Making Query Execution Over Encrypted Data Practical

Ken Smith, M. David Allen, Hongying Lan, and Andrew Sillers

Abstract The benefits of data outsourcing continue to grow, however owners of sensitive data cannot take full advantage due to its risk profile. Encrypted query processing promises to change this situation and allow data owners to securely outsource their sensitive data: data is encrypted, installed in a database on a remote (e.g., cloud) server, and standard queries are processed against the remote encrypted data. Correct query answers are returned without ever exposing plaintexts or decryption keys at the server. This chapter addresses three key challenges to realizing, as a practical option, the promise of encrypted query processing: handling query operations which cannot execute in ciphertext, implementing a working system, and achieving acceptable query performance.

1 Background: Clouds and Outsourcing

The trend to outsource data to third party clouds continues to grow, however for owners of sensitive data, clouds hold both great promise and vexing problems.

1.1 Outsourcing Data Management: The Promise

Renting a computing infrastructure frequently makes much better sense than owning and running one. Outsourcing the management of computing assets allows an organization to focus personnel, training, and hiring on their core business. It also offers unprecedented agility, such as near instant expansion and contraction of the organization's IT footprint as software development cycles and seasonal business

K. Smith (✉) • M.D. Allen • H. Lan • A. Sillers
The MITRE Corporation, McLean, VA, USA
e-mail: kps@mitre.org; dmallen@mitre.org; hlan@mitre.org; asillers@mitre.org

S. Jajodia et al. (eds.), *Secure Cloud Computing*, DOI 10.1007/978-1-4614-9278-8_8,

demands require, and takes advantage of the cost efficiencies of a volume provider of computing services, which have been compared to the efficiencies of household gas, water, and electric utilities. Especially in the era of big data, the cost of servers, disks, space, power, and cooling can far exceed the budget. Once purchased, computing assets must be actively patched, repaired, and upgraded; such costs can be avoided by renting.

In addition, outsourcing providers now offer a continually growing array of services that its customers could not afford to develop themselves. For example, Amazon Web Services offers rentable services such as inexpensive data archival, on-demand map reduce clusters, and subnets with private IP addresses [18].

This combination of rentable computing infrastructure and novel computing services makes widely available modes of computation which were previously impossible, or out of reach due to cost. Consider a medical experiment which generates and analyzes huge genetic datasets. The research funding to rent storage and computing on an as needed basis is far less than that required to purchase these and to pay professional staff to manage them. Using an outsourced infrastructure, novel studies can be proposed which might not otherwise be feasible under research funding.

1.2 Outsourcing Data Management: The Problem

Owners of *sensitive* datasets however, can be caught between the promise of outsourcing and the problem of losing control of part of the computing stack (Fig. 1). For an infrastructure as a service (IaaS) cloud, these stack layers include: hardware, virtualization, fabric, and customer-installed software applications (e.g., DBMS, web server, GUI); customers only control the final layer. Even with full confidence in cloud-supplied layers (e.g., the customer does not expect hypervisors to ever be compromised), cloud security engineering requires careful teamwork between

Fig. 1 Cloud security: the challenge of letting go

the outsourcing vendor and the customer. The security features of vendor-supplied layers and customer-supplied layers must mesh without a flaw when they are used to implement a solution together. In this case, the utility analogy breaks down, because consumers rarely interact with their household and gas utilities beyond simply paying bills and turning service on and off.

Owners of sensitive datasets must also worry about the other participants in a cloud ecosystem. Unlike a self-managed infrastructure, the cloud ecosystem includes *cloud neighbors*, who typically belong to unknown organizations. In several published attacks, the attacker becomes a cloud neighbor of their target to stage the attack. For example, [22] illustrates a *side-channel* attack on a physically collocated virtual machine (i.e., one sharing the same physical host as the attacker's virtual machine), enabling the attacker to steal a cryptographic key in the target virtual machine by examining shared hardware resources.

The cloud ecosystem also includes vendor-supplied *cloud administrators*, who are typically assumed to be "honest but curious" [11]. However, this is not always the case. Recently, German citizens hiding their money in Swiss bank accounts to evade high national taxation rates were identified because the German government bribed the bank's database administrator [17]. Owners of sensitive government data cringe at the thought that a foreign government could influence a cloud administrator to do something like this.

In addition, in the advanced persistent threat (APT) attacker model [21], cited for the exfiltration of significant amounts of intellectual property, any person's online identity can be compromised (e.g., via a phishing attack) allowing the APT attacker to masquerade with the full privileges of the compromised identity. Therefore, any member of a cloud ecosystem could potentially become an attacker.

Due to such problems, owners of sensitive data are currently conflicted with respect obtaining the agility, services, division of labor, and efficiencies clouds can offer.

2 Using Data Encryption

A potentially game-changing strategy is the use of encryption to protect sensitive data in clouds. Encrypted data is mathematically transformed so only the possessor of a decryption key can reconstitute the original *plaintext* data without A prohibitively expensive computational effort. Thus, if sensitive cloud data is encrypted, an exfiltration attack does not truly succeed unless the attacker can additionally obtain the decryption key, or successfully attack the cryptosystem. This is true *regardless* of the stack layer the attack originates from, or the cloud denizen who executes it.

2.1 Pre-transmission Dataset Encryption

A simple strategy having these benefits is to encrypt each dataset prior to its transmission to the cloud, and to only decrypt it upon retrieval from the cloud. The downside of this strategy is that cloud applications cannot operate over these encrypted datasets, they must be downloaded before use. Consider the query "What is the location of helicopter 21?". In a normal cloud database deployment, the database would look up helicopter 21, and return a very small result relative to the size of the entire data set. For monolithic encrypted files and datasets, there is no way for the server to look up helicopter 21. Instead of returning a small answer, the entire database would need to be retrieved. With "big data" era terabyte and larger datasets, downloading the entire dataset before use is simply impractical.

2.2 Data-at-Rest Encryption

Data at rest encryption protects sensitive data in a storage system, can be used with cloud-based data, and allows computation over that data. Data is encrypted when stored on any cloud storage device, and decrypted when requested by an application. Data at rest encryption is used for many types of sensitive data, including personal health data covered by HIPAA, and sensitive but unclassified military information. Data at rest encryption is especially useful against physical attacks, such as a stolen laptop or disk drive, and mature products exist in which the user need not be an expert cryptographer or make large performance sacrifices. For example, Oracle's Transparent Database Encryption product (TDE) [14] now provides at rest encryption for Oracle DBMS's, exploiting new hardware encrypt/decrypt instructions [10].

Unfortunately, data at rest encryption does not protect *data in use*. It requires a decryption key to be available in the cloud so data can be decrypted and used by applications. As mentioned earlier, many attacks are aimed exactly at obtaining the decryption key. Furthermore, the moment a query hits a cloud application (e.g., a TDE encrypted database), data is decrypted and brought into cloud memory as plaintext. Thus, attackers do not actually need to obtain the key to defeat data at rest encryption, they simply need to exfiltrate plaintexts from cloud memory. Note that performing *data in use* encryption has not been added to standard security requirements simply because useful commercial solutions do not exist at this time.

2.3 Homomorphic Encryption and Computing Over Ciphertexts

Homomorphic cryptosystems promise the best of both worlds:

1. The ability to expose neither plaintext data nor decryption keys in clouds.

2. The ability for applications to nonetheless compute over encrypted data while it resides in the remote cloud.

Cryptosystems are valued primarily for their ability to secure information. As a side-effect, however, operations on their corresponding ciphertexts in some cryptosystems correspond to useful operations on plaintexts, which is called a *homomorphism* [5].

For example, in the Paillier [15] cryptosystem, the modular multiplication of two ciphertexts corresponds to the addition of their plaintexts. Thus, for two plaintext numbers m_1 and m_2, given only $E(m_1)$ and $E(m_2)$ (the encryptions of m_1 and m_2 respectively), and the public encryption key, it is possible to compute $E(m_1 + m_2)$ without access to the corresponding plaintexts. Other pairs of ciphertext and plaintext operations, although not strictly homomorphic provide identical utility. For example, in any deterministic cryptosystem, equality tests on ciphertexts correspond to equality tests on plaintexts. Thus, through the use of such cryptosystem properties, it is possible to perform useful operations on data *without ever decrypting it*.

Paillier is *additively* homomorphic because its homomorphism implements addition over plaintexts. Other cryptosystems (e.g., RSA) are *multiplicatively* homomorphic. The question naturally arises as to whether any cryptosystem is *fully homomorphic*, enabling *any* computable operation over plaintexts to be performed using ciphertext datasets.

Since being posed in 1979, the fully homomorphic encryption (FHE) problem remained open for over 30 years. It was recently solved by Craig Gentry [6], for which he won the 2009 ACM Dissertation award. Although Gentry's cryptosystem is fully homomorphic, and semantically secure, its performance degrades sharply with its security parameter. For a practical degree of security, performance of Gentry's original algorithm has been estimated to be as bad as 10 orders of magnitude worse than the corresponding plaintext operations [4], such that a one second computation would take over three centuries. To address this disparity, in 2011 DARPA initiated the PROCEED program [4]; research on the optimization of FHE is now very active, with several orders of magnitude improvement realized for various portions of FHE (e.g., key generation) [7, 19]; portions of this research have also been released as open source code [9]. However, for the foreseeable future, FHE remains computationally impractical. In addition, an efficient FHE implementation would not immediately enable users to execute conventional queries in a cloud-based PBMS. As the entire DBMS would have to be rewritten as a homomorphic function, a massively complex undertaking. Thus, in the following, we focus on the use of homomorphisms within the context of an existing DBMS.

2.4 Making Practical Tradeoffs

The FHE algorithms in Gentry's thesis illustrate a general principle regarding homomorphic computing. As illustrated in Fig. 2, a three-dimensional space of desirable features exists for homomorphic encryption: *functionality*, *security*, and

efficiency. Gentry's FHE algorithms provide full computational functionality over plaintext, a very high level of security (i.e., semantic security), but very poor efficiency with respect to the equivalent operations over plaintext.

A cryptosystem with ideal qualities on all three axes does not exist, however, other useful points in this space make tradeoffs differently than Gentry's FHE. The Paillier cryptosystem has similar security to FHE, provides only partial homomorphic functionality (i.e., addition), but is *much* more efficient than FHE (within two orders of magnitude of plaintext addition. Microsoft researchers have recently developed a partially homomorphic cryptosystem [13] which can add integers in about 200 µs per addition (versus 15 µs in Paillier), however, their partially homomorphic functionality is much greater, enabling the computation of statistics like the variance over ciphertexts.

The key insight is that it is not necessary to realize *fully* homomorphic functionality to to provide practical benefits for users today who want to use sensitive data in clouds. It is sufficient to securely and efficiently achieve the functionality required to implement a useful cloud application. For example, most computations in the SQL language can be implemented without requiring full Turing-complete functionality.

2.5 The Database as a Service Architecture

In a groundbreaking 2002 paper [8], Hacigümüş et al. proposed a software architecture for implementing *practical* (i.e., sufficiently efficient, secure, and functional) SQL computations over a remote encrypted database server (e.g., hosted in an outsourced cloud infrastructure). Instead of relying on a single fully homomorphic cryptosystem, this architecture can utilize a carefully-chosen set of partially homomorphic cryptosystems. In other words, this architecture can be used to exploit the individual strengths of *multiple* points in the space of Fig. 2. Plaintext

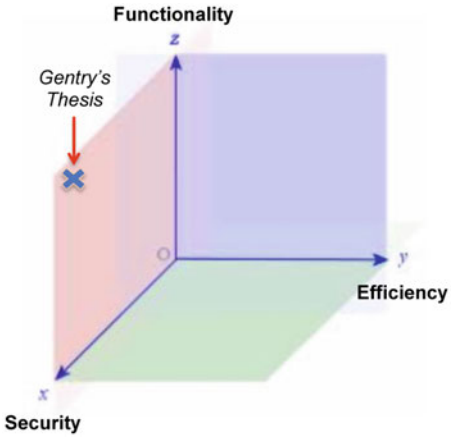

Fig. 2 A three dimensional tradeoff space for homomorphic encryption

SQL operations are translated into the appropriate homomorphic operations, similar to how a compiler translates programming language constructs into the appropriate machine codes.

As shown in Fig. 3, the user's original plaintext SQL query (bottom center) is translated into a query over encrypted data within a trusted client (left side). The correctness of the encrypted query is ensured by translation algorithms which substitute plaintext SQL operations for equivalent homomorphic operations.

The encrypted query is then sent off to a standard relational DBMS at the untrusted server (right side). While the table names, column names, and constants of encrypted query are ciphertext, the query itself remains a syntactically correct SQL query. The untrusted server DBMS thus naively executes it, and produces a set of encrypted results, which are then returned to the client (the temporary results area) and decrypted. In the final step, as discussed in the following section, the query executor applies any necessary *post-processing* to the decrypted results to generate the final correct plaintext answer, which is then returned to the user.

Thus, even though the database is fully encrypted and neither plaintexts nor decryption keys are ever exposed to the server, the end user issues the *same* SQL query and receives the *same* answer as if the database were standard plaintext.

2.6 Current Status and Prototypes

The vision of this paper has grown more compelling with time, as cloud architectures and their need for security has increased in importance, leading to its receipt of the 2012 ACM SIGMOD 10 year *Test of Time* award [1]. A large literature has also resulted from this initial paper, exploring suitable cryptosystems (e.g., varieties of order preserving encryption [2, 3]) and "bucketization" strategies which enable

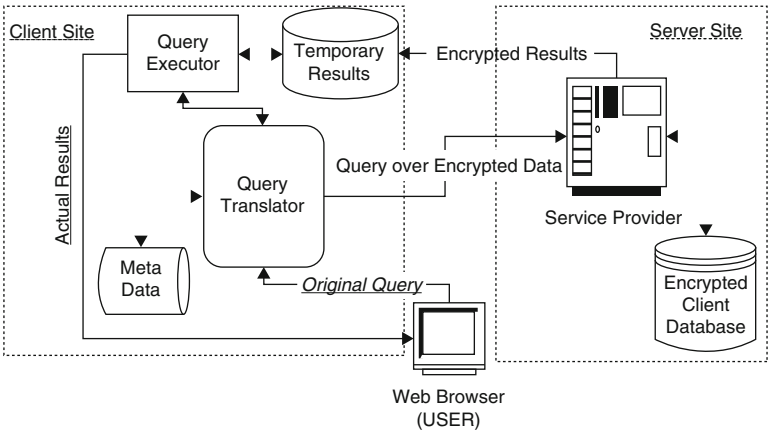

Fig. 3 Database as a service architecture

a tradeoff between the security and efficiency dimensions of the space in Fig. 2. However, as cited in the *Test of Time* award, no practical, commercially available, product which executes queries over encrypted data is available at this time.

Three notable prototyping projects exist, however, which provide valuable insights into the requirements for the practical realization of this technology. The first was developed as part of Hacigümüş' dissertation, and includes a general *planner* for encrypted query execution and introduced the *bucketization* strategy. The second, CryptDB [16] was developed as part of Raluca Popa's dissertation at MIT. CryptDB introduced features like onion encryption, and implemented several novel cryptosystems (e.g., a cryptosystem supporting dynamic joins between tables whose join keys were not previously encrypted with a congruent encryption key, along with algorithms for query processing time re-encryption of join keys). The third system is the MITRE DataStorm project [20], which contributed the IDEA system for generating the encrypted schema, a more detailed system architecture, and whose general focus is identifying and addressing the major barriers to practical computation over encrypted data.

3 Overview of Remainder of Chapter

These projects have yielded valuable insights. In the remaining sections we address three important challenges to the practical realization of this vision of executing database queries over encrypted data:

1. *Unexecutable query operations*: how do we execute query operations which *cannot* be executed over encrypted data? (Sect. 3),
2. *System implementation*: How do we mitigate the complexity of selecting an appropriate set of cryptosystems to apply to a specific user's query workload, using them to create an encrypted database on the server, and setting up a client-server system to service user queries over encrypted data? For this technology to be practical, a user should not be required to have a good understanding of fields like cryptography and query planning. (Sect. 5),
3. *Ciphertext query performance*: In addition to encryption and decryption, homomorphic operations over ciphertexts may be slower than their plaintext versions, and ciphertext expansion of plaintext may result in network delays. Where are the "sweet spots" for the performance of encrypted queries? (Sect. 6).

In each section, we describe the challenge, discuss how it can be addressed, and discuss the prospects for a practical solution. We draw heavily on the experiences of the DataStorm project due to its practical direction, however we also bring in lessons from the other two projects as well. Finally, in Sect. 7, we discuss general prospects for the future.

4 Unexecutable Query Operations

The first challenge is that some user queries may contain operations which, for several different reasons, cannot be executed over a ciphertext database. From the perspective of relational query optimization, we typically desire to *push* selections deeper in the query execution tree, but sometimes cannot. Analogously, here we desire to push operations in the plaintext query into an encrypted execution at the remote cloud server, but for the reasons given below, we cannot.

4.1 Reasons Operations Cannot Be Executed Over Ciphertext

While fully homomorphic encryption (FHE) is a reality, it is not a practical option for cloud users due to its current performance profile. Without the availability of a secure fully homomorphic cipher, we seek to compose a set of partially homomorphic ciphers which will cover the needs of database query operations. So far, we have presented the Paillier cryptosystem as a running example of a partially (additively) homomorphic cipher, but there are many other possibilities. For example, unpadded RSA and ElGamal [12] are multiplicatively homomorphic, and the Goldwasser-Micali cryptosystem is homomorphic with respect to the exclusive-or operation. However, at the present time, the set of operations in SQL is greater than the set for which we have direct translations into partially homomorphic cryptosystems. This is one reason for unexecutable query operations.

Furthermore, ciphers are of course not created equal with respect to their strength and security; in some situations (such as the use of unpadded RSA, which loses semantic security) although a partially homomorphic cipher may provide the desired operation, it not be a reasonable choice because it does not meet the security requirements of an application.

A third type of operation which cannot be executed in ciphertext is one that results in what we call an "encryption type mismatch", an issue first identified in [16]. Consider the < operation in the query segment `WHERE age < (SELECT SUM(years) FROM employee)`. If the input to the `SUM()` operation (an encryption of the integer `year`) is a Paillier ciphertext to enable the computation of a summation over ciphertexts, the output will also be a Paillier ciphertext. However, Paillier ciphertexts cannot be used in the ensuing order test, because Paillier is not an order-preserving cryptosystem. Although plaintext operands must only agree with their operator in datatype (e.g., string, integer), ciphertext operands must agree not only in datatype but also in encryption type. In this example, the < order test cannot be executed in ciphertext because its second operand is of the wrong encryption type.

So to summarize, there are three key reasons preventing operations in plaintext queries from being translated into operations which can be executed over ciphertext (i.e., pushed to a cloud).

1. *No available homomorphic operation*. The plaintext operation (e.g. string concatenation, cube roots) simply lacks an appropriate homomorphic ciphertext operation.
2. *Insufficiently secure homomorphic operation*. Although homomorphic ciphertext operations exist, none have a security profile which meets the requirements of a local security policy. For example, an order test in plaintext queries (e.g., `WHERE age < 21`) is directly and efficiently implemented via an order preserving cryptosystem. However, such a cryptosystem reveals the order of the encrypted plaintexts. If this is the only ciphertext implementation of an order test, and it violates local security policy, order tests cannot be executed over ciphertext.
3. *Encryption type mismatches*. A plaintext operation cannot be translated into a homomorphic operation whose operands have the required ciphertext type.

If a plaintext query, or a coherent plaintext query workload, contains any unexecutable operations, encrypted query execution is unavailable without way to address these operations. In the following we discuss the use of a *post-processing* architecture to enable the execution of queries and query workloads which contain unexecutable operations.

4.2 Post-processing

Post-processing is illustrated by the architecture in Fig. 4 (which is representative of the Hacigümüş and DataStorm prototypes). The data owner initially encrypts their schema and database instances and installs these on the outsourced server as the Encrypted DB. During query processing, the user or application submits a plaintext query Q to a middleware application (developed to enable encrypted query execution) within its trusted client. The middleware's planner rewrites Q into a set of queries represented by Q' and Q'' in Fig. 4 which execute: (a) at the server in the encrypted database, and (b) (for any query components with unexecutable operations) at the client in the middleware post-processor, over decrypted plaintext results returned from the encrypted server. A correct plan produces the same results as running Q against the original plaintext database.

Consider query Q in Fig. 5. Q's WHERE clause contains two parts, one which is executable over ciphertext and one which is not (due to the SQL LIKE clause). The planner generates query Q' for execution at the encrypted server. Note that table names, column names, and constants are all encrypted in Q', however it remains a well-formed SQL query. The encrypted database sends the results of Q' back to the middleware where they are decrypted. In the middleware post-processor, query Q'' is executed (in an in-memory DBMS) over the decrypted results, applying the final portion of the query and returning the final correct answer.

Post-processing thus makes it possible to execute queries containing operations which are unexecutable over ciphertext (e.g., LIKE, cos(), encryption type mismatches).

4.3 Planning

Query planning can be simple in many cases. If the query contains *no* unexecutable operations, all execution occurs at the server and the post-processing step is skipped. For many more queries, for example Q in Fig. 5, a relatively simple "U-shaped" plan is the best choice (i.e., it is correct and no more efficient plan can be found).

However, some queries require a more sophisticated plan. Consider a query to retrieve all 30 year old employees who make less than the average salary:

```
SELECT * FROM emp
WHERE emp.age = 30 AND emp.salary <
SELECT AVG(emp.salary) FROM emp
```

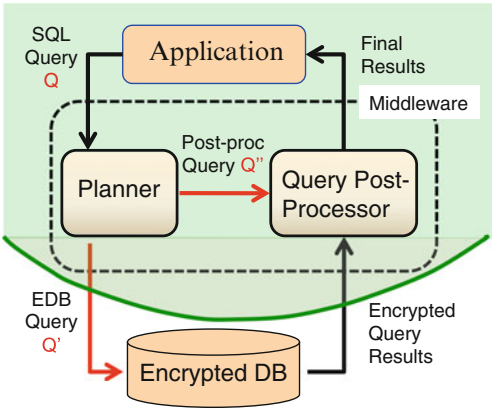

Fig. 4 Post processing architecture

Original Query (Q):

```
SELECT Name, Salary
FROM   Employee
WHERE Salary < 100,000
    AND Loc LIKE '%McLean%'
```

Encrypted DB Query (Q'):

```
SELECT e(Name), e(Salary)
FROM   e(Employee)
WHERE e(Salary) < e(100,000)
```

Post-processing Query (Q"):

```
SELECT col1, col2
FROM    Result
WHERE col3 LIKE '%McLean%'
```

Fig. 5 Query rewriting

Note that the average contains a division. If we encrypt salary with the Paillier cryptosystem, we can compute the sum (and count) at the server over ciphertext, but not the final division, which must be computed back at the client after decryption. However, to execute a simple "U-shaped" execution plan would require us to bring back the *entire* emp table across the network as well to compute the rest of the query, which could be extremely costly for a large table.

A *maximal push* (MP) heuristic, which constructs a better performing plan, is shown in Fig. 6. Starting with a baseline plan that returns everything to the client for post-processing, a maximal push plan is constructed by pushing every possible operation to the server. This heuristic is presented in the original Hacigümüş et al. paper: "we would attempt to rewrite the query tree, such that most of the effort of evaluating the query occurs at the server, and the client does the least amount of work [8]." Not only can the MP plan minimize the server result set size (and the resulting network traffic), pushing every possible operation to the server also exploits the query optimizer, any indices, and likely much more powerful computing resources available at the DBMS server.

Note, however, that an MP plan is not always optimal: it would be cheaper to return the operands of a cross product and compute the cross product at the client, than to compute the cross product at the server and return the entire result! Thus, straightforward heuristics like the U-shaped plan, and the maximal push plan, cover a great deal of practical cases. However, a very general query planner for encrypted query processing must be sufficiently sophisticated to generate and evaluate *alternative* query execution plans. The requirements for a given scenario depend on the type of queries being executed, the size of data tables, and (as the next section demonstrates) local security requirements.

5 System Implementation

A working client-server system which can execute query plans over encrypted data can be a practical challenge to implement, due to:

1. *Query diversity:* The plan for one query might require cryptosystems which are: additively homomorphic, order preserving, and deterministic, whereas the plan for another query might require none of these. And a query might have operations which cannot be executed by any currently available partially homomorphic cryptosystem. Thus, some type of automated query planning is needed.
2. *User scenario diversity:* Interactions with users have shown dramatically different requirements. For example, some users will not use a cryptosystem unless it is on an approved list. Others, realizing they are exposing plaintexts, welcome the use of novel forms of encryption. User priorities may change as well, for example if threat levels are very high. Thus, although a planner is needed, it is difficult

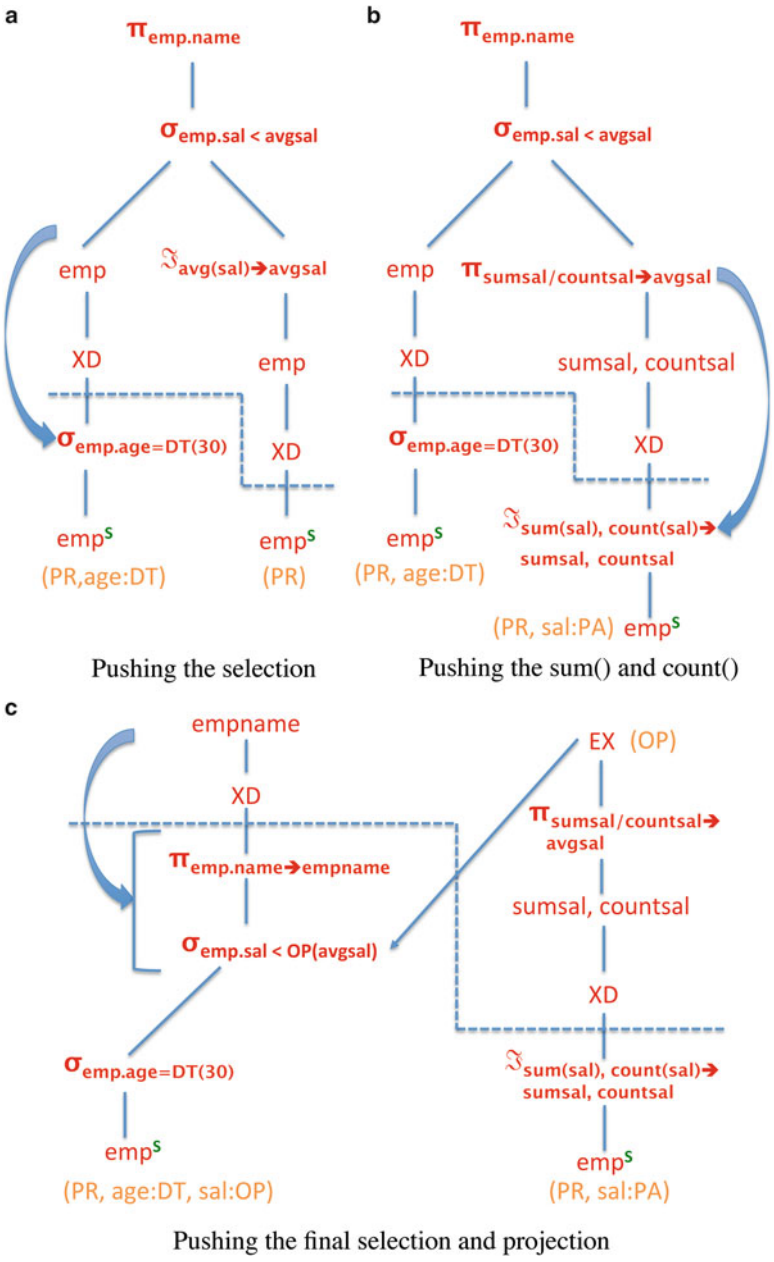

Fig. 6 Maximal push query plan

to *automatically* determine an encrypted query execution plan; unlike standard query optimization with its single focus on optimizing query performance, some user feedback about priorities is necessary to guide the generation of a good plan.
3. *Cryptosystem diversity:* Each partially homomorphic cryptosystem has a distinctive and complex profile of security, functionality, and efficiency features. There are frequently *multiple* cryptosystems of a given type (e.g., order preserving), and more are being published every day. Thus, the job of creating the encrypted database which will be installed on the server is a significant challenge.

Ordinary business users who simply want to execute their queries more security cannot be required to possess depth in both cryptography and database performance, or the promise of this technology will never be realized.

Given a plaintext database, and a query workload over it, users need to somehow generate a ciphertext database (involving various cryptosystems) and a set of query plans to execute their query workload over that database. To mitigate the complexity of system implementation, the DataStorm architecture, shown in Fig. 7, includes an intuitive multi-step workflow by which non-specialist users can accomplish these tasks. The first two steps (*design* and *migration*) help the user create an appropriate encrypted database. The third step (*execution*) enables the user to generate and execute query plans in a client-server architecture. These steps are discussed in more detail in the following.

1. *Design Time.* The *interactive database encryption advisor* (IDEA) automatically generates an *encryption map*, a mapping from plaintext columns to encrypted columns based on: (a) the original plaintext schema, (b) the user's plaintext query workload, and (c) the cryptosystems available in the encryption library. IDEA's initial mapping is based on generation of MP plans for each query. Users may interactively override IDEA's encryption recommendations (e.g., due to local security policy constraints), causing IDEA to suggest a new encryption map. IDEA uses a lattice-based visualization of encryption types to simplify interaction for those unfamiliar with cryptosystems, as described in [20].
2. *Migration.* Based on the final encryption map and the plaintext database, the *migration tool* creates the encrypted database on the server.
3. *Execution.* The *planner* takes a user's plaintext query as an input, and builds an execution plan, as described in Sect. 4.3. The plan object produced consists of (a) a set of queries executed at the server, (b) a set of queries executed at the client, and (c) operations to transform, decrypt, and encrypt data. The execution engine traverses the plan tree, sending queries to the client and/or server as appropriate, and assembles the final result.

Although DataStorm's architecture does not consist of commercial grade tools, their use has nonetheless made setting up a working client-server query system much easier.

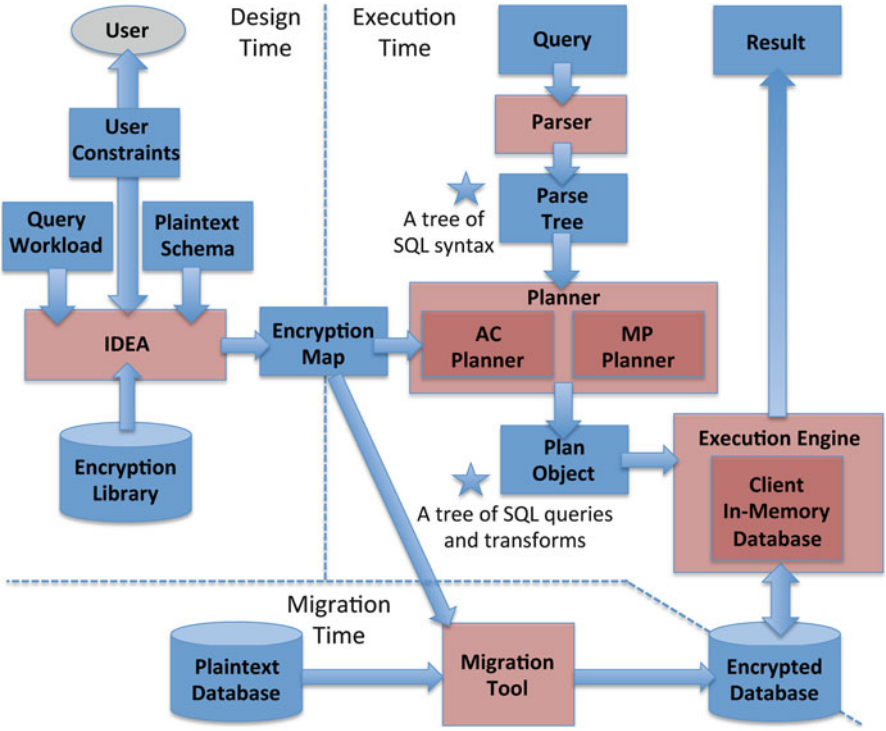

Fig. 7 DataStorm system architecture

6 Ciphertext Query Performance

A basic question potential users of encrypted database queries must ask is: how much does it cost in performance to execute a query over ciphertext, with respect to executing the same query over plaintext?

At a coarse level, there are three major categories of costs to consider:

1. *Client processing:* query planning, decryption of results, and post-processing.
2. *Network:* all transfers, especially returning ciphertext answers to the server.
3. *Server processing:* answering the ciphertext query.

Client processing times can vary widely, and are highly optimizable. For example, Paillier decryptions take a very slow 44 ms per integer. However, optimizations like hardware decryption (e.g., most new chips include AES decryption in their instruction set), and client-side ciphertext caching can speed decryption up significantly.

Network times are heavily impacted by the answer size. Note that this is true for plaintext queries at a remote server as well, however *ciphertext expansion* (the factor by which a ciphertext is larger than its corresponding plaintext) compounds

this cost. Network times are also impacted by the network's overall speed and by competition for network bandwidth, but often not in a predictable fashion due to network protocols which dynamically allocate bandwidth.

Despite hard-to-quantify variability, however, for both client processing and network transfers smaller answer sizes are strongly correlated with better overall query performance. This is ideal for a simple and common query like "Find the location of Helicopter 21". Even on a database of many terabytes, the answer size remains small, thus little client processing or network transfer cost is incurred.

In the following, we focus solely on a direct comparison of server processing speeds for plaintexts and ciphertexts. In Fig. 8, we compare a basic equality test query (like the Helicopter 21 query) on an indexed field. Identical server databases were set up in three sizes (10, 100, 1,000 K tuples), one in plaintext and one in ciphertext at each size. The ciphertext database used AES encryption (deterministic AES was used for the indexed field). Both query plans are identical, use the index, and return 4 % of the server database as the answer. To eliminate network variability, both databases were run on *localhost* using a standard Intel laptop with 3.5 GB of RAM running Postgres 9.1.4. After cache warmup, timings were computed as the average of 10 runs.

Fig. 8 Tests on equality query; times in ms

	PT total	CT total	CT/PT ratio
10K	0.6286115	1.0018130	1.59
100K	3.2718125	7.1076260	2.17
1000K	220.8141110	369.9193835	1.67

Figure 8 shows a stable ratio of ciphertext to plaintext execution times (between 1.59 and 2.17); ciphertext being around twice as slow. Decryption and query planning are a negligible fraction of these times. Much of this slowdown is attributable to the increased size of ciphertexts, resulting in more data pages being touched for the same query and data. A two-fold slowdown at the server is acceptable in many cases, especially for queries with a small fixed size answer (e.g., for web queries which populate forms) whose overall cost is dominated by the delay of communicating with a remote server over a network.

Figure 9 shows a similar experiment for a summation query; Paillier encryption was used for the field being summed. The database performed the homomorphic operation, a modular multiplication of ciphertexts, via a user defined aggregate function (UDAF) written in C in about 15 µs. In this case, working with ciphertexts is 67–82 times worse than plaintexts (still much faster than current fully homomorphic encryption algorithms). Note, however, that summation is an "embarrassingly parallel" operation, and commercial clouds make it possible to rent groups of compute servers for parallel computations. Paillier summations are thus an ideal candidate for speedup via cloud parallelism, and this ratio could be substantially reduced, even to unity. In addition, summation is a dramatically data reducing operation: terabytes operands can be reduced to a single answer, which incurs little

network delay. Thus, massive summations and the queries that rely on them (e.g., averages, business intelligence aggregates) are promising candidates for encrypted query execution as well.

Fig. 9 Tests on summation query; times in ms

	PT server	CT server	CT/PT ratio
10K	1.498111111	111.9696768	74.74
100K	14.01076768	1153.161051	82.31
1000K	169.0187677	11356.59457	67.19

7 Conclusions

In conclusion, challenges to query execution over encrypted data do exist, including individual query operations which cannot be executed over ciphertext, implementing a working client-server query execution system, and the performance of queries executed over ciphertext. However, as discussed in this chapter, these are well addressed by query planning, tools which assist the user with system implementation, and aiming for performance sweet spots, such as queries retrieving small objects and parallel summation queries.

As new cryptosystems are continually being developed and cloud services (e.g., parallelism on demand) grow, the future of encrypted query processing for cloud security is promising. Synergy between the information management and cryptography research communities, with their differing focus and priorities, is improving and has recently resulted in beneficial research. As that dialogue continues to grow, it will benefit this developing area. Further work is needed to develop commercial grade query planners/optimizers and system implementation tools, and efficient and secure cryptosystems with the partially homomorphic functionality to enable more types of queries to be processed at the server instead of post-processing.

References

1. ACM, *Test of time award*, www.sigmod.org/2012/awards_sigmod.shtml, 2012.
2. Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu, *Order preserving encryption for numeric data*, Proceedings of the 2004 ACM SIGMOD international conference on Management of data (New York, NY, USA), SIGMOD '04, ACM, 2004, pp. 563–574.
3. Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill, *Order-preserving symmetric encryption*, Advances in Cryptology – EUROCRYPT 2009 (Antoine Joux, ed.), Lecture Notes in Computer Science, vol. 5479, Springer Berlin Heidelberg, 2009, pp. 224–241.
4. DARPA, *The darpa program for programming comuptation on encrypted data (proceed)*, http://www.darpa.mil/Our_Work/I2O/Programs/, 2013.

5. Caroline Fontaine and Fabien Galand, *A survey of homomorphic encryption for nonspecialists*, EURASIP Journal on Information Security **1** (2007).
6. Craig Gentry, *A fully homomorphic encryption scheme*, Ph.D. thesis, Stanford University, 2009.
7. Craig Gentry and Shai Halevi, *Implementing gentry's fully-homomorphic encryption scheme*, Advances in Cryptology – EUROCRYPT 2011 (KennethG. Paterson, ed.), Lecture Notes in Computer Science, vol. 6632, Springer Berlin Heidelberg, 2011, pp. 129–148.
8. Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad Mehrotra, *Executing sql over encrypted data in the database-service-provider model*, Proceedings of ACM SIGMOD (New York, NY, USA), SIGMOD '02, ACM, 2002, pp. 216–227.
9. IBM, *Ibm homomorphic encryption library project on github*, https://github.com/shaih/HElib, 2013.
10. Intel, *Intel advanced encryption standard instructions (aes-ni)*, http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni/, 2011.
11. Witold Litwin, Sushil Jajodia, and Thomas Schwarz, *Privacy of data outsourced to a cloud for selected readers through client-side encryption*, Proceedings of the 10th annual ACM workshop on Privacy in the electronic society (New York, NY, USA), WPES '11, ACM, 2011, pp. 171–176.
12. A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone, *Handbook of applied cryptography*, Discrete Mathematics and Its Applications, Taylor & Francis, 2010.
13. Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan, *Can homomorphic encryption be practical?*, Proceedings of the 3rd ACM workshop on Cloud computing security workshop (New York, NY, USA), ACM, 2011, pp. 113–124.
14. Oracle, *Oracle advanced security transparent data encryption best practices*, http://www.oracle.com/technetwork/database/security/twp-transparent-data-encryption-bes-130696.pdf, March 2012.
15. Pascal Paillier, *Public-key cryptosystems based on composite degree residuosity classes*, Advances in Cryptology (EUROCRYPT '99), Lecture Notes in Computer Science **1592** (1999), 223–238.
16. Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan, *Cryptdb: protecting confidentiality with encrypted query processing*, Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (New York, NY, USA), SOSP '11, ACM, 2011, pp. 85–100.
17. Reuters, *German state ready to buy stolen bank data source*, blogs.reuters.com/financial-regulatory-forum/2010/02/04/german-state-ready-to-buy-stolen-bank-data-source/, 2010.
18. Amazon Web Services, *products page*, aws.amazon.com/products, 2013.
19. N.P. Smart and F. Vercauteren, *Fully homomorphic simd operations*, Designs, Codes and Cryptography (2012), 1–25.
20. Ken Smith, Ameet Kini, William Wang, Chris Wolf, M. David Allen, and Andrew Sillers, *Intuitive interaction with encrypted query execution in datastorm*, 2012 IEEE 28th International Conference on Data Engineering (ICDE), April 2012, pp. 1333 –1336.
21. Colin Tankard, *Advanced persistent threats and how to monitor and deter them*, Network Security **2011** (2011), no. 8, 16 – 19.
22. Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart, *Cross-vm side channels and their use to extract private keys*, Proceedings of the 2012 ACM conference on Computer and communications security (New York, NY, USA), ACM, 2012, pp. 305–316.