely a privacy violation.

We consider two types of sharing for our mobile healthcare app: among different systems or application and with third parties. A user may operate different system, e.g. a smartphone, a tablet and a PC. He may want to use or analyze his data on all three systems potentially using different applications. Note that in this case the person accessing the data remains the data owner, but he is accessing it using different systems.

In the other case the data owner may want to share his data with third parties, e.g. his physician or a medical expert. In this case the data is accessible by this third party and a potential privacy violation may occur. We again do not investigate the legal obligations for such sharing – for the data owner or the recipient. Instead, we focus on the implementation of appropriate cryptographic security mechanisms.

4.5 Sharing Across Systems

In this case the access rights remain the same, i.e. the data owner is allowed to access its data, but nobody else. Therefore no additional cryptographic keys are necessary, as long as the data owner remains in sole possession of the key. Note that this also ensures technical protection against the cloud service provider.

Often it is not possible or too cumbersome to directly link two systems, such as smartphone and tablet. Instead each device is connected to the Internet sharing common access to servers, such as the cloud server of the health care application. Fortunately, there is a standardized way of securely transferring data between the systems using this server.

Note that the only information (state) that needs to be shared are the cryptographic keys. All other information is stored encrypted in the cloud server's database. Cryptographic keys can be securely stored in a password protected

file. The password is used to generate a cryptographic key which encrypts the other cryptographic keys. This is a common mechanism for implementing secure key stores.

The design idea for a shared tablet, mobile app application is now to store this key store also in the cloud. At the start of the application, it loads the key store from the cloud. Then, the user enters a password which is used to decrypt the keys for the encrypted database. All systems can securely use the keys in this manner. There is no additional effort for the user, since he needs to authenticate using a password in any case.

4.6 Sharing With Third Parties

In this case an additional party gets access and there is need for an additional key. Not all data may be shared eternally and hence the user should not share his key. Instead, he should make a copy and make this copy available to the third party.

A simple solution for this data sharing problem is using public keys. The user downloads the data he wants to share to his system and encrypts it with the public key of the recipient. For example, for e-mail messages there is the quasi-standard of PGP.

A public-key encrypted message can only be read by the recipient. Of course, the key exchange needs to be secured, but the availability of a public key infrastructure can be assumed. It is likely that a cloud service provider for health data offers its customers a connection secured by the transport layer security (TLS – formerly SSL) protocol. This protocol already requires the server to obtain a certificate from an authority part of a public key infrastructure.

5 Conclusions

We have shown how an adjustably encrypted database operates and how to use it. Based on this we have done a case study for a mobile personalized healthcare app. This app uses our encrypted database and we have shown its basic functions. We have first applied a straight-forward approach to the SQL queries and showed its impact on the encryption state of the database. Then, we have further improved the protection by modifying the queries. Ultimately, we reached a state where only provably encrypted data is revealed. We therefore conclude that with proper use an encrypted database may protect personal identifiable information in the cloud against the service provider. It is therefore a technical alternative to organizational measures, such as storing personal identifiable information of European Union citizens only on servers hosted in the European Union. Hence, we further conclude that all cloud or hybrid applications designed with privacy in mind should consider the use of an encrypted database. It provides technical data protection even when combined with organizational measures. We envision such transparent database protection to become a standard measure for technical privacy protection in the cloud economy.

References

1. Divyakant Agrawal, Amr El Abbadi, Fatih Emekçi, and Ahmed Metwally. Database management as a service: challenges and opportunities. In *Proceedings of the 25th International Conference on Data Engineering*, ICDE, 2009.
2. Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM International Conference on Management of Data*, SIGMOD, 2004.
3. Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. Deterministic and efficiently searchable encryption. In *Advances in Cryptology*, CRYPTO, 2007.
4. Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. Order-preserving symmetric encryption. In *Proceedings of the 28th International Conference on Advances in Cryptology*, EUROCRYPT, 2009.
5. Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. Order-preserving encryption revisited: improved security analysis and alternative solutions. In *Proceedings of the 31st International Conference on Advances in Cryptology*, CRYPTO, 2011.
6. Octavian Catrina and Florian Kerschbaum. Fostering the uptake of secure multiparty computation in e-commerce. In *Proceedings of the 3rd International Conference on Availability, Reliability and Security.*, ARES, 2008.
7. Leonardo Weiss Ferreira Chaves and Florian Kerschbaum. Industrial privacy in rfid-based batch recalls. In *Proceedings of the International Workshop on Security and Privacy in Enterprise Computing*, INSPEC, 2008.
8. Carlo Curino, Evan P. C. Jones, Raluca Ada Popa, Nirmesh Malviya, Eugene Wu, Sam Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational cloud: a database-as-a-service for the cloud. In *Proceedings of the 5th Conference on Innovative Data Systems Research*, CIDR, 2011.
9. Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. *Journal of Computer Security*, 19(5), 2011.
10. Jannik Dreier and Florian Kerschbaum. Practical privacy-preserving multiparty linear programming based on problem transformation. In *Proceedings of the 3rd IEEE International conference on Privacy, security, risk and trust*, PASSAT, 2011.
11. Hakan Hacigümüs, Balakrishna Iyer, and Sharad Mehrotra. Efficient execution of aggregation queries over encrypted relational databases. In *Proceedings of the 9th International Conference on Database Systems for Advances Applications*, DASFAA, 2004.
12. Hakan Hacigümüs, Balakrishna R. Iyer, Chen Li, and Sharad Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *Proceedings of the 2002 ACM International Conference on Management of Data*, SIGMOD, 2002.
13. Hakan Hacigümüs, Sharad Mehrotra, and Balakrishna R. Iyer. Providing database as a service. In *Proceedings of the 18th International Conference on Data Engineering*, ICDE, 2002.
14. Mohammad Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In *Proceedings of the 19th Network and Distributed System Security Symposium*, NDSS, 2012.
15. Florian Kerschbaum. Simple cross-site attack prevention. In *Proceedings of the 3rd International Conference on Security and Privacy in Communications Networks.*, SECURECOMM, 2007.

16. Florian Kerschbaum. Building a privacy-preserving benchmarking enterprise system. *Enterprise Information Systems*, 2(4), 2008.
17. Florian Kerschbaum. An access control model for mobile physical objects. In *Proceedings of the 15th ACM symposium on Access control models and technologies*, SACMAT, 2010.
18. Florian Kerschbaum. Automatically optimizing secure computation. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS, 2011.
19. Florian Kerschbaum, Debmalya Biswas, and Sebastiaan de Hoogh. Performance comparison of secure comparison protocols. In *Proceedings of the International Workshop on Business Processes Security*, WSBPS, 2009.
20. Florian Kerschbaum, Martin Härterich, Patrick Grofig, Mathias Kohler, Andreas Schaad, Axel Schröpfer, and Walter Tighzert. Optimal re-encryption strategy for joins in encrypted databases. In *Proceedings of the 27th IFIP Conference on Data and Applications Security*, DBSEC, 2013.
21. Florian Kerschbaum, Martin Härterich, Mathias Kohler, , Schaad Andreas Hang, Isabelle, Axel Schröpfer, and Walter Tighzert. An encrypted in-memory column-store: the onion selection problem. In *Proceedings of the 9th International Conference on Information Systems Security*, ICISS, 2013.
22. Florian Kerschbaum and Nina Oertel. Privacy-preserving pattern matching for anomaly detection in rfid anti-counterfeiting. In *Proceedings of the 6th Workshop on RFID Security*, RFIDSEC. 2010.
23. Florian Kerschbaum and Alessandro Sorniotti. Rfid-based supply chain partner authentication and key agreement. In *Proceedings of the 2nd ACM conference on Wireless network security*, WISEC, 2009.
24. Florian Kerschbaum and Orestis Terzidis. Filtering for private collaborative benchmarking. In *Proceedings of the International Conference on Emerging Trends in Information and Communication Security*, ETRICS. 2006.
25. Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of the 18th International Conference on Advances in Cryptology*, EUROCRYPT, 1999.
26. Richard Pibernik, Yingying Zhang, Florian Kerschbaum, and Axel Schröpfer. Secure collaborative supply chain planning and inverse optimization–the jels model. *European Journal of Operational Research*, 208(1), 2011.
27. Stephen C. Pohlig and Martin E. Hellman. An improved algorithm for computing logarithms over gf(p) and its cryptographic significance. *IEEE Transactions on Information Theory*, 24(1):106–110, 1978.
28. Raluca Ada Popa, Frank H. Li, and Nickolai Zeldovich. An ideal-security protocol for order-preserving encoding. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, S&P, 2013.
29. Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP, 2011.
30. Axel Schröpfer, Florian Kerschbaum, and Günter Müller. L1 – an intermediate language for mixed-protocol secure computation. In *Proceedings of the 35th IEEE Computer Software and Applications Conference*, COMPSAC, 2011.
31. Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. In *Proceedings of the 39th International Conference on Very Large Data Bases*, PVLDB, 2013.