An Encrypted In-Memory Column-Store: The Onion Selection Problem

Florian Kerschbaum
with
Martin Härterich, Mathias Kohler
and
Isabelle Hang, Andreas Schaad, Axel Schröpfer, and Walter Tighzert

SAP
Karlsruhe, Germany
firstname.lastname@sap.com

Abstract. Processing encrypted queries in the cloud has been extended by CryptDB's approach of adjustable onion encryption. This adjustment of the encryption entails a translation of an SQL query to an equivalent query on encrypted data. We investigate in more detail this translation and in particular the problem of selecting the right onion layer. Our algorithm extends CryptDB's approach by three new functions: configurable onions, local execution and searchable encryption. We have evaluated our new algorithm in a prototypical implementation in an in-memory column store database system.

1 Introduction

Security is a major obstacle for widespread cloud adoption. Criminal hackers or foreign government organizations may try to gain access to the data stored in the cloud. Encryption of this data may provide a solution, but it may also prevent the use of and computation with it in the cloud. A current research challenge is therefore to enable the processing of the data in the cloud while it is encrypted.

One of the many cloud offerings is database-as-a-service [10]. A service provider offers database store and query capabilities, e.g., through an SQL interface, and charges for storage and computation. Clearly, there is less benefit if query processing would be done at the client and only storage in the cloud. Therefore, when using encryption queries need to be processed over encrypted data – ciphertexts.

This was first introduced in [9]. Relational database operators have been adapted to allow for processing over encrypted data. Binning was introduced as a method to handle range queries. In binning values are put in larger bins and all bins of a range are queried using equality matching. Values which are in the selected bins, but are not in the range are filtered using post processing on the client.

This approach was improved by order-preserving encryption [1]. Order-preserving encryption preserves the order of the plaintexts in the ciphertexts. It allows implementing range queries using the same relational operators as used in plaintext databases. Databases encrypted using order-preserving encryption can already perform most queries (except aggregation) without modification of the database operators.

Order-preserving encryption was first put on a formal basis in [4]. Boldyreva et al. developed the concept of a random order-preserving function that mapped a smaller domain to a larger domain while preserving the order of the inputs. They proved that their encryption scheme was a uniform random choice among all random order-preserving functions. Therefore no order-preserving encryption scheme could be better than theirs.

Still, what this security guarantee entails for the user remained (and somewhat remains) unclear. Later Boldyreva et al. analyzed their own scheme [5] and concluded that roughly the upper half of the plaintext bits leaks to a passive observer. They introduced the notion of ideal security for order-preserving encryption, i.e., that nothing should be leaked except the order.

The best ideal security order-preserving encryption is replacing the plaintexts by their order. This encryption is a mapping of a larger domain of plaintexts to a smaller domain of ciphertexts. The challenge is to be able to accommodate future plaintexts not foreseen when choosing the encryption. This challenge has not yet been fully solved, but Popa et al. propose an interactive protocol to compute the ciphertexts [21].

Even with ideal security the security of order-preserving encryption remains questionable. Popa et al. developed adjustable onion encryption as a means to reveal the order-preserving ciphertexts only when necessary for a specific query. The idea is to wrap onion layers of more secure encryption schemes around the order-preserving ciphertext. These layers are only removed when a range query requiring order preservation is processed. Using this approach most queries can be handled without resorting to order-preserving encryption.

In this paper we investigate the problem of identifying the right layer of encryption for processing a query, i.e., the onion selection problem, in more detail. We define our requirements of *policy configuration*, *alternative resolution* and *conflict resolution*. Based on these requirements we derive and present a new onion selection algorithm.

The remainder of the paper is structured as follows: In the next section we describe the different onions and their layers in default mode. We then define the onion selection problem and present our algorithm in Section 3. Finally, in Section 4 we give an outlook on the remaining challenges in processing queries over encrypted data.

2 Adjustable Onion Encryption

Adjustable onion encryption was introduced by Popa et al. in [22]. In this section we present the layers of the onion as described in their approach. We later outline

a number of alternatives and extension that complicate our problem of onion selection.

Differently from [22] which used a row-based, disk-store database management system (DBMS) we use a column-based in-memory DBMS [7]. The impact of the encryption schemes on the dictionary compression used in column-stores differs widely. We refer the interested reader to [15] for details. Still, also for onion handling column stores offer advantages. Namely, it is easy to add and remove columns on the fly during operation. This creates the opportunity to remove onions when no longer useful – a further option for future investigation.

2.1 The Onion Layers

We denote $E_C^T(x)$ the ciphertext (encryption) of plaintext x in type T (e.g. order-preserving) with key C (usually for a column). The corresponding decryption is $D_C^T(c)$, i.e., $D_C^T(E_C^T(x)) = x$ as in Dolev-Yao notation. We sometimes omit the key, if it is a single key. Note that all keys initially reside at the client. The database has no access to any key, unless revealed during the adjustment of encryption, i.e. "when peeling the onion".

In the "standard" onion there are three encryption types: order-preserving (OPE), deterministic (DET) and randomized (RND). The term onion stems from the layering of encryption, i.e., the output (ciphertext) of one encryption type is fed as in input to another. The onion of [22] looks like the following:

$$E^{RND}(E_C^{DET}(E^{OPE}(x)))$$

We describe the layers of this onion from the innermost – the plaintext – to the outermost – randomized encryption. The choice of this layering is not arbitrary. Each outer layer adds more security, i.e., the better the onion is preserved the better the security starting with industrial strength. Each inner layer adds functionality, i.e., inner layers allow more database operators to function as on plaintext. It is important to note that inner layers preserve the functionality of all outer layers, i.e., they never remove any functionality.

Plaintext We treat the plaintext x as its own layer. This has two reasons: First, not each encryption type has the plaintext as the innermost layer – see searchable encryption (Section 2.2). Second, addressing it as its own layer enables us to make it configurable. As we describe in our requirements (Section 3.2) a user may want to prevent access to the plaintext or not encrypt certain columns at all.

Clearly, the plaintext allows processing all queries, but provides no security.

Order-Preserving Encryption Order-preserving encryption (type: OPE) preserves the order of the encryption:

$$x \leq y \iff E^{OPE}(x) \leq E^{OPE}(y)$$

We use the scheme of Boldyreva et al. [4], since the ideally secure scheme of Popa et al. [21] is incompatible with adjustable encryption. We omit the key, since we use the same key for each column of the same data type. This is necessary in order to allow joins using deterministic encryption of the upper layer.

Since the order is preserved, order-preserving encryption enables processing range queries on ciphertext in the same way as on plaintexts. Furthermore, since our order-preserving scheme is also deterministic, it also still can process equality matching and joins. As already mentioned the security of order-preserving encryption is still debated.

Deterministic Encryption Deterministic encryption always produces the same ciphertext for a plaintext:

$$x = y \iff E_C^{DET}(x) = E_C^{DET}(y)$$

Deterministic encryption used to be the standard mode before randomized encryption was introduced. Bellare et al. have proven that only deterministic encryption can allow sublinear search [2]. Therefore equality matches and joins thereon can be performed more efficiently, i.e., using an unmodified database operator. Deterministic encryption still allows statistical attacks, such as [11]. Yet, since the order is permuted, it is more secure than order-preserving encryption.

Initially we use a different key C for each column. Clearly, this prevents joins based on equality matches, since the columns are encrypted differently. Blaze et al. introduced proxy re-encryption [3] – a technique to transform a ciphertext from key A to key B without decrypting it or needing to know any of the two keys A or B. We use this proxy re-encryption in order to adjust the encryption keys just before performing a join operation. For details how to choose which column to re-encrypt we refer the reader to [16].

We use the proxy re-encryptable encryption scheme of Pohlig and Hellman for deterministic encryption [20]. A similar scheme on elliptic curves has been used in [22].

Randomized Encryption Randomized encryption has been formalized by Goldwasser and Micali [8]. It has become the standard mode of encryption. A randomization parameter ensure that each ciphertext – even from the same plaintext – is indistinguishable. We use the AES standard in CBC mode for this encryption. We can use a single key, since each plaintext is indistinguishable.

Randomized encryption is the most secure, but allows no operation except retrieval to the client.

2.2 Alternative Encryptions

The layering of the onion does not need to be same in all cases and next to this "standard" onion other encryption schemes are necessary or useful. We introduce them in the following subsections.

Homomorphic Encryption Homomorphic encryption allows arithmetic operations, most notably addition, using the ciphertexts. In order to support aggregation without the plaintext, it is therefore necessary to incorporate homomorphic encryption. We use the encryption scheme by Paillier [18] which allows only addition:

$$D^{HOM}(E^{HOM}(x) \cdot E^{HOM}(y)) = x + y$$

The following operation for multiplication by a plaintext can be easily derived

$$D^{HOM}(E^{HOM}(x)^y) = x \cdot y$$

Paillier's encryption scheme is indistinguishable under chosen plaintext attack – as is randomized encryption. Therefore only a single key is necessary. Furthermore Paillier's encryption scheme is public key, but we keep the public key secret and treat it like a secret key encryption scheme.

There are some difference between the above encryption schemes and homomorphic encryption scheme. First homomorphic encryption requires aggregation (sum operator) to be processed differently than on plaintexts. Additions is replaced by multiplication, actually modular multiplication. This requires implementation of one's own operator. In [22] the authors use user-defined functions, we implemented our own database operator.

A second difference is that the result of the database operator – the sum – is still encrypted. Therefore it is not possible to use this result in many subsequent operations such as range queries. Queries using a chain of such operations need to be partially executed at the client on plaintext.

Searchable Encryption Searchable encryption allows the private or secret key holder to issue a search token for a query plaintext. This search token can be used to match ciphertexts for plaintext equality or range inclusion to the query. Unless a token has been issued searchable encryption is as secure as randomized encryption, but for search token the accessed ciphertexts are leaked. Let $T^{SRC}(x)$ denote the search token and M^{SRC} the matching operation.

$$x = / \neq y \iff M^{SRC}(T^{SRC}(x), E^{SRC}(y)) = \top / \bot$$

The research on searchable encryption has been sparked by Song et al. [24]. We use a scheme very comparable to theirs and implemented it using user-defined functions. Search time can be significantly improved by building an index, but this requires a modification of the database. Hence, we prefer the linear search.

Security for searchable encryption has been formalized by Curtmola et al. in [6] and they also present an index-based scheme. In their security definition access pattern leakage is explicitly excluded. This has been extended to dynamic searchable encryption which also allows updates [12]. Here, update patterns are also explicitly excluded which raises questions about the advantages for the user compared to deterministic encryption. Note that the scheme of [24] is more securely

updateable, since it does not use an index. The technique of Shi et al. can be adapted to also enable range queries [23].

Searchable encryption must not be decryptable. In our scheme [24] – and several others – the plaintext is used to construct key and a random number is encrypted. Therefore searchable encryption may not be used for retrieval.

Efficiency of searchable encryption is significantly lower than deterministic encryption. It can therefore be advisable to not include it as an option for database operations. Before considering searchable encryption we call a function to make this decision.

Client Decryption Next to functionality as in homomorphic encryption and security as in searchable encryption also efficiency can be a reason to add an onion. A bottleneck during encrypted query processing is decrypting the result on the client. It can therefore be more efficient to use a specific onion with efficient decryption. We use AES in CBC mode as in randomized encryption, but directly on the plaintext.

When specifying the configuration of this onion, care must be taken not to use the plaintext for database operations unless intended. We accomplish this by introducing our own layer RET for retrieval.

2.3 Common Onion Configurations

As the previous sections describe there are different alternatives for encryption. We enable the user to configure its onions for a specific column. As it is too difficult to foresee all possible onion configuration and capture all semantic restrictions of the encryption schemes, we cater for a number of common onion configurations. We successfully tested our algorithm on these onions. Other configurations are possible and often work correctly, but are not set of the defined scope.

We foresee three options: for processing queries in the cloud, for strong encryption and processing queries on the client and for no security. The option for processing queries in the cloud is our default. It is similar to [22], but extended and adapted as above. This option consists of four encryptions:

$$O_1 : [RET]$$
$$O_2 : [RND[DET[OPE]]]$$
$$O_3 : [HOM]$$
$$O_4 : [SRC]$$

We can extend this option by omitting the OPE layer and preventing from decryption to order-preserving encryption (while allowing deterministic encryption). As such, this extended option is "middle ground" between processing all queries in the cloud and our next option of processing all queries at the client.

The option for strong encryption requires all processing (except aggregation) to be processed at the client. Aggregation is done using homomorphic encryption

which is as secure as randomized encryption. Hence, it can be included without sacrificing any security.

$$O_1 : [RET]$$
$$O_2 : [HOM]$$

The third option is to not use encryption at all. This can be used for non-critical data and increases the efficiency of processing. Note that it is not possible to combine this data with the encrypted data in the cloud, since the query processing option does not include an accessible plaintext.

$$O_1 : [PLN]$$

3 Onion Selection

In this section we describe the algorithm to select the onion layer – and corresponding database operator – for performing the query.

3.1 Problem Definition

The client issues its query to its database driver. The database driver intercepts and analyzes the query. It constructs an initial query plan based on relation algebra of the query. This plan consists of a tree of database operators O. The set of database operators includes projection (column selection), selection (where conditions for equality and ranges), joins (equality and ranges), grouping, sorting, aggregation (sum), union and a few others.

Each database operator's input and output are tables and each operator performs an operation on one or two columns. The semantic of this operation, e.g., an equality match, is encoded in the operator's type. Operators are connected into a tree with the root returning the result set of the query and the leaves being raw database tables (or views).

For each column there is a configurable set of onions – as described in Section 2.3. The original query executes on a virtual table that has been converted in the ciphertexts of this onion. Therefore the query needs to be rewritten in order to return the same result as the original query on the original table.

Definition 1. *The* onion selection problem *is to select the onion layer for performing an equivalent operation on the encrypted data as part of the original query on plaintext data.*

3.2 Requirements

As mentioned in the introduction, we intend to meet three requirements: *policy configuration*, *alternative resolution* and *conflict resolution*.

Policy Configuration This requirement refers to the user ability to control the onion selection. A common policy could be, e.g., a security policy to never reveal a plaintext in the cloud or to never even reveal a non-randomized encryption. We implement these policies indirectly via configuration of the onions. For each column the user can specify the onion. If an onion layer, e.g., plaintext, is not available, then it is a not an option for the onion selection algorithm.

Of course, the onion configuration also has implications on efficiency and maybe even functionality. An onion configuration therefore must be carefully chosen and tested.

Alternative Resolution This requirement refers to the availability of multiple onion layers option for a corresponding plaintext database operation. A typical example is equality matching in a where clause. This can be either fulfilled using deterministic encryption or searchable encryption. Deterministic encryption is more efficient and searchable encryption is more secure.

Our onion selection algorithm needs to make a choice. Its choice is to always select the most secure variant and if there are still multiple, then to select the most efficient remaining one. This order is motivated by the purpose of encryption: to ensure security. Furthermore, the user can influence the level of security by the onion policy configuration.

Conflict Resolution This requirement refers to the situation where database operators are incompatible due to the encryption mode. A typical example is performing a range query on aggregated data, e.g. `SELECT x FROM t GROUP BY x HAVING SUM(y) > 10`. The aggregation can only be computed using homomorphic encryption which cannot be used for range queries on the client. Another example are joins between columns that use different onion configurations. An onion for processing data in the cloud cannot be matched to plaintext, since there is no plaintext.

Our onion selection algorithm needs to detect these situation. Furthermore, it follows a simple strategy to resolve these conflicts. As many database operators as possible are executed on the server. Once there is a conflict the intermediate result table will be transferred to the client and processing continues there on the plaintexts. As such, all queries are executable at the expense of transferring (and decrypting) additional data.

3.3 Algorithm

In order to make a selection of the most secure onion layer we impose an order on the encryption layers. We call the innermost layer the smallest or minimum layer and the outermost layer the largest or maximum layer. Let PLN be the plaintext layer.

$$T : RND, HOM, SRC, RET > DET > OPE > PLN$$

Our algorithm proceeds in five steps.

1. We build a graph of columns used in the query.
2. For each node of the graph we select the maximum layer that can fulfill the query.
3. For each connected component we select the minimum layer necessary.
4. If there are multiple parallel onions remaining, we choose the most efficient for the database operation.
5. We scan the operator tree from the leaves to the root. On the first conflict we execute the upper part on the client.

Step 1: Column Graph We create a node for each raw table column used in the query, i.e., columns selected, columns in where or having conditions, groupings or sorted, or columns used in aggregation functions. Note that we do not create virtual columns for aggregations.

For each node we retrieve the onion configuration and attach it to the node as a structure. This may be multiple onions as in our configuration for processing encrypted queries in the cloud. We use the onion configuration as it is currently stored in the database, i.e., previous query may have already removed layers of onions.

We create an edge for each join operator between the columns used in the condition or conditions. The resulting graph is undirected, but only in rare cases connected.

An example is the query *SELECT t1.x, SUM (t2.y) FROM t1, t2 WHERE t1.i > 10 AND t1.i = t2.i GROUP BY t1.x*. This query uses the columns $t1.x$, $t2.y$, $t1.i$ and $t2.i$. In the resulting graph there is an edge between $t1.i$ and $t2.i$.

Step 2: Column Layer Selection We iterate through all operators in the tree. For each operator we retrieve the onion and its associated onion structure. Each operator has a type (and semantics) that imply one or more necessary onion layer. For the encryption type *SRC* we call a specific function that determines whether it can be considered for this operator. If not, then it is not considered a match for the operator. If none of these onion layers is currently accessible (because it is wrapped in another layer), we successively remove layers from the onions – starting with the topmost only – until a layer appears that fulfills the semantic requirements.

Already at this step we may encounter a conflict when no onion layer for the operation is available. We then mark the structure empty as an indication that there is a conflict.

Note that we assume that layer removal increases the query processing functionality of the ciphertext. This needs to be considered when configuring the onions. It is easy to violate this assumption and then the algorithm may fail.

Consider the example query from above: *SELECT t1.x, SUM (t2.y) FROM t1, t2 WHERE t1.i > 10 AND t1.i = t2.i GROUP BY t1.x*. For simplicity assume that all onions use the same configuration for processing queries in the cloud. Furthermore, assume that no queries have been processed so far and state in the database is the initial state.

The first operator is a projection for $t1.x$. This can be fulfilled by RET, RND, DET, OPE, HOM or PLN. There are multiple layers in the current onion structure and no modification is necessary.

The second operator is an aggregation on $t2.y$. This can be fulfilled by HOM or PLN. Again, such a layer is in the current onion structure and no modification is necessary. In an alternative configuration one could drop the homomorphic encryption. Then, there would be a conflict and the structure would end up empty.

The third operator is a range query on $t1.i$. This can be fulfilled by OPE or SRC. Assume that searchable encryption is not an option due to our function call. Then there is no such layer in the current onion structure. We therefore remove the uppermost layers RND and DET.

The fourth operator is a join on equality matching. This can be fulfilled by DET, OPE or PLN. For $t1.i$ such an OPE layer is in the current onion structure due to the third operator and no further modification is necessary. For $t2.i$ no such layer is in the current onion structure. We therefore remove the uppermost layer RND.

The fifth operator is a grouping on $t1.x$. This can also be fulfilled by DET, OPE, or PLN. There are multiple layers in the current onion structure and no modification is necessary. Since the first operator made no modification there is no such layer in the current onion structure. We therefore remove the uppermost layer RND.

Step 3: Connected Component Selection We now iterate through each connected component of the column graph. For each node in a connected component we retrieve the onion structures. For each onion we select the minimum layer in any of the structures. The resulting minimum onion structure is stored in all nodes of the connected component.

Note that the minimum for a common layer of deterministic encryption (DET) includes the use of a common key. This common also needs to be computed and proxy re-encryption to be performed. See [16] for details.

We may encounter a conflict here if the columns in a connected component use different configurations. In this case we abort and leave all onion structure as they are. We handle this conflict in step 5 of the algorithm.

We continue the example from above: *SELECT t1.x, SUM (t2.y) FROM t1, t2 WHERE t1.i > 10 AND t1.i = t2.i GROUP BY t1.x*. For the nodes $t1.x$ and $t2.y$ we are already done, since they are single nodes. The onion structure for $t1.i$ includes OPE whereas for $t2.i$ it includes DET. We set all to OPE, i.e., the minimum and perform the join on order-preserving encryption.

Step 4: Alternative Selection We again iterate through all operators in the tree. For each operator we select the most efficient onion layer of the available choices in the current onion structure. We mark the operator with the selected onion layer.

We iterate through all operators in the tree a second time and mark all onions that are used. If a layer of an onion is supposed to be removed, but not used, we restore the onion in its structure to its current state, i.e., the layer removal will not be executed.

In the example from above there is only one operator which can use multiple layers: projection of $t1.x$. This operator can use DET or RET. The most efficient is RET and hence used.

In some databases, e.g. [7], there is a semantic dependency between grouping and projection (or sorting). Only the same column can be used in both, i.e., in our example $t1.x$ must be the same column in grouping operator and projection. Nevertheless, the RET and the DET layers are in different onions and hence columns. In our implementation we capture this dependency using a specialized operator. Then, we use the minimum operator on the RET layer in order to return a unique result.

Step 5: Local Execution Split In the last step we scan the operator tree from the leaves to the root. If we encounter a conflict, the children of this operator are cut and it and the parents are marked to be executed on the client. Conflicts are

- There is no onion layer to fulfill the operation. Examples that can occur in a "correct" configuration are, e.g., an onion for processing on the client with an operator to be executed.
- The operation is on an *aggregate function*, but not the same aggregate function. An example is `SELECT x FROM t GROUP BY x HAVING SUM(y) > 10` with our onion configuration for processing in the cloud.
- A join operation on different onion configurations.

In our running example there are no conflicts and the entire query is executed on the server. If a query is to be executed on the client, the subtrees rooted at its children are synthesized into SQL queries. These queries are then executed and their result stored in local, temporary tables on the client. The temporary tables are decrypted and the upper part of the query is synthesized. This query is then executed locally on the temporary tables.

4 Summary and Outlook

Compared to *CryptDB* [22] we introduce three main extensions. First, we allow the user to configure the onion in almost arbitrary ways directing the onion selection. Second, we introduce a local execution split in order to allow queries that cannot be otherwise fulfilled. Third, we enable the use of searchable encryption.

Already these three simple extensions make the algorithm significantly more complex. They introduce additional problems, such the availability of multiple or none encryption layers for executing the query. Furthermore, theoretically they can handle all SQL queries by executing them on the client. They also

may increase security by using searchable encryption instead of deterministic or order-preserving encryption.

Still, there are many possibilities for misconfigurations by the user. An option to reduce these is a tool that checks the most important semantic constraints of an onion configuration. This tool can check for the most common problems and warn in case it does not recognize a configuration.

The use of searchable encryption is subject to further evaluation. Once our results are available we intend to publish a full report. Particularly, the implementation of the function whether to use searchable encryption is critical.

Another future problem is to enable collaboration in an encrypted database. Consider a number of organizations joining forces in intrusion detection. Each organization may want to share selected events with certain other organizations. Still all data is to be stored centrally in the cloud. Other such examples are benchmarking [13, 17] and supply chain management [19].

Note that our onion selection algorithm already performs a lightweight optimization: first security, then efficiency. Clearly, it is interesting to investigate more alternatives to make such choices. An interesting example from secure computation on a generic programming language is [14]. SQL query optimization is a well studied field which can offer a number of insights.

References

1. R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2004.
2. M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and efficiently searchable encryption. In Advances in Cryptology (CRYPTO), 2007.
3. M. Blaze, G. Bleumer, and M. Strauss. Divertible protocols and atomic proxy cryptography. In Advances in Cryptology (EUROCRYPT), 1998.
4. A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In *Advances in Cryptology (EUROCRYPT)*, 2009.
5. A. Boldyreva, N. Chenette, and A. O'Neill. Order-preserving encryption revisited: improved security analysis and alternative solutions. In *Advances in Cryptology (CRYPTO)*, 2011.
6. R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, 2006.
7. F. Färber, N. May, W. Lehner, P. Groe, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database – an architecture overview. *IEEE Data Engineering Bulletin 35(1)*, 2012.
8. S. Goldwasser, and S. Micali. Probabilistic encryption. *Journal of Computer and Systems Sciences 28(2)*, 1984.
9. H. Hacigümüs, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2002.
10. H. Hacigümüs, B. Iyer, and S. Mehrotra. Providing database as a service. In *Proceedings of the 18th IEEE International Conference on Data Engineering (ICDE)*, 2002.

11. M. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, 2012.
12. S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, 2012.
13. F. Kerschbaum. Building a privacy-preserving benchmarking enterprise system *Enterprise Information Systems 2(4)*, 2008.
14. F. Kerschbaum. Automatically optimizing secure computation. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.
15. F. Kerschbaum, P. Grofig, I. Hang, M. Härterich, M. Kohler, A. Schaad, A. Schröpfer, and W. Tighzert. Demo: Adjustably encrypted in-memory column-store. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, 2013.
16. F. Kerschbaum, M. Härterich, P. Grofig, M. Kohler, A. Schaad, A. Schröpfer, and W. Tighzert. Optimal re-encryption strategy for joins in encrypted databases. In *Proceedings of the 27th IFIP Conference on Data and Applications Security (DBSEC)*, 2013.
17. F. Kerschbaum, and O. Terzidis. Filtering for private collaborative benchmarking. In Proceedings of the International Conference on Emerging Trends in Information and Communication Security (ETRICS), 2006.
18. P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology (EUROCRYPT)*, 1999.
19. R. Pibernik, Y. Zhang, F. Kerschbaum, and A. Schröpfer. Secure collaborative supply chain planning and inverse optimization-the jels model. *European Journal of Operational Research 208(1)*, 2011.
20. S. Pohlig, and M. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. *IEEE Transactions on Information Theory 24*, 1978.
21. R. Popa, F. Li, and N. Zeldovich. An ideal-security protocol for order-preserving encoding. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (SP)*, 2013.
22. R. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.
23. E. Shi, J. Bethencourt, H. Chan, D. Song, and A. Perrig. Multi-dimensional range query over encrypted data. In *Proceedings of the 28th IEEE Symposium on Security and Privacy (SP)*, 2007.
24. D. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 21st IEEE Symposium on Security and Privacy (SP)*, 2000.