On-the-fly AES256 Encryption / Decryption for Trusted Cloud SQL DBS

Position Statement

Sushil Jajodia
George Mason University
Fairfax, VA, USA

Witold Litwin
Université Paris Dauphine
Paris, France

Thomas Schwarz
Marquette University
Milwaukee, Wisconsin, USA

***Abstract*—We propose client-side AES256 encryption for a cloud SQL database. We rely on the safety of DB run-time values e.g. through a moving target defense. This applies to keys sent by the client with a query as well as decrypted values needed for query evaluation. We trust that the DBS clears these values at or before the end of query evaluation. Query results may be in cipher for certain clients or in plaintext e.g. for a browser. The scheme offers all the functionality of a SQL DBS that does not use encryption, but stores all data in encrypted form. An implementation should be straightforward.**

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

I. Introduction

A database (DB) outsourced to a cloud can obviously benefit from client-side encryption. The universal consensus holds that cloud storage does not offer confidentiality and that sensitive data should only be stored in encrypted form. In [2], we propose a new paradigm based on trusting the confidentiality of volatile data in the cloud Database System (DBS), even against insiders with administrative privileges, while persistent values, namely the stored data, are only made safe by encryption.

While this seems to be a new paradigm for cloud DBS, in fact billions of users rely on it when using popular OS and browsers to which they entrust passwords, credit card numbers and other highly sensitive values at run-time. A smaller amount of users entrust their system with storing these data permanently unless they are stored in encrypted form secured by a password [14]. Our paradigm applies this universal practice to cloud DB.

The traditional way of building databases for the cloud stores data in encrypted form, downloads encrypted data to the client, and only there decrypts them for query evaluation. This architecture implicitly only trusts the DBS at the client, which is almost always a third-party product and which obviously has network access. Our paradigm requires only additional trust for DBS in the cloud. There is however no reason to trust an SQL server or any other commercial product for *local* runtime but simultaneously *not* trust any other product elsewhere. Most attack vectors are *a priori* as likely at a client as at a cloud node.

In order to be practical, the cloud SQL DBS has to evaluate most queries in the cloud. This is especially the case for the select-project-join (SPJ) queries ubiquitous in OnLine Transaction Processing (OLTP). The only, but usually impractical alternative is to transfer data to the client. An SQL query often includes value expressions, which are frequent for OnLine Analytical Processing (OLAP) where aggregation is typical. Under the usual security paradigm, homomorphic encryption is necessary to avoid large transfers to clients. To be practical, homomorphic encyrption must be fast, so that response times for processing unencrypted and encrypted data are comparable. While homomorphic encryption has been a long-lasting research goal, up to now only semihomomorphic schemes such as the additively homomorphic scheme of Pailliers are fast enough for specific applications. See [2] for further details.

Our paradigm led to a new quasi-homomorphic encryption scheme called THE-scheme [2], which is faster and more general than previous proposals, but limited to a specific finite, numerical value domain. The THE-scheme encloses some sensivite metadata, the *client secret*, with each query involving an arithmetical operator other than addition. The cloud DBS uses the client secret and discards all the secret run-time values before the query execution terminates.

Below, we explore this paradigm further. A query might now include sensitive metadata such as cryptographic keys. Processing a SELECT query may decrypt a table into a run-time value. An UPDATE query may produce on-the-fly encryption to produce a ciphertext to be stored in the DB. The metadata and any plaintext produced is cleared before the query execution terminates. The DB behaves functionally as if it were to use full homomorphic encryption.

We call the cloud DBS designed as outlined *trusted*. We believe that trusted DBS are a promising avenue for research in the cloud. Below, we first propose a reference architecture. This architecture is software only, i.e. we forego the use of any specific hardware add-on, as we want to serve mass production. We also propose that any DB managed by a trusted DBS uses AES256 encryption. We define a deterministic and a probabilistic AES-based encryption scheme, the latter providing protection against frequency analysis whereas the former is faster for SPJ queries and hence more attractive for OLTP.

We give an overview for the rules of a query execution plan for an AES DB. We then analyze performance, where we focus on the processing time and the storage overhead compared to a DB that stores data in unencrypted form. We show that for modern multi-core processors such as the INTEL I5, processing and storage overhead increase only negligible. For the probabilistic version, this should still be true for aggregations. This makes our proposal eighty times faster than Paillier's scheme. In contrast to our deterministic encryption (but not to Paillier's), the storage doubles. See [3] for details.

We base our conclusions on recent AES benchmarks [5]. Some show the need for experiments specific a cloud SQL DBS. This task remains for future work. We fell that our proposal is of major practical importance, especially because important products such as Google Cloud SQL and MS Azure SQL offer already server-side AES256 encryption transparent to the user. Unfortunately, these products do not protect against cloud insiders as the key is stored persistently in the cloud. Azure also offers client-side encryption, which unfortunately requires transfering entire tables or even the entire DB between servers and the client [13]. There are also other limitations, for example, no equijoins on probabilistically encrypted columns [9].

The next section presents the AES DBS. We discuss the reference architecture of a trusted DBS and define the use of AES for a cloud DB. We also discuss the generation of a query execution plan. Section 3 analyzes the processing and storage overhead. We draw conclusions and discuss future work in Section 4. We stress that implementing a trusted DBS should be easy, for example using an existing cloud MySQL implementation or an SQL DBS with user defined functions.

II. The trusted AES DBS

A. Reference Architecture for a trusted cloud DBS

Figure 1 depicts our reference architecture. It applies to any cloud DBS designed according to our paradigm, regardless of the encryption method use. An AES-based trusted DB is only a specific case of the architecture. The DB administrator at a client site initiates the cloud DB through some kind of upload not discussed here. The DB itself is encrypted. Clients operate on the DB through SQL queries. While an SPJ query might manipulate directly relations encrypted deterministically (seebelow), most queries need to work on relations in plaintext and hence need to decrypt. An update also entails encrypting relations. The cloud DBS performs decryption and encryption on the fly, while reading run-time plaintexts from and writing them to the DB. All data exchanged are protected in transport through some of the usual protocols and schemes (SSL, RSA, Diffie-Hellman, etc.) The DBS instantiates run-time variables with the metadata sent along with the query. It deletes any sensitive run-time content, including the metadata and retrieved or calculated plaintext data at the latest when query processing ends.

The cloud processes the queries using some core DBS. This is some full-fledged "classic" plaintext DBS. Our trusted cloud DBS is the core DBS reinforced whenever needed through security oriented (software-only) re-engineering. The reengineered trusted DBS runs on typical cloud hardware without the need for dedicated hardware add-ons such as a trusted computing module (TCM) [14]. The protection offered can be thaught of as a vault, armor, shield, or firewall. Independent of the name, it protects the core DBS from exploits that can reveal run-time values.

We believe that trusted DBS are a promising goal for cloud DB research. One technical base for the trust mechanism is *moving data defense.* Such defenses are extensively discussed in [4]. They dynamically and secretely shuffle the storage representation and location of the program instructions and run-time variables. The concept goes back to decade-old proposals for secure VMs [1]. Alternatively, many users might be happy with a trusted DBS running on a cloud node without any security reengeneering relying on a gradual increase in security through software and hardware updates. For instance, Intel announced its Software Guard Extensions (SGX) in 2013 and markets them in late 2015. Processors with SGX guard specific RAM areas called *enclaves*. No software outside an enclave can read or write the enclave's content regardless of its current privilege level and CPU mode. This mechanism protects run-time values for a trusted DBS.

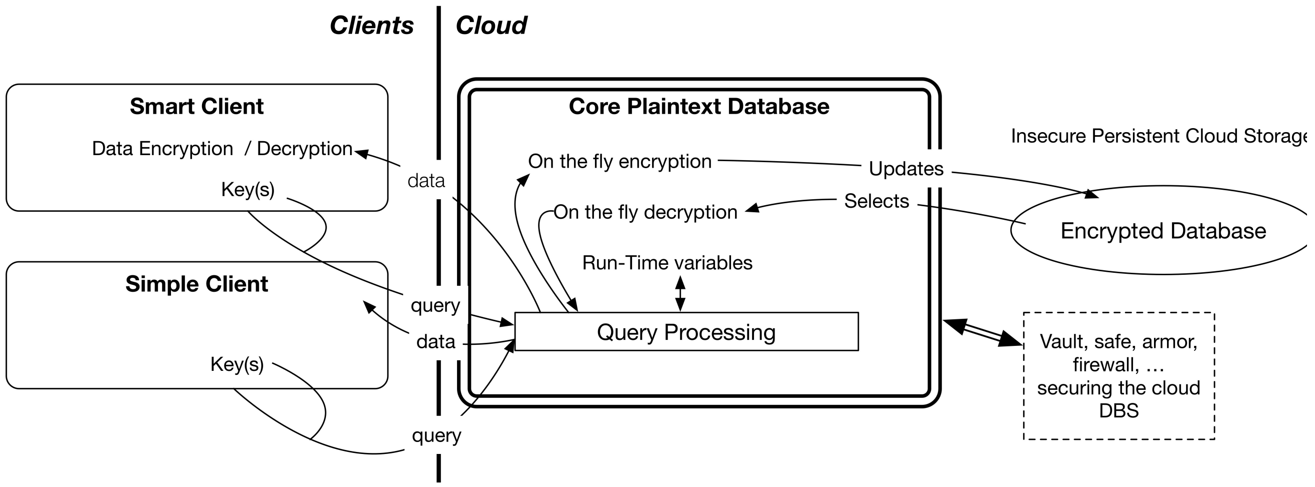

Fig. 1 Reference Architecture of Trusted Cloud DBS

While these techniques are promising, the security of a trusted cloud DBS will need to be adjusted to new technical developments and the comfort level and security needs of its users.

The trusted DBS may send 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 securely the decryption burden may clearly make many clients happier. In practice, even the popular browsers should thus be apt to serve as clients of a trusted DBS. All this appears an advantage of the new paradigm. The traditional one obviously excludes such clients.

As the result, two kinds of clients and queries appear, Fig. 1. A smart client of a trusted DBS is basically a client DBS able to locally encrypt/decrypt. A simple, (thin, dumb…) client does not have this capability. A smart client may send a ciphertext query. Such a query uses explicitly the encryption function on cipher-text constants or columns. It may carry ciphertext constants or bring ciphertext to the client for final decryption. A simple client emits plaintext queries only. Those cannot bring a ci-phertext. The cloud DBS interprets every constant of a plaintext query as a plaintext and every column name as refer-ring to a plaintext. A plaintext query must include the metadata with the key(s) (Conclusion of [3] shows alternative conventions). Every encryption/decryption is done on-the-fly at the cloud. A ciphertext query may avoid the metadata, as it might not need the decryption/encryption on the cloud. This is a potential security advantage. Coming however, we recall, at the price of burden at the client. A smart client may of course formulate any plaintext query with the key(s) as well. When there are several keys, e.g., one per column, a smart client may also send a ciphertext query with only some but not all the keys required by a plaintext one.

Example 1. (a) The following SPJ query is in plaintext. Metadata with the key `ABCD' follows with the semicolon terminating the SQL statement.

(1) *Select C.Name, A.Transactions from Customer C, Account A where C.Id = '123' and C.Id = A.Id ; 'ABCD'*

To evaluate the query, the cloud DBS may perform on-the-fly encryption of the selection constant '123'. Or, it may on-the-fly decrypt the visited Id ciphertexts. The choice depends on whether Id encryption is deterministic or probabilistic, see below. In every case, the DBS decrypts the final projection (C.Name, A.Transactions) by default.

(b) We denote the encryption function as AESE. The next query could be a ciphertext one of some smart client:

(2) Select Into Client.CacheDB.CipherTbl AESE (C.Name), AESE (A.Transactions) from Customer C, Account A where AESE (C.Id) = AESE ('123') and AESE (C.Id) = AESE (A.Id);

Unlike query (1), query (2) does not carry the key. As said, it is potentially a security advantage over query (1). In contrast, as we detail below, query (2) is valid only if the encryption of Id column is deterministic. Also, it is up to the client to encrypt '123'. If the clause was simply C.Id = '123', it would mean for the cloud necessarily the on-the-fly decryption of Id to test the match of '123'. The query would be rejected as not carrying the key. As is, the query brings 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 function, say AESD:

Select AESD (C.Name), AESD (A.Transactions) From CipherTbl;

B. Deterministic Encryption for an AES DB

From now on, we consider specifically the on-the-fly encryption/decryption through AES, as in Ex. 2. In other words, we restrict our focus to AES DBs only. As usual, we distinguish between OLTP and OLAP queries. It is well-known the former perform best with deterministic encryption. We recall that any given plaintext is then encrypted always to one ciphertext. Provided the client-side decryption and the individual encryption of each column plaintext value, i.e., no grouping of column plaintext values into a single ciphertext, (see [3] for that option), the cloud DBS can evaluate the selections and equi-joins of a typical SPJ query over the ciphertext. This could be the case of query (2) provided Name, and Id column individually and deterministically encrypted. Transactions column may or may not be deterministic. The client must be a smart one.

We recall that AES is a symmetric, deterministic block cipher using 16B blocks. Padding is necessary if values such as a double precision floating point number has a smaller size or to adjust to the length of strings. The most frequently used version of AES uses 256b keys.

C. Probabilistic Encryption for an AES DB

Deterministic encryption is vulnerable to a frequency analysis and therefore secure only for low-entropy domains. Its use is fine for random ID, social security numbers, tax payer ID, *etc*., but not for the transactions in the previous examples, ZIP codes, or salaries. In general, it should be avoided fro any column with a skewed domain value distribution [3].

We can use AES for probabilistic encryption by adding random elements to the values to be encoded. For example, if column values contain double precision floating point numbers, then we can add 8 random bytes to each value, so that each numerical value can be encrypted in 2**64 different ways. We can also define formats for small strings that contain random elements so that the same string can be encoded in many different ways. For very long strings, we can use cipher block chaining with an initialization vector to achieve the same end.

The drawback to the use of probabilistic codes is the impossibility to perform SPJ queries on encrypted data. Even an equi-join on probabilistically encrypted data fails because the same value is likely to be encrypted in two different ways in two records. Joins can only be executed by decrypting the values on which we do the join. The trusted DBS can do so only if it receives the key in the metadata.

A query to an AES DB referring to a column with a probabilistic ciphertext may be a plaintext one. E.g., - like query (3) but with the key in the metadata. The client does not need to know what the encryption of a column is really. A smart client may alternatively issue a ciphertext query, provided it includes the key as well. The DBS is obviously functionally able to evaluate any join or selection entirely at the cloud through their on-the-fly decryption. See again [3] for deeper discussion.

D. *Query Execution for an AES DB*

With respect to the execution plan, any trusted DBS, the AES DBS in particular, blends the on-the-fly decryption into the execution plan optimal for the DB as if it was the plaintext one. In other words, it generates some plan as if the DB contained the plaintext. Then, it decrypts on-the-fly when-ever needed every ciphertext selected during the query execution. In this way, the DBS is able to execute for any encrypted DB any query valid for the plaintext one. See Ex. 3 and Ex. 4 in [3] for samples of executions plans.

Notice already nevertheless that to execute query (4) at the cloud under the traditional paradigm, the encryption of Trans-actions should be fully homomorphic. There is no such scheme providing even remotely practical response time as yet.

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 sever-al recent benchmarks of AES: [5], [6], [10]. These 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 in-stance, - 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 proces-sors family, several CPUs have instructions for AES encryp-tion/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 I5, as the most used. Ac-cording 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 im-pressive 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 encryp-tion 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 decrypt-ed/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 im-pacts 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 from relatively a few only. The writing timing of these results should therefore very little impact of the read-only re-sults 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, allow-ing 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 rea-son is so-called key set-up time. Experiments show neverthe-less that the key set-up may cost for the Rijndael's algorithm as little as 15% slow-down [8]. An SQL query is typically ex-pected to do a bulk search. We thus neglect this (small anyway) specificity in what follows.

Finally, the AES algorithms above discussed appear pro-grammed in assembly language. Use of a higher-level compil-er, 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 result further. Using best optimized implementation for a cloud DBS seems natural. The subject requires nevertheless a specific study.

B. Storage Overhead

Our deterministic AES scheme may have theoretically no storage overhead. In practice, a negligible one may occur de-pending on specifics of a DB scheme. The probabilistic encryp-tion carries at best the 100% overhead, i.e., doubles the plaintext storage. This is nevertheless what probabilistic ho-momorphic schemes typically need at least as well, e.g., Pailli-er's scheme. So our scheme is not worse on storage require-ments.

C. Query Processing

The basic measure of this one is the overhead of on-the-fly decryption on the otherwise plaintext execution plan for the same SQL query. The overhead may depend on the execution plan. Globally, the decryption may deal even with GBytes of data per second. Also, the study of the read/write speeds for a ciphertext and a plaintext above has shown only 13% overhead. All this suggests that even for a RAM DB, the overhead of the on-the-fly decryption on the execution could be usually limited to a dozen of percent or so as well. It should become negligible if the DB is solid state or hard disc resident. Another measure can be the query execution speed of a query with respect to the same query executable also using a homomorphic encryption. The Select SUM(x)… query adding 100K plaintext values at the cloud server using Paillier, reputed the fastest traditional homomorphic scheme, needed 14ms per addition, [11], [12]. Our scheme could add up at best 0.2ms as above discussed. The overhead could thus be as low as 1.5 %. All other bench-marks we cited would cost only a few milliseconds at most. AES DB should thus be even in this limited case about eighty times faster. See again [3] for more.

IV. Conclusions

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 to offer in practice at present all the functional capabilities of a plaintext relational DBS. It is also the only to allow 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 should be also at least two orders of magnitude faster than for any known homomorphic encryption scheme. 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. Limited space does not let us to discuss most of those. See [3]. The main conclusion is that building an often likely to be sufficient AES DBS appears astonishingly easy. Nowadays, as we discussed, one may indeed reasonably trust the safety of the run-time variables of a major plaintext SQL DBS, e.g., in enclaves. Free MySQL appears then 1st choice. Its AES_ENCRYPT() and AES_DECRYPT() may implement our AESE (c, k) and AESD (c, k) functions. The RAND function should help with the probabilistic encryption. More generally, our goal seems also easy if an existing plaintext cloud SQL DBS supports user defined functions (UDFs), (unlike, e.g., Google Cloud version of MySQL at present). SQL Server seems then a good candidate as well, with the cloud-side AES256 encryption already offered in addition, [9]. In each case, a browser suffices to run plaintext queries as a simple client. It is likely the way to start practicing our proposal.

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, Comp. Arch. News 33 (1):34-41.

[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] I Jajodia, S. Litwin, W. Schwarz, Th. On-the fly AES Decryption/Encryption for Cloud SQL Databases. Lamsade Res. Rep. June 2015. http://www.lamsade.dauphine.fr/~litwin/Next%20step%205.pdf

[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.