Trusted Cloud SQL DBS with On-the-fly AES Decryption/Encryption

Sushil Jajodia
George Mason U.
Fairfax, VA, USA
jagodia@gmu.edu

Witold Litwin
Univ. Paris Dauphine
Paris, France
witold.litwin@dauphine.fr

Thomas Schwarz
Marquette University
Milwaukee, Wisconsin
tschwarz@jesuits.org

***Abstract* — A Trusted Cloud Database System manages client-side encrypted cloud DBs. Queries may include encryption keys. The DBS decrypts/encrypts the data on-the-fly at the cloud. Plaintext is only in protected run-time variables. Stored data are by default probabilistically encrypted through AES. Any SQL queries are feasible, with negligible processing overhead and practical storage overhead. This is a major advance over the current alternative research proposals. We detail capabilities of a trusted DBS. We adapt SQL to client-side key management. Queries may remain usually almost as non-procedural as now. A prototype implementation appears easy.**

***Keywords*—cloud DB; client-side encryption; AES; trusted DBS;**

I. INTRODUCTION

A cloud outsourced database (DB) evidently welcomes the client-side encryption. The traditional security paradigm for a cloud DBs is that any cloud storage is simply (somehow uniformly) insecure against unwelcome reading. Sensitive plaintext data should never be at the cloud. In [2] we proposed a new paradigm. We trust the cloud Database System (DBS) secure at least for its run-time (temporary, fugitive…) variables with sensitive client plaintext (meta)data, encryption keys especially. We trust these protected, even against disclosure by insiders of the cloud administration. Only the (persistent) files in cloud storage are eventually insecure, as usual. The sensitive plaintext never gets to the storage. DBS wipes it out from the cloud at the end of a query processing.

Our paradigm seemed new for a cloud DBS. It comes nevertheless hardly out of the blue. Billions of users similarly trust their favorite industrial OSs and browsers. We all have no choice but to readily type-in various sensitive data: passwords, credit card numbers, you name it. Trusting firmly the values provided secure in some run-time variables only. We are many to refuse Google Chrome offer to turn these persistent. Likewise, we often do not trust the security of (persistent) outsourced plaintext files. Usually we provide the sensitive ones with the password encryption at least, [14]. Our paradigm simply transposes this overwhelming reality to the cloud DB arena.

Besides, the only difference between both paradigms is that the new one trusts the DBS at the cloud and the one at the client, while the traditional one trusts the latter only. In practice, both should be made by third-part industrial players. Both are also obviously Internet accessible. No reason to trust one installation more than the other. E.g., there is no reasons to trust SQL Server secure for run-time variables locally and *a priori* distrust it elsewhere. We all know also that most attacks are remote now. Life shows finally that the insider attacks are *a priori* about as likely at a client node as at the cloud.

We recall that some clients at present apparently trust also the cloud-side encryption, e.g., an option in Azure version of SQL Server 2016. The encryption keys reside then in cloud storage, *a priori* insecure, we recall, [9]. Our paradigm amounts to a fugitive only presence of keys at the cloud and only in run-time variables. Perhaps, even in the same variables as for the cloud-side encryption that also uses the on-the-fly encryption/decryption, we recall. The security brought by our paradigm appears thus inherently stronger than the current industrial offering[a].

To be practical, the cloud SQL DBS should evaluate most queries at the cloud. This especially concerns the select-project-join (SPJ) queries, usual for the online transaction processing (OLTP). If the cloud processing is not feasible, an impractical transfer to the client may be the must. This constraint appears plaguing the Online Analytical Processing (OLAP), with frequent value expressions. Under the traditional paradigm, to avoid the transfers to the client, a practical fully homomorphic encryption is necessary. Possibly – providing for queries to an encrypted (ciphertext) DB as rapid as queries to the plaintext DB. An intensive research over almost four decades under the traditional paradigm failed to deliver up to now. Only somewhat homomorphic schemes, e.g., the popular additively homomorphic Paillier scheme, appear fast enough for specific applications. See [2] for details.

Our paradigm led to a new homomorphic encryption schema called THE-scheme [2]. THE-scheme was faster than previous proposals, but limited to predefined finite domains. THE-scheme sends to the cloud some sensitive metadata, called *client secret* with each arithmetical query. The cloud DBS discards all the secret run-time values at the end of each query at latest.

In position paper [3], we proposed to explore our paradigm further. We now develop the ideas only outlined there. A query may carry the encryption/decryption key(s). Cloud DBS uses these keys to encrypt/decrypt the data on-the-fly, in some run-time variables, at the cloud. Cloud stored data are by default a ciphertext. The plaintext, the keys and any

[a] https://www.youtube.com/watch?time_continue=15&v=e4BS5A4yqp0 instructs how to disclose SQL Server cloud-stored AES key(s) and DBs encrypted with.

other run-time variables used by the query reside during the processing at the cloud in so-called *trusted* query execution environment only. Clients trust this environment protected, at least through the requirements already outlined. We discuss further possible protection tools for the trusted execution environment later.

A Select query uses the keys for the on-the-fly decryption of the ciphertext into a run-time plaintext processed at the cloud. An Update query with the key(s), uses furthermore the on-the-fly encryption producing the updated ciphertext. Likewise, an Insert query may bring plaintext and key(s) from the client to the trusted environment. The on-the-fly encryption there produces again the ciphertext for the cloud DB. Under these principles, any relational operation possible over a plaintext, functionally applies to the ciphertext. Whatever is the encryption scheme the cloud DB uses, it behaves functionally as under a fully homomorphic encryption.

We call *trusted* the (cloud) DBS designed as outlined. It manages *trusted* (cloud) DBs. We believe trusted DBSs promising research goal. Below, we start with the reference architecture. We stress that it is software-only, i.e., without any specific (trusted) hardware add-on box, unlike e.g., [14], [15]. We believe our approach the only practical at present for existing cloud nodes, likely many millions. Furthermore, we postulate that trusted DB should be AES256 encrypted. We speak accordingly about (trusted cloud) AES DBS and AES DB.

We consider that the client-side defined encryption key may have three levels of *granularity*. It can be a DB key, valid throughout the DB. Clients trusting finer granularity more secure can specify alternatively several keys, valid each only for a table, or even only for a column. The column granularity is backward compatible with MySQL. The DB key one is the basis for the cloud-side encryption by SQL Server. We adopt the DB key granularity as the default. Other keys eventually override the default. For finer granularities, the client may leave plaintext columns. This choice may help backward compatibility, but we discourage it. Frequency analysis of plaintext may endanger some ciphertext, reasonably secure otherwise. Finally, the keys encrypt by default data and metadata names, i.e., DB, table and column names.

Next, we propose the basic encryption schemes for an AES DB. These are deterministic or probabilistic. There are two modes of the deterministic encryption. We call them respectively *group* and *individual*. The former typically uses less storage. Both usually save storage with respect to our probabilistic encryption. The individual one is the only backward compatible, e.g., with MySQL. It can be also the fastest for an SPJ query. It may not need the decryption, hence the send-out of the keys altogether. This may be convenient for OLTP. The well-known caveat of deterministic schemes is the vulnerability to frequency analysis. Even a skilled client may overlook an attack opportunity. The group encryption should be usually less vulnerable that the individual one for the same data. A probabilistic scheme is free from this caveat. We take the stand that the probabilistic encryption is the default for an AES DB.

We then propose T-SQL : an SQL dialect for AES DBS. T-SQL data definition language (DDL) statements complete the usual SQL with clauses for encryption type, key granularity etc. Metadata names can get also client-side encrypted or may remain plaintext. We recommend the former choice. The metadata encryption shares the data encryption key(s). T-SQL data manipulation language, (DML), provide for non-procedural key management in queries. We show that T-SQL queries to ciphertext may remain almost as non-procedural as these to the same plaintext data. Actually, Create Database and Use Database statements may be the only affected. This implies defaults. T-SQL queries become unavoidably more procedural otherwise.

Next, we overview T-SQL query execution plan generation. We complete the legacy rules with new, specific to on-the-fly encryption/decryption. We then address performance analysis. We focus on decryption/encryption processing overhead, using recent AES benchmarks, [5] especially and on storage overhead. The latter depends on encryption mode. We show that on modern multi-core processors, the popular Intel I5 especially, both processing and storage overhead may be negligible for the group deterministic one. For the probabilistic encryption, it should still be so for the processing overhead, e.g., as little as 1.5¨%, benchmarked for the SUM aggregation. This feature makes in particular our proposal almost eighty five times faster than Paillier cryptosystem scheme. In contrast, the storage overhead should usually at least double. This is still however at least eight times less than for Paillier scheme.

The analysis shows the need for experiments specific to a cloud DBS. These remain a future work. We conjecture our proposal nevertheless the first generally practical. Especially, since heavy weights of cloud DB industry like Google with its version of MySQL and MS offer already the cloud-side AES256 encryption. Implying however by its sheer principle the security limitation we already hinted to for SQL Server 2016. MS proposes in fact also a client-side encryption. However, this one requires the cumbersome download/upload of entire tables to encrypt or even of the entire DB, to/from the client [13]. There are also other severe limitations, e.g., no equijoins on probabilistically encrypted columns, [9].

Next section presents AES DBS. We first discuss the reference architecture of a trusted DBS. We then discuss our AES encryption modes. Next, we define the extensions to SQL. We then discuss the query execution plan and the implementation issues. Section III addresses the processing and storage overhead. Finally, we conclude and discuss further work in Section IV. We stress that prototyping an AES DB appears finally perhaps surprisingly easy.

II. THE AES DBS

A. *Reference Architecture of a Trusted Cloud DBS*

Fig. 1, at end of the article, presents our reference architecture. We intend it for any trusted cloud DBS, regardless of the cloud DB encryption scheme. In other words, AES DBS is only a specific case of the architecture. The DB administrator (DBA) at a client (site) initiates the cloud DB through some upload. The DB should be strongly encrypted since in presumably insecure storage. Clients manipulate the DB through SQL queries. An SPJ query may end up manipulating a deterministically client-side encrypted ciphertext only, i.e., without any cloud-side decryption, as it will appear. We expect most queries nevertheless needing the plaintext data, hence a cloud-side decryption. Any such query brings the encryption key(s) for this purpose. An update may require a re-encryption. The cloud DBS performs the decryption/encryption on-the-fly, while reading/writing the DB to/from run-time plaintext values. All the data exchanged are encrypted for the transport using some usual scheme (SSL, RSA, Diffie-Hellman…). The DBS instantiates run-time variables with the metadata brought-in. It deletes any sensitive run-time content, i.e., the metadata and any retrieved/calculated plaintext data, at most by the query processing end.

The cloud processes the queries using some *core* DBS. This one is supposed to be some current plaintext cloud DBS with additional capabilities for client-side encryption management we detail progressively in what follows. The current DBS could be, e.g., MySQL with its AES encryption and decryption scalar functions. Our trusted cloud DBS is the core DBS supposed reinforced with or up-front characterized by security oriented *software (only)* engineering. The rationale for our requirement is that trusted DBSs should run on mass-produced cloud hardware, without then dedicated add-on hardware (trusted computing module (TCM), FPGA circuit…, [14], [15]). One may think of the protective component as of a vault, firewall…. Whatever the name is, it should protect the query execution so that clients may trust the run-time values secure in practice against exploits disclosing them. Our threat model for these is the popular honest-but-curious one, an insider being most likely culprit. The strong and whenever needed probabilistic encryption usually protects enough the stored data against such threat provided the attacker does not learn the encryption key. Exploits through malware disclosing the run-time variables while they contain the key(s), appear then the basic way to disclose the trusted cloud DB content.

We believe trusted DBSs a promising goal for cloud DBS research. Our requirement on sensitive data to be, at the cloud, only fugitive run-time vales, is open-ended. As mentioned, further technical base may be a *moving target defense*, [4]. Such defenses secretly shuffle the location of the program instructions and of run-time data. The results appear promising. The concept seemingly traces back to a decade old proposals for secure VMs, [1]. Alternatively, many, perhaps most, users, may be happy with a trusted DBS running on a cloud node simply as usual for software today. First, the run-time data of trusted DBS are supposed highly volatile only. This characteristic makes them usually trusted more secure. We already said that this trust appears universal for passwords released to major browsers. Recall also reminders of your favorite bank to change your password often. Next, the DBS should usually run in a VM. The security of client software is a prime goal for VMs for decades. The software protection is also gradually increasing in general through the re-engineering taking advantage of new capabilities of the basic hardware. E.g., Intel announced in 2013 its Software Guard Extensions (SGX) for the likely the most popular I5 and I7 processors. Their current versions with SGX are marketed since 2015. Probably there are already millions around. The processor guards then specific RAM areas called *enclaves*. No software outside an enclave can access enclave's content regardless of its privileges and CPU mode (ring3/user-mode, ring0/kernel-mode, SMM, VMM, or within another enclave). One may thus (reasonably) trust enclaves. A trusted DBS protected software environment (the vault) should be able to profit from these and from future novelties.

We do not know however any formal model quantifying the trust or protection against exploits. While the discussed and other techniques are promising, one should not expect a trusted cloud DBS secure once and forever. Trust is also subjective: a cloud DBS trusted by some may not be by others. Nevertheless, usual cloud DBs should not need the strongest possible protection. Recall, e.g., that IRS usually trusts its checks to paper envelopes. Also, stronger protection is likely to come for a cost. Altogether, one should expect the trusted DBS technology to face endless race between offerings, exploits, patches, new exploits and so on. As it is for our universally trusted OSs, browsers, VMs…

The trusted DBS may send back the selected data as ciphertext or plaintext or mix both. The ciphertext requires the post-processing decryption at the client. Unlike under the traditional paradigm, a client of a trusted DBS may alternatively request the entire decryption at the cloud. Avoiding the decryption burden may clearly make many clients happier. Even the popular browsers could then be clients of a trusted DBS. The traditional paradigm obviously excludes such clients.

As the result, two kinds of clients and queries appear, Fig. 1. A *smart* client is basically a client DBS able to locally encrypt/decrypt. A *simple* client, e.g., a popular browser has not this capability. A smart client may send a *ciphertext* query. Such query exchanges ciphertext between the cloud DBS and the client. It may return selected column ciphertext for final decryption at the client. An insert or update query may in turn provide the ciphertext prepared at the client. Finally the query may have a clause, e.g., a restriction of a column ciphertext to some (ciphertext) constant. The cloud detects a ciphertext query when it lacks a key to on-the-fly decrypt/encrypt a column referred to by the query. The query may also explicitly state that some column values should be dealt with encrypted.

A simple client emits in contrast the *plaintext* queries only. Those do not exchange any ciphertext. The cloud DBS interprets every constant of a plaintext query as a plaintext and every column name as referring to a plaintext. A plaintext query must include the key(s), to encrypt/decrypt every column for which there is some data exchange by the query. Every encryption/decryption is done on-the-fly at the cloud. A ciphertext query may avoid sending any keys, as it might not need the decryption/encryption on the cloud. Both properties may be considered advantageous by some users over a plaintext queries, decrypting/encrypting the related plaintext. Coming however, we recall, at the price of burden for the client. Also, as we already stated, many typical queries cannot be in practice dealt that way. Altogether, the advantage does not appear compelling. A smart client may of course formulate any plaintext query as well.

Ex. 1. (a) The following SPJ query is a plaintext one to the well-known Supplier-Part DB, usually called S-P. We recall it has the suppliers in table S, parts supplied in table P, and actual supplies in table SP. We discuss details of S-P for trusted AES DB soon. The query provides two table encryption keys, using SQL syntax extended for trusted AES we present soon;

(1) Select S.Name, SP.P# From S Key 'S123', SP Key 'P123' Where S.S# = SP.S# and SP.Qty = 200;

AES DBS recognizes that it is the plaintext query since it finds the key 'S123' valid for all S columns and the key 'P123' for all SP columns. It uses the keys to decrypt the final projection (S.S#, S.Name, P#) by default. We discuss later how DBS may actually evaluate the query.

(b) The next query could be a ciphertext one of some smart client. The hex constant is presumed 16B long and encrypting 200 into an AES bloc.

(2) Select Into Client.CacheDB.CipherTbl S.Name, SP.Qty From S, SP where S.S# = SP.S# and SP.Qty = Ox1234... ;

Query (2) does not carry any keys. AES DBS cannot decrypt any column the query names. It thus considers the hex constant in the query a ciphertext. Any of these properties makes the query a ciphertext one.

The query brings back the ciphertext. The result goes to CipherTbl table. The query dynamically creates it in some cache DB for the cloud DB at the client, named CacheDB. The client must decrypt the result for the user (application). E.g. through the following local query, calling the decryption scalar function, say AESD, using some encryption key, say '123', known to the client:

Select AESD (S.S#, '123'), AESD (S.Name, '123') From S;

As said, not sending any keys and economy of decryption time at the cloud could be an advantage over query (1). However, as it will appear, the time saved is negligible. There is also the time spent for decryption at the client then, in exchange. Next, as we stated and detail below, query (2) is correct only if the encryption of S# and QTY columns is deterministic and individual. It will appear that some storage gain may result, compared to a probabilistic encryption, especially for strings. Deterministic encryption however is then potentially not great in turn for S-P security. At least QTY here may be supposed vulnerable to frequency analysis. The cloud has no way also to figure out the intended plaintext for the restriction, i.e., 200. The correctness is solely in client's hands.

Notice finally also that, under AES encryption, similar ciphertext query would not be possible, if the clause on QTY was the QTY >200, a quite typical clause obviously. The cloud would need the key for '>" evaluation. Otherwise, the client would need to basically ask for the transfer of entire S (S#, SNAME) and SP (S#, QTY) at least to post-process the clause locally. This should be usually clearly a highly impractical operation.

Ciphertext query were the must for the traditional paradigm. The example illustrates that, altogether, their practicality for AES DBS does not appear compelling. Likely, the most frequent client of a trusted DBS should be a simple one, with plaintext queries thus. The design of AES DBS follows this rationale.

B. Encryption Modes for an AES DB

From now on, we consider the AES256 on-the-fly encryption/decryption only, as in Ex. 2. We consider three encryption modes for AES DBs. Two are deterministic and one is probabilistic.

1. *Individual deterministic encryption:* We recall that an encryption is deterministic, if the same plaintext, e.g., in the same column in different rows of a table, produces always the same ciphertext. The first mode encrypts every column plaintext as a distinct ciphertext. We qualify it of individual or, sometimes, of ungrouped.

We recall that AES is a symmetric basically deterministic scheme that encrypts plaintexts into 16B blocks. Modern workstation processors shifted to 64b arithmetic. We thus consider that a numerical value is up to 8B long. Each numerical plaintext gets encrypted thus to a single AES bloc. A string may have any length, hence lead to any number of blocs. If a plaintext does not fill a bloc, the client trivially pads the plaintext with a given constant to the block length before encryption. The cloud DBS knows the padding by default or infer it from the metadata we discuss in next section.

Fig. 2 illustrates the deterministic encryption modes with 128b wide AES blocs shown in plaintext ready for encryption. The upper blocs contain 64b wide integers. The left bloc has a single integer left padded with zeros, ready for the individual deterministic encryption. The lower two blocs are filled up with a string. The string is presumed encoded using the 8b UTF-8 symbols and left-padded with six spaces. These blocs are also ready for the individual encryption creating a two-bloc long AES ciphertext.

The individual encryption is the usual one. It is well-known in particular that the OLTP queries perform best if columns are encrypted accordingly. Provided the client-side decryption, the cloud DBS can use the ciphertext directly to evaluate the selections and equijoins of a typical SPJ query, i.e., without a value expression in particular. This could be the case of query (2) provided S# and Qty columns are individually deterministically encrypted. The client must be a smart one, we recall. AES DBS may only send the outcome of query (2) encrypted.

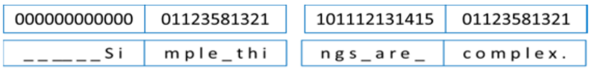
2. *Group deterministic encryption:* Our second deterministic encryption is *group* encryption. A *group* is a concatenated collection of plaintext column values, possibly, without any intermediate padding. AES DBS encrypts the whole group into a single ciphertext. In Fig. 2, the upper right 128b AES bloc groups two integers for their deterministic encryption within a single bloc. Saving the cumulated 128b long padding that would be added to their individual encryption, into two blocs instead.

Notice that for the group encryption, even if one of the values grouped in a bloc remains constant for selected rows, e.g., QTY = 200 as intended for our queries (1) and (2), the other(s) usually should change among these rows. Variable length strings may also shift somehow in the bloc or even to another one, the constant plaintext. Altogether, likely, a pseudo-random ciphertext within the blocs encrypting QTY = 200 should result. Unlike for the individually encrypted ciphertext, ciphertext queries like query (2) become impossible. AES DBS decrypts therefore on-the-fly every group encrypted ciphertext. The query must bring the key. AES DBS knows the encryption mode from the metadata. These also indicate how to ungroup the plaintext provided by the decryption.

The default and only choice for a group in what follows is the plaintext of all the column values formatted into a row by the DBS. The rationale is that the usual internal DB structure to be reused for an AES DB as we discuss later, should be row-based. An AES DBS may alternatively be columnar however. This could be when the core DBS reuses a columnar DBS, e.g., Monet or Vertica. The group could be then a plaintext columnar page.

The group deterministic encryption has no sense for the traditional paradigm, as the cloud needs the key. The rationale is usually negligible or no storage overhead with respect to the plaintext, unlike for the individual encryption. We have outlined also that group encrypted plaintext enhances the variability of the ciphertext among blocs in different rows encrypting same columns, with respect to the individual deterministic encryption of these columns. The group encrypted deterministic ciphertext should be in consequence usually less vulnerable to the frequency analysis.

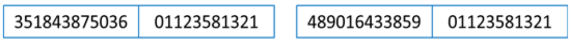
We recall that the latter may render insecure an individually encrypted column with a popular skewed value distribution, e.g., the zip code, [3]. The price to pay for the group encryption, we recall as well, is the systematic on-the-fly decryption. It will appear nevertheless that for AES, the processing overhead of this operation should be negligible in practice. Also, for an OLAP oriented AES DB, the simple SPJ queries, like query (2), should be exceptions. While the predominant there complex query should require the systematic decryption anyhow. All the discussed properties weighted, one may expect a large fraction of AES DBS clients preferring the group encryption over the individual.



000000000000	01123581321	101112131415	01123581321
______Si	mple_thi	ngs_are_	complex.

Fig. 2. Deterministic encryption for AES DBS. Left upper 128b AES bloc will individually encrypt 64b integer, left-padded with zeros. Right upper bloc will group encrypt two integers, saving the padding. Lower two blocs serve the individual encryption of a string.

Probabilistic AES DB encryption: We recall that any probabilistic encryption encrypts occurrences of the same plaintext into seemingly random (different) ciphertexts. This frees the ciphertext from the discussed vulnerability. It should be clear that AES is basically a deterministic encryption. Our third mode turns it into a probabilistic encryption for AES DB as follows. We fill each bloc to encrypt with the plaintext up to 64b large only, Fig. 2. Say putting it to the right half. The left half of the block contains client-side generated pseudo-random padding, i.e., an Initialization Vector. For a numerical plaintext, the right half would typically contain a 64b integer or real number. For a string, it may contain, e.g., up to 8 Utf-8 symbols.



351843875036	01123581321	489016433859	01123581321

Fig. 3. Probabilistic encryption for AES DBS. Blocs contain the same plaintext in their right half. Left half of each bloc is randomly padded. Ciphertext values will be necessarily different.

For the same plaintext in a bloc, AES may now generate up to 2^{64} apparently equally-likely ciphertexts. Encryptions of a plaintext value in different blocs should be all different. The result should preclude any frequency analysis. In turn, as for the group encryption, this one also impairs SPJ queries over the ciphertext. In particular, there is no practical way to evaluate a join. The on-the-fly decryption is also the must [3].

In other words, the result of query (2) should be usually folkloric for any AES DBS encryption mode other than the individual deterministic one. We therefore consider, by the way, that AES DBS prohibits such outcomes. Query (1) does not care about encryption type. It evaluates joins and restrictions over the on-the-fly decrypted plaintext anyhow.

C. *SQL for AES DBS*

As Ex. 1 hinted to, the client-side encryption under trusted DBS requires additional SQL clauses. We now propose such clauses specifically for AES DBS, i.e., its core DBS. The clauses extend both DLL and DML statements. New DML clauses let the client to specify key values in queries, e.g., as in Ex. 1. Through specific DDL clauses the client defines or modifies key granularity or encryption type. The client may also encrypt the metadata, the data names specifically, as we spoke about or choose not to for selected name. There are numerous default options. These aim on the least procedural plaintext queries to the AES DB. Possibly, no more procedural that would be equivalent queries to the plaintext SQL DB, except for the key(s) definition perhaps. Like the queries in Ex. 1.

We refer to the result as to *Trusted* SQL, (T-SQL). The client trusts indeed the security of the sensitive values provided. It would not make sense to define here T-SQL grammar. We use only basic SQL syntax that does not follow strictly any specific dialect. We discuss T-SQL only informally, through examples. S-P DB is our motivating canvas.

As usual, a T-SQL query may involve several statements. A key defined in a statement may apply to others in the query, but only to these ones. It's wiped out from the cloud after the query processing, we recall.

The DDL statements define metadata for AES DB catalogs. These are supposed the usual SQL catalogs extended with AES DBS encryption management metadata. T-SQL new DDL clauses concern first the Create Database statement. Any of these clauses applies to all the data and metadata by default. A clause may be overwritten however for a selected table in subsequent Create Table or Alter Table statement. First default for Create Database is that if the client does not provide the key, AES DBS stores the DB name provided in plaintext. Same for all subsequent data names, unless the client provides the key locally to a table or column definition. If the client provides the key in a DDL statement, then the key implicitly encrypts all the related data name(s) in the statement, unless the client states explicitly not to be so for specific data definition. The reason for the default is our belief that the metadata encryption is safer for the client. Plaintext metadata may disclose some knowledge of the DB evidently. For obvious practical reasons, we limit this encryption to probabilistic only. As for DML queries, the key itself remains in secured run-time variable only and is wiped-out once DBS processes the DDL query.

With respect to data, the probabilistic encryption is the default as well. The client may override it by either type of the deterministic one. The scope is according to the key granularity.

Ex. 2. The following query may create S-P with its tables. All tables share the DB key ‘S-P123’. All data names are encrypted accordingly. String symbols are, for change, in Unicode, with 16b symbols, and integers are 64b wide. All S-P column values are probabilistically encrypted.

Create Database ‘S-P’ Key ‘S-P123’;

Create Table S S# Char (6), SNAME Varchar (40), STATUS Int, CITY Varchar (50). Primary Key S# :

Create Table SP S# Char (6), P# Char (6), QTY Int, Primary Key (S#, P#) ;

Create Table P, P#..., PNAME…, COLOR…, CITY…, WEIGHT… Primary Key P#;

The Create Database statement has the T-SQL specific key definition clause. As one result of this one, the name S-P gets probabilistically encrypted for the AES DBS catalogs. The statement is the only in the query with any T-SQL specific clauses. One effect is that all the other data names declared in the query are also stored in AES DBS catalogs probabilistically encrypted. The key related clause makes the above Create Database slightly more procedural than the corresponding plaintext statement. In contrast, all the Create Table statements are as their counterparts for the plaintext S-P. The former and the latter are thus equally non-procedural for the clients.

Notice that with the above S-P definition, query (2) is impossible. By default, all columns would be probabilistically encrypted. To avoid it, DBA should explicitly define above the columns referred to in Where clauses of (2). E.g., as S# Char 12 IND what would stand for individual deterministic encryption. Also Create Database Key clause should become the self-explaining Key ‘S-P123’ DATA ONLY. But this would leave in turn in plaintext in the catalogs by default not only S-P, but all data names. Unless some Create Table overrides this default for itself. E.g., the following one: Create Table S Key ‘S123’, S# Char 12…. would (probabilistically) encrypt the name S and all the column names in table S. It would leave all the other data names in S-P in contrast here by default in plaintext. @

In what follows we suppose that operationally, AES DBS produces the ciphertext using the AESE (X, K, E) scalar function. Here, X is a column name or plaintext to encrypt, K is the key and E is the encryption type for X. K is optional. First, the function may reuse the key that the query provides elsewhere. Next, the purpose of the whole statement may be only to explicitly state that ciphertext stored in column X should be dealt with as is. Finally, E is also optional. Its absence means the default probabilistic encryption for X or values within. If E is there, it is E = IND or E = GRP. It stands then for the individual or group deterministic encryption. Here, the (probabilistic) encryption of, e.g., S-P name could alternatively result operationally from AESE (‘S-P’) reusing ‘S-P123’ as the key.

Alter Table DDL statement of T-SQL is supposed extended similarly. Using it for AES DB, DBA may alter all encryption related new features. E.g., DBA may decide to protect PNAME column specifically, by altering the default definition as follows:

Alter Table P Key ‘S-P123’ As Alter PNAME To PNAME Key ‘PNAME123’ ;

Notice that this could be a costly operation. It re-encrypts both the column and its content. Actually, most encryption related alterations would do and cost similarly.

Finally, Drop Table T-SQL statement is the usual plaintext one extended at most by the KEY clause only. DBS needs the key to locate the table in the catalogs whenever its name is encrypted as well. What should be the wise practice, as we stressed.

The T-SQL DML statements also the usual ones extended with the Key clause. In T-SQL Use statement this clause provides the value of the DB key. This one may apply by default to all the subsequent statements. Actually, these may also be the DDL statement(s) Alter or Drop. In Select DML statement, the clause adds to the usual ones in From clause. It may follow a table name, applying then by default to all its columns. Preceded with ‘,’, it may follow the list of all the tables, providing then the DB key. This applies to all the columns named in the Select. Except for the columns of a table with its own key provided in the statement as well or for any column with its own key. The column key may be provided also through Key clause, but may result the invocation of the already addressed AESD function. This choice is basically backward compatible with MySQL.

For the Insert DML statement, Key clause may similarly appear after the table name. In contrast to Select, it may also follow a column name. It then applies to every tuple that follows within the usual Values clause. For Update statement, it may follow the table name or any column modification clause. Finally, it may follow the table name in Delete statement.

Ex. 3. Here are a few almost self-explanatory T-SQL queries to S-P as defined through Ex. 2. Queries may have comments, as usual after ‘/*’.

Use S-P Key ‘S-P123’ ; Select S.* From S, SP Where S.S# = SP.S# and SP.QTY > 200 ; /* Key value from Use applies by default to all columns in Select statement that is then simply the usual SQL one.

Use S-P; Select AESD (PNAME, ‘PNAME123’), S.* From P Key ‘P123’, SP, S, Key ‘S-P123’ Where P.P# = SP.P# and S.S# = SP.S# and SP.QTY > 200 ; /* S-P name is in plaintext, P has its own table key, except for PNAME, Select statement provides the DB key value for SP and S.

Use S-P Key ‘S-P123’; Update P Key ‘P123’ Set

COLOR = 'Red', PNAME Key 'PNAME123' = 'Big Wrench' Where P# = 'P3' ;

Insert P Key 'P123' ([P#], PNAME Key 'PNAME123' Use S-P Key 'S-P123';) Values ('P8', 'Nut'), ('P9', 'Wrench') ;

Use S-P Key 'S-P123'; Drop P Key 'P123' ;

D. *T-SQL Query Execution Plans*

As usual, to evaluate a T-SQL query, AES DBS should generate some execution plan, possibly optimizing processing time or storage. We see such plan as the one generated as if the query manipulated the plaintext data only, with additional decryption/encryption operations blended in. The latter facet is subject to its own basic and amelioration rules. As usual, the ameliorations aim on query performance, succeeding often, but not always. It would not make sense here to define rules specific to the encryption/decryption formally or exhaustively. We only sketch a few informally. Here are two basic rules for the query plan generator, one could say, naïve.

1. Let A be a column referred to in the query. For every A, if the query provides the key for A, then decrypt or encrypt on-the-fly every manipulated value of A. Otherwise, manipulate every A value as a ciphertext. Call further rules to decide whether to decrypt or encrypt A values, or decrypt then re-encrypt them. If A is a ciphertext, call further rules to decide whether query formulation allows it to be processed altogether or the plan generation should be halted.

We skip the further rules to which rule (1) refers as simple but tedious. In short, they should imply a decryption for a Select query. Also basically, they should imply an encryption of values provided for Insert and Update queries. A Select clause in these queries may however require a decryption, perhaps followed by an encryption, perhaps of even a different type. Details depend then on the column definition in AES DBS catalogs. Finally the Select Into query may require the same treatment.

2. Let the query have a restriction clause A θ C, where C is a constant and $\theta \in \{=, \geq, \leq, <, >, <>\}$, as usual for θ. If A can be decrypted then consider C plaintext constant. Otherwise, search for A in the catalogs. If A is declared plaintext, then consider C plaintext as well. Otherwise, consider C a ciphertext.

The following two rules are specific to the individual deterministic encryption.

3. Regardless of basic rules (1) and (2), let column A be individually deterministically encrypted and subject to the restriction by ORed equality clauses only: A = C1 Or A = C2 Or…., where each C is a plaintext constant. Then encrypt each C, and match visited A values as they are in AES DB.

4. Likewise, if two individually deterministically encrypted columns A and B are in equijoin clause A = B, then match the manipulated A and B values as they are stored.

Rules (3) and (4) are examples of the amelioration rules specific to AES DBS encryption/decryption in the query plans. The basic rules would systematically decrypt the visited values. Usually, these or others amelioration rules save some processing time, but not always. Other clauses may imply that all the matched values are finally decrypted anyhow.

There are also rules for value expressions. Likewise, there are basic rules for correctness of the query with respect to its decryption/encryption facet. These are the rules we referred to in rule (1). We also talked about, while discussing why query (2) may only support the individual deterministic encryption. These rules may consequently halt the plan generation and the query processing. Or, at least, may issue a warning SQL Error code. Next, there should be obviously also the implementation dependent rules, e.g., related to indexing. One may expect these to attempt to ameliorate the basic rules sometimes. Altogether, as for a plaintext DB, the plan generation appears a complex matter. We leave the subject for future work.

The following example shows a few plans for queries to our encrypted S-P.

Ex. 4. Consider query (1) to S-P created encrypted with DB key 'S-P123' and defaults only. All columns are thus probabilistically encrypted. The typical plaintext only plan would start with processing the restriction on QTY, followed by the processing of join over previously selected rows and of the final projection. The AES DBS specific facet will apply he basic rules (1) and (2) only, decrypting thus on-the-fly all the visited stored values. Amelioration rules would come to play only if QTY or both S# columns were individually deterministically encrypted.

Consider now query (2) from the same example. The plaintext plan would be the same. The AES DBS specific facet should first include the test of query correctness, i.e., whether all S# and QTY columns are really individually deterministically encrypted. If not the plan could issue the critical error and halt. Otherwise, it can continue, generating for the ciphertext the plan otherwise for a plaintext DB, same as for query (1). It may also perhaps issue a warning about potential issue with C we spoke about.

Consider finally the query: Use S-P Key 'S-P123'; Select STD (Qty) from SP ;. Suppose SP stored row-wise as usual. Then, the optimal plan for the plaintext S-P usually would first have the value expression rewritten to SQR ((AVG(QTY**2) – AVG (QTY)**2) at least. Perhaps, AVG would be decomposed even further. Then, the plan would perform a single scan of entire SP, processing every QTY value once and only once (to process VAR without the above rewrite would need a double scan). As outlined, AES DBS should generate this plan with the additional rewrite, replacing on-the-fly each visited QTY value V with AESD ('V', 'S-P123').

Notice that we are not aware of any somewhat homomorphic encryption able to evaluate this query entirely at the cloud at present, (actually our THE scheme would do, provided one first tabulates encrypted SQR function for the domain of QTY). The fastest current solution for bigger SP at present would be likely the lengthily upload of the whole SP or of, at least, its QTY column, to the client for the plaintext processing there. Likely, it would be the same processing as the one above generated by AES DBS at the cloud. But then, the cloud would send back a single value only, i.e., the final one.@

E. *Implementing AES DBS*

Creating AES DBS from scratch seems a utopia. Rather the core DBS, Fig. 1, should embed a legacy DBS. This one should provide for encryption/decryption, at least through User Defined Function (UDF). A 2-layer functional architecture appears accordingly for AES DBS, Fig 4. Both layers should process queries only within the trusted query execution environment, as the reference architecture stipulates. The legacy DBS code might need an update, invaliding perhaps any code saving plaintext run-time values, e.g., into a log file.

The T-SQL layer is the client interface. Functionally, it is the only to communicate with the clients. It receives T-SQL queries and sends back the results. The legacy DBS manages the DB and operationally processes all the queries. It must be provided with encryption/decryption functions. It is the legacy DBS indeed that ultimately generates the query plan and executes it using the on-the-fly decryption/encryption internally. MySQL provides such functions as the native scalar ones for the individual deterministic encryption decryption and for this encryption mode of AES DBS only. We are not aware of any other legacy DBS with such functions. At least the remaining encryption/decryption modes for MySQL and all the modes for another legacy DBS must thus be added. The general way is to create UDFs. Most major DBS support UDF libraries. Perhaps even, the most practical way to implement the T-SQL layer itself is to create an UDF.

Any legacy DBS will use some SQL dialect. Functionally, T-SQL layer should reformulate T-SQL queries in this dialect to get the required legacy services. At this stage we can only outline the main services needed:

- AES DB metadata management. AES DB needs its own catalogs used at T-SQL layer. These should contain the original AES DB scheme, including encryption metadata, e.g., the one of S-P we discussed. To implement these catalogs, the easiest is probably to internally create and manage them as tables by the legacy DBS.
- Internal scheme management. Legacy DBS needs its own scheme for AES DB. This should and may only contain the usual characteristics of a DB managed by this DBS. These do not concern the encryption metadata, in first place. In particular, the original plaintext column data type at T-SQL layer should usually be different from the encrypted one. In general, the latter should be Blob () or Varbinary () for AES blocs. The former can be any, e.g., Float for a numerical column. The on-the-fly decryption using any existing algorithm of our knowledge produces a Blob () string. This one needs *in fine* get converted to the original plaintext type. T-SQL layer has to handle the details of the internal scheme generation and of the decrypted plaintext value type generation. Details may be surprisingly tedious at present, e.g., for MySQL, [16].
- For the probabilistic or individual deterministic encryption, the internal DB scheme may have a column for each original one. In contrast, for row based group-deterministic encryption, it seems that most practical is to service a single internal column per original table. Encryption/decryption should be able to scope any individual column anyhow, i.e., select, insert or update an individual column value. As we said, the legacy DBS services for T-SQL providing most or all of these capabilities remain to be designed.
- Altogether, implementing an often likely to be sufficient AES DBS architected as in Fig. 4, appears surely tedious, but likely easy. Nowadays, as we discussed, one may indeed reasonably trust the security of run-time variables in a major legacy DBS, e.g., perhaps already or soon all in enclaves. MySQL appears 1st choice for the legacy DBS. Its code is public. One can inspect the run-time management & adapt to AES DBS any perhaps subsisting need. The AES encryption/decryption we spoke about consists of SQL scalar functions AES_ENCRYPT() and AES_DECRYPT(). These should serve the AES DBS AESE ( ) and AESD () functions we discussed, for the individual deterministic encryption at least. The RAND function should help with the AES DBS probabilistic encryption. Finally, MySQL not only supports UDFs to create the AES DBS specific functions, as we discussed already, but these functions may be created alternatively as so-called native function or stored functions. Each function type has its pros and cons, widely discussed in MySQL manual and blogs.

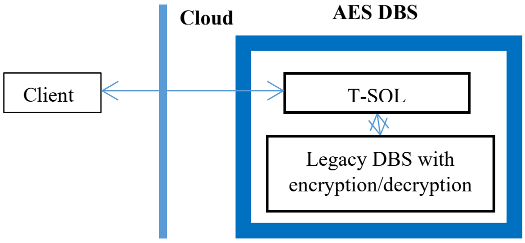


Fig. 4. The 2-layer Functional Architecture for AES DBS.

- More generally, as said already, our goal seems feasible for any existing plaintext cloud SQL DBS supporting UDFs. Some major cloud SQL DBSs do not have this capability, e.g., Google Cloud. Otherwise, SQL Server seems a runner up candidate. Its code is not public, but it has native AES encryption/decryption, although cloud-side only, [9]. It encrypts/decrypts DB file pages, hence can be seen as providing a group deterministic encryption, although only this one and not our default one. In each case, a usual browser should suffice as a simple client. These guidelines are likely **the way** to start practicing our proposal.

III. PERFORMANCE ANALYSIS

A. *Processing Overhead*

What matters most for our proposal is the overhead of on-the-fly AES256 decryption and encryption at the cloud, induced by a query to the ciphertext in AES DB. There are several recent benchmarks of AES256: [5], [6], [10]. They consider the popular multi-core processors. Most of them naturally consider the ciphertext in RAM cache or disk. The encryption/decryption result can be measured as sent out (or simply dropped) or with every ciphertext/plaintext written back to RAM. The former measure is the basic one for Select queries. The latter one adds up for a systematic Update query. For instance, - adding 10% to every price in some table. The main measure is the number of encrypted/decrypted bytes per second (MBs). The decryption can be little faster than encryption.

The encryption can be entirely in software. Two popular public-domain algorithms are Truecrypt and Twofish. The former uses the Rijndael's algorithm that won NIST competition. The latter was a competitor as well, but appeared slower, for 64b processors especially, [8], [7]. Within Intel I5 processors family, several CPUs have instructions for AES encryption/decryption hardware acceleration. These are so-called AES-NI instructions. Some Xeon CPUs also do, e.g., Xeon X5690. Pricing with or without NI is in practice the same. Truecrypt 7.0a takes advantage of AES-NI. Twofish does not. The benchmarks show that AES-NI effectively speeds up the processing. Results vary among benchmarks.

For our purpose, we concentrate on Intel I5, seemingly the most used. According to [G2], the bulk raw (straight) encryption using the Truecript 70.a without RAM re-writing provides the impressive 1900 MBs encryption/decryption rate. Twofish leads to 273 MBs "only". More recent results in [5] for a wide range of CPUs, report for I5 661 CPU specifically, an even more impressive 4133 MBs rate. Presumably, - with Truecript 70.a as well. Results for other CPUs vary, the slowest being 317 MBs and the average being 1.9 GBs. For the deterministic encryption this leads up to 516,5M for AES-NI and to 34M for Twofish pf plaintexts/ciphertexts processed per second. To decrypt 100K values, e.g., for sum SUM function, may take thus as little as 0.2 ms with Truecript 70.a (and 3ms with Twofish). For our probabilistic encryption, the timing multiplies by two.

The processing naturally slows down when every decrypted/encrypted value is written back to RAM. Only [6] reports the related experiment, using Truecript 70.a. It performed at 763 MBs. However, the plaintext writing rate was then limited to 880 MBs. Encrypting led thus to 13% overhead only. Per value rate is about 100 - 50Ms and 100K value decryption takes 1-2ms for our encryptions. How the RAM writing impacts an SQL query depends obviously on the aggregates and clauses (GROUP BY, ORDER BY TOP…). Nevertheless, Select queries serve generally to produce few values only. An aggregate is expected to read perhaps very many tuples, but to produce a few only. The writing timing of these results should therefore very little impact of the read-only results above. It is not the same for a large update. We come back to the issue below in SQL specific analysis.

The bulk transfer rate from hard or SS (flash) disk is disk technology dependent. They appear to be at most 150MBs in practice (SATA-3 interface). The random access times are well-known, i.e., about 10ms in practice for a hard disk and 1ms for an SSD. The AES overhead appears negligible, allowing for the real-time processing (Aegis Padlock disks).

The results for the decryption/encryption of selected values or of small groups of those are slower than for bulks. The reason is so-called *key set-up* time. Experiments show nevertheless that the key set-up may cost for the Rijndael's algorithm as little as 15% slow-down [8]. An SQL query is typically expected to do a bulk search. We thus neglect this (small anyway) specificity in what follows.

Finally, the AES algorithms above discussed appear programmed in assembly language. Use of a higher-level compiler, e.g., Java, may have a severe impact. For Oracle JDK 1.7, Intel reports thus at best 80 MBs rate, for AES-NI, [I5]. This is 10M values per second for us, "only". The overhead goes up to 10ms per 100K decryptions. We do not analyze this DBS seems natural. The subject requires nevertheless a specific study.

B. Storage Overhead

As we hinted, the AES DBS deterministic ciphertext may have negligible or no storage overhead with respect to the plaintext. The latter should occur mainly for the group deterministic encryption, e.g., for our SP table, encrypted into two blocs per row. In practice, a group is likely to have some leftmost or rightmost padding. This could be the case of our S table. Usually, a row has hundreds bytes at least. The padding incidence should be thus negligible. The exact overhead per row, table or entire DB, should be DB specific. The individual deterministic encryption should obviously usually increase the padding, hence the overhead. The actual value is again DB specific. For SP we would have three blocs with 16B of padding total, hence 33% overhead.

The probabilistic encryption should be the most storage thirsty. It at least doubles the storage of every plaintext column. It carries the overhead of 100% for numerical columns. The overhead may be still higher for strings, especially short ones, say a dozen of bytes at most. For our SP, we would end up with five blocs hence 150% overhead. Such cases appear nevertheless unlikely. One may expect the overall usual overhead for a table or DB still close 100%. It could be the case for S and P tables, because of longer strings they should typically contain. The actual overhead should be again obviously AES DB specific.

An AES DB may thus fit at best about the same storage as the equivalent plaintext DB. At worst, it may require about twice that storage. Especially, if the client chooses the default encryption that is the probabilistic one we recall. The worst case is still practical, e.g., it is about the overhead of the popular DB mirroring. However, it may be of concern to clients paying the cloud storage. The client may then perhaps rethink the default choice of best possible randomness for ciphertext. Perhaps, s/he can after all trust the limited randomness of the storage-savvy group deterministic encryption. Notice nevertheless that, even for the default choice, AES DB overhead remains far smaller than a typical one of probabilistic homomorphic schemes. E.g., Paillier scheme requires at least 1Kb ciphertext per plaintext integer. Thus about eight times more than AES DB in worst case.

C. Query Processing

The basic measure here is the overhead of on-the-fly decryption for the otherwise plaintext execution plan. Such decryption may deal with GBs. The study of the read/write speeds for a ciphertext and a plaintext above showed only 13% overhead then. This suggests that even for an AES DB in RAM storage, the overhead of the on-the-fly decryption, including auxiliary data, e.g., indexes, could be usually limited to a dozen of percent or so as well. It should become negligible for AES DB on solid state or hard disc, with an order of magnitude at least slower read/write speed thus. Another measure can be the AES DBS query execution speed with respect to the same query over a homomorphic encryption. The Select SUM(x)… query adding 100K plaintext values at the cloud server using Paillier, reputed the fastest traditional additively homomorphic scheme, needed 14ms for plaintext additions, and 1153ms for Paillier additions, followed by 50ms decryption time at the client [11], [12]. Our scheme adding up at best 0.2ms to the plaintext time, its overhead could thus be as low as 1.5%. This result appears thus even better than 13% above, matching actually AES overhead of reads by SQL Server on AMD processors, of 1.91 – 2.86%, [17]. All other benchmarks we cited would cost a few milliseconds at most. With respect to the homomorphic encryption, finally, AES DBS could thus be up to eighty five times faster.

IV. CONCLUSION

The on-the-fly decryption/encryption by a trusted cloud DBS, appears the first generally practical architecture for a client-side encrypted relational cloud DB. It roots in the intensive research for almost four decades. It is the only at present potentially offering to large public any functional capabilities, current or future, of a plaintext relational DBS. It is also the only allowing for simple clients. The on-the-fly decryption/encryption run-time overhead should be negligible for an AES DB, whether it uses the deterministic or our probabilistic encryption. The queries can be also expected about two orders of magnitude faster than for known homomorphic schemes. In addition, the functional and processing capabilities of all those schemes perhaps suffice for selected applications, but are largely limited with respect to our scheme.

Our study has shown several directions for further work. We have highlighted some, see [3a] for more. The main conclusion is that building an often likely sufficient AES DBS appears rather easy. Nowadays, as we discussed, one may indeed reasonably trust the safety of run-time variables of major SQL DBSs, e.g., perhaps already or soon in enclaves. Free MySQL appears the 1st choice as already stressed. Likewise, SQL Server seems the runner up candidate. In each case, a browser suffices to run plaintext queries as simple client. Let us stress again that all this seems *the way* to start practicing AES DBSs.

V. REFERENCES

[1] Holland, David A., Ada T. Lim, and Margo I. Seltzer. 2005. An architecture a day keeps the hacker away. 2004 Workshop on Architectural Support for Security and Anti-Virus. Boston, MA. Special issue,

[2] Jajodia, S. Litwin, W. Schwarz, Th. Numerical SQL Value Expressions over Encrypted Cloud Databases. 8th Intl. Conf. on Data Management in Cloud, Grid and P2P Systems (Globe 2015). In DEXA 2015. Springer, 2015.

[3] Jajodia, S. Litwin, W. Schwarz, Th. On-the fly AES256 Decryption/Encryption for Cloud SQL Databases. Position Paper. BDMICS 2016, Porto (Sept. 2016), 5p, IEEE, publ., to app. (a) Extended Preliminary Version: Lamsade Res. Rep. June 2015, 13p.

[4] Jajodia & al. eds. Moving Target Defense. Advances in Information Security. Vol 1 & 2. Springer, 2011-3.

[5] SiSoftware AES256 Benchmark. 2015. http://www.sisoftware.co.uk/?d=qa&f=cpu_vs_gpu_crypto&l=en&a=

[6] Grant. Hardware AES Showdown - VIA Padlock vs Intel AES-NI vs AMD Hexacore. 2011. http://www.grantmcwilliams.com/tech/technology/387-hardware-aes-showdown-via-padlock-vs-intel-aes-ni-vs-amd-hexacore

[7] Dandalis & al. A Comparative Study of Performance of AES Final Candidates Using FPGAs In: Cryptographic Hardware and Embedded Systems – CHES 2000, 2nd Intl. Workshop. Worcester, MA, USA, 2000. Lecture Notes in Computer Science, Springer (publ.).

[8] Schneier, B. & al. AES Performance Comparisons. http://csrc.nist.gov/archive/aes/round1/conf2/Schneier.pdf.

[9] Hammer, J. Szymaszek, J. Overview and Roadmap for Microsoft SQL Server Security. Microsoft Ignite, Chicago, Apr. 2015. https://channel9.msdn.com/Events/Ignite/2015?t=mission-critical-oltp .

[10] Intel. AES-NI Performance Testing on Linux*/Java* Stack, 2012. https://software.intel.com/en-us/articles/intel-aes-ni-performance-testing-on-linuxjava-stack#aes256

[11] Smith, K., Allen, D., Sillers, A., Lan, H., Kini, A.: How Practical Is Computable Encryp-tion? http://csis.gmu.edu/albanese/events/march-2013-cloud-security-meeting/04-Ken-Smith.pdf

[12] Smith, K., Allen, M., D., Lan, H., and Sillers, A. Making Query Execution Over Encrypted Data Practical. Secure Cloud Computing. Springer, 2014, Jajodia, S. & al eds, 173-190.

[13] Szymaszek, J. Encrypting Existing Data with Always Encrypted. SQL Server Security Blog, July 28, 2015 http://blogs.msdn.com/b/sqlsecurity/archive/2015/07/28/encrypting-existing-data-with-always-encrypted.aspx

[14] Trusted Computing Group. www.trustedcomputinggroup.org

[15] Arasu & al.Transaction Processing on Confidential Data using Cipherbase. 31st Intl. Conf. on Data Engineering, ICDE-15.

[16] Ahsan, M. Encrypt MySQL data using AES techniques http://thinkdiff.net/mysql/encrypt-mysql-data-using-aes-techniques

[17] Garrison, R. Performance Testing SQL 2008's Transparent Data Encryption. Database Journal, April, 2009

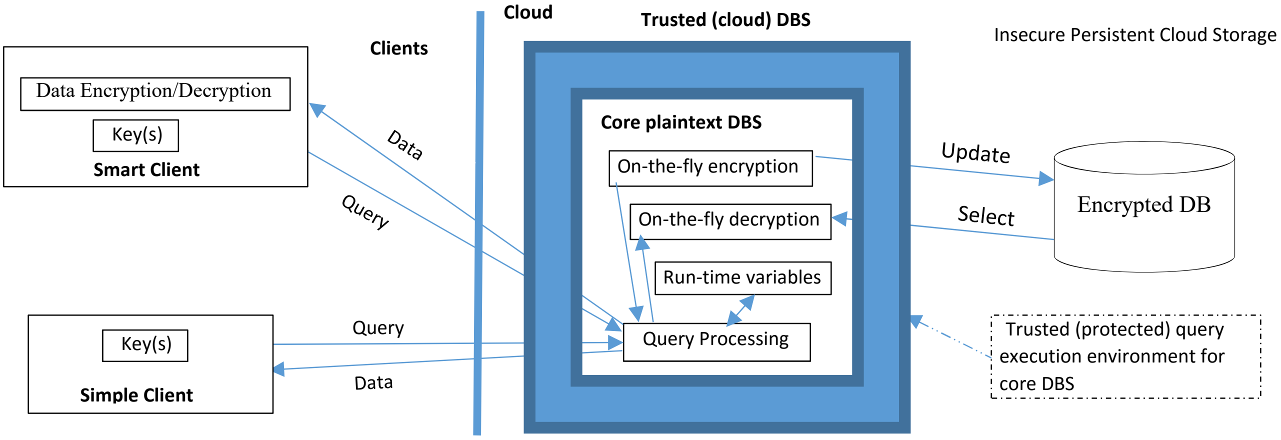


Fig. 1 Reference Architecture of Trusted Cloud DBS