Release-aware In-out Encryption Adjustment in MongoDB Query Processing

Maryam Almarwani, Boris Konev and Alexei Lisitsa
Department of Computer Science, University of Liverpool, Liverpool, U.K.

Keywords: Querying Over Encrypted Data, Document Database, CryptDB.

Abstract: Querying over encrypted data typically uses multi-layered (onion) encryption, which requires level adjustment when processing queries. Previous studies, such as on CryptDB, emphasize the importance of inward encryption adjustment, from outer layers to inner layers, releasing information necessary for query execution. Even though the idea of outward encryption adjustment, which is used to re-establish the outer layers after query processing, is very natural and appeared already in the early papers on CryptDB as a topic for future work, no prior studies have addressed it systematically. This paper extends previous work on intelligent, release-aware encryption adjustment for document-based databases querying with the outward adjustment policy. We define the resulting Release-Aware In-Out Encryption Adjustment principles and report on their empirical evaluation using both local and cloud deployment of MongoDB. The evaluation utilizing datasets of different sizes shows that the proposed method is efficient, scalable, and provides better data protection.

1 INTRODUCTION

To keep data protected, it is usually stored in an encrypted form, and it needs to be decrypted before processing or querying. Decryption, however, could make the data vulnerable to server-side attacks. To mitigate the issue, querying over encrypted data has been proposed, in which multi-layered (onion) encryption is commonly used. Multiple layers protect information and, at the same time, may release some partial information on the data depending on the types of encryption used at different layers. Consequently, a reasonable encryption adjustment policy is needed, which given a query, leads to decryption of some layers to reveal the information sufficient for the query execution. Such a scheme was first proposed for relational databases and SQL querying in a well-known work on CryptDB (Popa et al., 2011) and then was transferred further to other types of databases and query languages/APIs, see e.g. (Aburawi et al., 2018b) for graph databases, (Xu et al., 2017),(Almarwani. et al., 2019),(Almarwani et al., 2020a) for document databases, and (Shih and Chang, 2017),(Waage and Wiese, 2017),(Wiese et al., 2020) for wide-column database. As it has been already noticed in (Popa et al., 2011) a simple encryption adjustment policy (SEA) may lead to an unnecessary revelation of more information about the data than required for the query execution. To alleviate the issue to some extent, a new Join-aware encryption adjustment (JAEA) schema for relational databases was proposed in (Popa et al., 2011). Similarly, traversal-aware schema (TAEA) for graph databases and release-aware schema (RAEA) for document databases were proposed in (Aburawi et al., 2018a) and (Almarwani et al., 2020b), respectively. These schemata turned out to be more secure than variants of SEA by releasing less information when executing JOIN, traversal, and conjunctive/disjunctive queries, respectively.

The main focus of the above-mentioned adjustment technologies was in applying decryption to the layers from outer to inner, releasing more information about the data and making the data less secure. Far less attention was paid to the question of what to do after the query has been completed and data protected by the inner layers remain less secure. In the original paper on CryptDB (Popa et al., 2011), a very natural approach was outlined – to re-encrypt the data to the original levels of encryption and restore protection, but an elaboration of the details was left to future work. The development of encryption adjustment techniques has shown that the details are important here, as adopting different policies would likely affect both efficiency and security. To the best of our knowledge, these type of questions has not been addressed systematically by the previous stud-

Almarwani, M., Konev, B. and Lisitsa, A.
Release-aware In-out Encryption Adjustment in MongoDB Query Processing.
DOI: 10.5220/0010395007140722
In *Proceedings of the 7th International Conference on Information Systems Security and Privacy (ICISSP 2021)*, pages 714-722
ISBN: 978-989-758-491-6
Copyright © 2021 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

ies. In this paper, we address the questions of inward and outward adjustment for querying encrypted document databases and introduce Release-Aware In-Out (RAIO) Encryption Adjustment. This is an extension of the RAEA policy we proposed early for inward adjustment in (Almarwani et al., 2020b). We develop RAIO for NoSQL document-based MongoDB databases. NoSQL databases become very popular in recent years to handle large amounts of unstructured data, and MongoDB is the most popular NoSQL database according to the ranking in The contributions of this paper can be summarized as follows:

- We propose an outward adjustment procedure, an additional phase after processing queries to restore the outer encryption layers that were adjusted before processing. Both phases constitute a novel RAIO policy.
- We analyse the security and incurred overhead of our new policy compared to the original policy.
- We report on the empirical evaluation of the scalability of our policy in comparison with the original policy.
- We provide an assessment of the performance of our policy on MongoDB deployed locally and on the cloud from three cloud service providers.

This paper is organized as follows. Section II gives a brief overview of multi-layer encryption and the MongoDB API. A Release-aware In-Out Encryption Adjustment is introduced in Section III. Our experiments and results are reported in Section IV. Section V summarizes relevant work. Our conclusions and future work are drawn in the final section.

2 BACKGROUND

2.1 Multi-layered Encryption

Multi-layered, or onion encryption is a fundamental concept used in the implementation of querying over encrypted data since the influential paper on CryptDB (Popa et al., 2011). The idea is simple and powerful. Each data element is encrypted by several layers of encryption of different types. Full data protection can be achieved by using a *random* layer of encryption (RND), which does not reveal any information about encrypted data, not even equality. *Deterministic* layer (DET) allows testing the equality between encrypted values. More advanced functionality can be provided by *partially-* or *fully- homomorphic* encryption (PHE, or FHE), allowing to perform computations over encrypted data. Furthermore, *order-preserving* encryption (OPE) and *searchable* encryption allow cryptographic numerical values to be compared and the search carried out. Depending on what queries are required, the data elements can be protected by different onions combining the layers of various types of encryption. Executing a query over such protected data may require encryption adjustment. For example, if a query requires an equality check (e.g., salary = 30.000) and the outer layer of encrypted salary values is RND and a layer immediately under it is DET, then the outer layer has to be removed by decryption before the query may proceed while still keeping data protected at DET level. The query pre-processing, the encryption adjustment, and post-processing of encrypted results in CryptDB-like schemes are performed by a proxy component placed between the server and a client. Figure 1 outlines the corresponding workflow. The proxy proceeds as follows: (i) it receives a query from an application/client and adjusts data on the database (ii) rewrites the query by replacing its explicit values/constants with their encrypted versions and sends the resulting query to the database; (iii) decrypts received results and sends them to the user. The approach we propose in this paper is generic and may work together with different types of encryption. However, for the validation and empirical evaluation, we have implemented a research prototype using three types of encryption methods for built-in layers, including AC, DET, and OPE. Access Control (AC)(Almarwani et al., 2020a) layer presents the values encrypted using *Cipher-Policy Attributes-Based Encryption* (CP-ABE) (Bethencourt et al., 2007). *AC* layer provides protection similar to *RND* (equal plaintext values are very likely encrypted to different ciphertexts), and additionally, it enables the implementation of access policies that determine the users authorized to access the data. *AC*, like *RND*, enables no computational operations concerning the encrypted value, while *DET* and OPE layers enable equality and range computation, respectively. Each encryption type releases some information about encrypted values, and the information inclusion induces a *partial* order on the encryption types. For the encryption types we use in this paper, this order is actually *linear*: $AC \preceq DET \preceq OPE \preceq PLAIN$, where *PLAIN* denotes "no encryption".

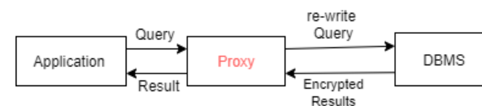


Figure 1: Workflow of Querying over Multi-layers Encryption data.

2.2 MongoDB API

In this paper, we develop a new encryption adjustment policy for querying encrypted document databases. We present and evaluate the concept using MongoDB. MongoDB data consist of collections, and each collection includes a number of documents. Each document is formed by fields and values. Unlike many other types of databases, MongoDB doesn't support a particular query language. Instead, an Application Program Interface (API) allows to manage MongoDB documents programmatically. MongoDB API high-level syntax is given below (Tutorial, 2017): To read

```
<Query>                  ::= dbName.CollectionName
                             .<Query-Type>
<Query-Type>             ::= <Insert> | <Find>
                           | <Update> | <Delete>
<Insert>                 ::= insert(<Insert-Criteria>)
<Find>                   ::= find(<Find-Criteria>)
                                  [.<Display-result>]
<Update>                 ::= update(<Update-Criteria>
                                   ,<Change-Criteria>)
<Delete>                 ::= delete(<Delete-Criteria>)
<Find-Criteria>          ::= "" |{<Expression>}
                          |{<Existence-Check>}
<Display-result>         ::= count()|pretty()|sort()
<Expression>             ::= <Expression1>|<Expression2>
<Expression1>            ::=  <expression1>
|<Logical-operators>:

                              {<expression1>
                              [,<expression>1]+}

                            |<Logical-operators>:

                              {<Logical-operators>

                              <expression1>

                   [,<expression1>+]}
<Expression2>             ::=<expression2>

                          |{<Logical-operators>:

                           [<expression2>

                           [,<expression2>]+]}
<expression2>             ::={$expr:{<Comparison-operators>:
                             [<Argument>[,<Argument>]+]}}
<Argument>               ::=<Field>
<Logical-operators>      ::= $and | $or | $nor | $not
<expression1>            ::= <Field> :
                             <Comparison-operators> <Value>
<Field>                  ::= ((A...Z)| (a...z))+
<Comparison-operators>   ::=$eq| $gt | $gte | $lt | $lte
<Value>                  ::= ((A...Z)| (a...z)|(0...9) )+
{<Existence-Check>}      ::= {$exist:[true|false]}
<Insert-Criteria>        ::={<Insert-Expression>
                            [,<Insert-Expression>]+}
<Insert-Expression>      ::=<Field> : <Value>
<Update-Criteria>        ::=<Find-Criteria>
<Change-Criteria>        ::=<Insert-Criteria>
<Delete-Criteria>        ::=<Find-Criteria>
```

data from MongoDB, the *find()* function is used. If the *Find-Criteria* is empty, all documents in the collection will be retrieved and returned. In contrast, if the *Find-Criteria* is defined, documents that only satisfy certain criteria will be retrieved and returned. To modify data the functions *insert()*, *update()*, *delete()* are used.

Insert() function is to add a document to the collection; therefore, the *Insert-Criteria* no need operators or conditions to do it.

Update() function is to update fields values in *Change-Criteria* Parameter for fetched documents by *Update-Criteria*.

Delete() function is to delete documents that match the *Delete-Criteria*. Further details of the syntax and semantics of the API can be found in (Tutorial, 2017).

3 ENCRYPTION ADJUSTMENT POLICIES

To perform queries and updates on encrypted MongoDB data, one should evaluate *Conditions* over such data. According to MongoDB API BNF, *Conditions* are boolean combinations of atomic *Expressions*, which are build using comparison operators. For a comparison operator op we denote by SL_{op} the minimal wrt to $\preceq$ encryption level (*S*upport *L*evel) of the arguments of op sufficient for evaluation of op on these arguments. We have $SL_{=} = DET$, while $SL_{<} = SL_{\leq} = SL_{\geq} = SL_{>} = OPE$. For a MongoDB API query Q we denote by $E(Q)$ the set of all atomic expressions occurring in Q. For an atomic expression (`<Expression 1>`) e we denote by $op(e)$ the operator occurring in e and by $field(e)$ the field name occurring in e.

For a query Q and a field name f occurring in Q we define $S_Q(f) = max_{\{e \in E(Q) | field(e)=f\}} SL_{op(e)}$.

Notice that the same field name f may occur in different expressions, which may require a different level of adjustment. Taking maximums (wrt to $\preceq$) adjustment levels for all f as assumed in the definition of $S_Q(f)$ makes it possible to evaluate all expressions in $E(Q)$ and execute Q over encrypted data store. For a MongoDB database instance I and a field f we denote by $\#f_I$ the number of the documents in I containing the field f.

3.1 Simple Encryption Adjustment

Simple Encryption Adjustment policy (SEA) follows very simple principles. According to SEA, *before* an execution of a query Q the encryption levels of *all* values of *all* fields f occurring in Q have to be adjusted to the encryption levels $S_Q(f)$. The query Q itself has to be rewritten into its "encrypted" version Q^*: all plain values of all fields f in Q have to be replaced by encrypted $S_Q(f)$ values.

Such defined policy operates *inwards* - starting with the maximum security level (or minimum wrt to information release order $\preceq$) it adjusts the encryption to lower security (higher wrt $\preceq$) levels for some data elements fields.

SEA policy was first proposed for relational databases in (Popa et al., 2011) and its variant for document-based DBs was considered in (Almarwani et al., 2020a). SEA can be extended naturally to the

outward mode too: after SEA applied inwards, and a query is executed, the encryption of all adjusted values has to be restored to the outer layer of maximal security.

While SEA policy is easy to define and implement, it may incur large overhead for large databases (too many data values have to be decrypted/re-encrypted). It may reveal more information about data elements than necessary for the query execution (Popa et al., 2011). To alleviate such effects, (Almarwani et al., 2020b) proposed Release-Aware Encryption Adjustment (RAEA) policy, originally in inward mode only.

3.2 Release-aware In-out Encryption Adjustment

The Inward RAEA strategy releases less server-side information than SEA by reducing the number of encryption adjustments to the inner layer(s). This limits the number of decryptions by 1) performing the adjustment not before but alongside the query execution and 2) taking into account the number of occurrences ('popularity') of the various fields. We now describe an Outward Adjustment extension of RAEA, which we call the Release-Aware In-Out (RAIO) Encryption Adjustment. The policy is defined for both Conjunctive Conditions and more general Disjunctive Normal Form (DNF) conditions for the MongoDB API calls. These Condition classes in the MongoDB API can be defined as follows:

```
<Class-Condition> ::= <Conjunctive> | <DNF>
<Conjunctive>     ::= $and <Expression>
                       [,<Expression>+]
<DNF>             ::= $or<Conjunctive>
                       [,<Conjunctive >+]
```

RAIO encryption adjustment consists of two parts. The first part, Inward adjustment, expands The Inward RAEA policy from (Almarwani et al., 2020b) to the wider class of conditions. The second part, Outward adjustment, is a newly proposed policy to deal with the re-establishing protection of the data whose encryption layers have been adjusted for querying.

(A) Release-Aware (Inward) Encryption Adjustment(RAEA)

Find Queries. First, we consider the Conjunctive Conditions case. A *Find query Q* in MongoDB API has a form `db.collection.find(C(Q))`, where for conjunctive queries $C(Q) = e$ or $C(Q) = \texttt{\$And}\{e_1,\ldots,e_n\}$ and e and e_i are atomic expressions. For $C(Q) = \texttt{\$And}\{e_1,\ldots,e_n\}$ and $1 \le k \le n$ we denote $\texttt{\$And}\{e_1,\ldots,e_k\}$ by $C_k(Q)$. For $C(Q)$ and $C_k(Q)$ we denote by $C^*(Q)$ and $C^*_k(Q)$ their encrypted variants obtained by adjustments of constant values in all e_i to the encryption levels $S_Q(field(e_i))$. Corresponding queries will be denoted by Q^* and Q^*_k, respectively. For a find query Q and a database instance I we denote by $D(Q,I)$ the set of documents returned by execution of Q on I.

Given a conjunctive query Q and a MongoDB database instance I, RAEA adjustment and query execution proceed as follows:

If $C(Q) = e$, then SAE policy is applied: *all* values of $field(e)$ in I are adjusted to the encryption level $S_Q(field(e))$. The query Q is encrypted to Q^* by adjusting the constant value in e to the same level $S_Q(field(e))$. Then Q^* is executed.

If $C(Q) = \texttt{\$And}\{e_1,\ldots,e_n\}$, $n \ge 2$ then popularity in I of all fields f occurring in $C(Q)$ is calculated by calling $\#f_I$ = `db.collection.find($f$:{\$exist:true}) .count();` The original query condition $C(Q)$ is re-written by sorting atomic expressions according to the *popularity order*, that is to $C(Q_{\le}) = \texttt{\$And}\{e_1,\ldots,e_n\}$, where $\#field(e_i)_I \le \#field(e_{i+1})_I$, $i = 1,\ldots,n-1$. Then the following iterative steps are executed. Let $I_0 = I$.

- All values of $field(e_1)$ in the whole I_0 are adjusted to the encryption level $S_Q(field(e_1))$ resulting in a database state I_1;
- The values of $field(e_2)$ occurring in the documents $D(Q^*_{\le_1},I_0)$ are adjusted to $S_Q(field(e_2))$ resulting in I_2;
- ...
- The values of $field(e_i)$ occurring in the documents from $D(Q^*_{\le_{i-1}},I_{i-2})$ are adjusted to $S_Q(field(e_i))$ resulting in I_i;
- ...

These steps are completed either

- by the execution of the last possible $Q^*_{\le_n}$, that is $Q^*_{\le}$ on I_{n-1} with the result being the required (encrypted) result of the execution of the whole query Q on encrypted instance I; or
- by a query $Q^*_{\le_k}$ with $k < i$ returning empty set of documents.

In either case, n, respectively k are recorded as *stop points* of the execution. This information is needed for the procedure of outward adjustment which should follow afterwards.

As the fields can be repeated in the query, we assume that before an adjustment of any field to be done, it is checked whether it is already been done, and if so, it is skipped.

The presented policy may reduce the number of adjustments/re-encryptions compared with SEA, since adjustments of the fields' values are starting

with the least popular fields and are performed on gradually narrowing sets of documents. Consider an example. Let Q be the following query:

```
db.collection.find($AND:{Name:$eq Alice,
Salary:$gt 9000, Salary:$lt 40000,Age:$lt
50,Age:$gt 25,Department:$eq Computer})Q1
```

Assuming $\#Salary_{I_0} \leq \#Age_{I_0} \leq \#Department_{I_0}$ in an instance I_0 we have $C(Q_{\leq}) =$

```
$AND:{Salary:\$gt 9000, Salary:\$lt 40000,
Age:\$lt 50,Age:\$gt 25,Name:\$eq Alice,
Department:\$eq Computer})
```

- As $field(e_1) = Salary$ *all* values of *Salary* in the whole I_0 are adjusted to the encryption level OPE resulting in I_1.
- As $field(e_2) = Salary$ and the encryption level of *Salary* values has already been adjusted we skip its further adjustments.
- The values of *Age = field(e3)* occurring in the documents $D(Q^*_{\leq_2}, I_1)$, that is those matching $e1^* \wedge e2^*$, are adjusted to the encryption level OPE resulting in I_3;
- Skip further adjustments of *Age = field((e4)* as it has been already adjusted.
- The values of *Name = field(e5)* occurring in the documents $D(Q^*_{\leq_4}, I_3)$, that is those matching $e1^* \wedge e2^* \wedge e3^* \wedge e4^*$, are adjusted to the encryption level DET resulting in I_5;
- The values of *Department = field(e6)* occurring in the documents $D(Q^*_{\leq_5}, I_4)$, that is those matching $e1^* \wedge e2^* \wedge e3^* \wedge e4^* \wedge e5^*$, are adjusted to the encryption level DET resulting in I_6;
- Finally, assuming all previous partial queries did not return empty set of the documents, we execute $Q_{\leq} = Q^*_{\leq_6}$ on I_6 and $D(Q^*_{\leq_6}, I_6)$ is the (encrypted) result of the execution of the original Q on the encrypted instance I_0.

Each time a new f is adjusted, the number of *selected* documents is lower than that in the previous step, despite the new f being no less/more popular in I. This is because the increasing number of constraints e_i in $C^*_k(Q)$ with growing k, leads to the reduction of the numbers of the documents in $D(Q^*_{\leq_{k-1}}, I_{k-2})$ that are more likely to satisfy Q.

Disjunctive Normal Form (DNF) Case. For a DNF query Q we have $C(Q)$ = `$Or`$\{c_i, \ldots c_k\}$, where c_i = `$And`$\{e_1, \ldots, e_{n_i}\}$. The processing of such a Q is done by a separate processing of conjunctive queries with conditions c_i as explained above and taking the union of the sets of the documents returned by these conjunctive queries as the result. To ensure correctness and efficiency, the information on already adjusted fields is passed to the the processing of further conjunctive sub-queries. In this case there are many opportunities for further optimizations and this is a subject of our ongoing work.

Write Query Processing. For the case of Update and Delete queries, the proxy processes $C(Q)$ using the same steps as for a Read (Find) queries except there is no need to receive and decrypt a result. For the case of Insert queries, the proxy encrypts all inserted values by layers to the maximum security level.

(B) Outward Encryption Adjustment(OEA) Restoring the data values to the maximum security level after executing queries reduces the amount of information exposed to the database after query processing. The main difficulty here is to find out which values have been adjusted to the lower levels. We propose here the Outward Encryption Adjustment (OEA) policy, which works on pair with inward RAEA and allows to determine which values need to be restored to the maximum security level without disclosing more than necessary information to the database.

Find Queries. First, we consider the Conjunctive Conditions case. We assume that a query Q has already been processed using above RAEA policy. If $C(Q^*)$=e^* then *all* values of $field(e)$ in I are restored to the maximum security level.

If $Q^*_{\leq_s}$ = `$And`$\{e^*_1, \ldots, e^*_s\}$, where s is the stop point of RAEA adjustment and $s \geq 2$, then $Q^*_{\leq_s}$ is re-written in reverse to sort atomic expressions descending to the *popularity order*, that is to $Q^*_{\geq_s}$ =`$And`$\{e^*_s, \ldots, e^*_1\}$, where $\#field(e^*_i\})_I \geq$ `"#"`$field(e^*_{i-1}\})_I$, $i = s - 1, \ldots, 1$.

Then the following iterative steps are executed. Let $I_0 = I$.

- The values of $field(e_s)$ occurring in the documents from $D(Q^*_{\geq_{s-1}}, I_0)$ are restored to the maximum security level resulting in I_{s-1};
- $\cdots$
- The values of $field(e_i)$ occurring in the documents from $D(Q^*_{\geq_{i-1}}, I_{s-i})$ are restored to the maximum security level resulting in I_{i-1};
- $\cdots$
- All values of $field(e_1)$ in the whole I_0 are restored to the maximum security level.

We illustrate the proposed outward adjustment procedure by the following example.

Let $C(Q_{\leq}) =$

```
$AND:{Salary:\$gt 9000, Salary:\$lt 40000,
Age:\$lt 50,Age:\$gt 25,Name:\$eq Alice,
Department:\$eq Computer})
```

re-write $C(Q_{\geq}) =$

```
$AND:{Department:$eq Computer,
Name:$eq Alice,Age:$gt 25,Age:$lt 50,
Salary:$lt 40000,Salary:\$gt 9000})
```

where $\#Department(e6) \geq \#Name(e5) \geq \#Age(e4) \geq \#Age(e3) \geq \#Salary(e2) \geq \#Salary(e1)$.

- The values of *Department = field(e6)* occurring in the documents $D(Q^*_{\leq 5}, I_0)$, which are matching $e5^* \wedge e4^* \wedge e3^* \wedge e2^* \wedge e1^*$, are restored to the maximum security level resulting in I_6;
- The values of *Name = field(e5)* occurring in the documents $D(Q^*_{\leq 4}, I_3)$, which are matching $e4^* \wedge e3^* \wedge e2^* \wedge e1^*$, are restored to the maximum security level resulting in I_5;
- Skip *Age(e4)* that is repeated field and not been restored yet.
- The values of *Age = field(e3)* occurring in the documents $D(Q^*_{\leq 2}, I_1)$, which are matching $e2^* \wedge e1^*$, are restored to the maximum security level resulting in I_3;
- Skip *Salary = field(e2)* that is repeated field and not been restored yet.
- All values of *Salary = field(e1)* in the whole I_0 are restored to the maximum security level.

The case of outward adjustment for Disjunctive Normal Form (DNF) conditions is dealt with similarly to the case of inward adjustment: each of the conjunctive components is processed separately.

Write Queries. Here, the *Insert query* does not need any action in OEA while the *Delete query* follows the same steps as for a Find query. Whereas, for *Update query*, if at least one of the field in the *change-criteria* is present in the *Update-criteria*, updated documents must be processed to restore values to the maximum security level because these documents do not meet the *update-criteria*. Then, the steps for Find query using *update-criteria* are followed for non-updated documents.

(C) Case Study of SEA VS.RAIO

This section presents a case study and compares the SEA and RAIO applied to the same small database instance. Figure 3 presents a sample of the document database with nine documents, each containing up to four fields with a total of 29 values, as shown in Figure 2 in the left part. The data is encrypted with one onion by using three algorithms(CP-ABE(AC), AES(DET), Order-preserving Encrypted(OPE)), as shown in Figure 2 in the right part. Q3 is an example of the query which is executed over encrypted data using inward and outward encryption adjustments following two policies, SEA, and RAIO. Therefore, it is noticed that *all* values (i.e. *29*) in $E(Q)$ are adjusted in IEA and are restored in OEA for SEA, as shown

	Raw Data				Encrypted Data				
Feild	Name	Salary	Department	ID	Name	Salary	Department	ID	IV
Values	Coby	8297	Human Resources	2B0CDFC6	Tfgfnkjqc	tES9AK	YCKKuFYeFlk	bbePArciX	wabtnb3M
	Garrett	3025	Media Relations	""	FGZXQTPP	TGRsED	pfhpfwkgjwW	""	PxcrchUN7
	Arden	""	Public Relations	DB555805	SBFJlwuy	""	JGipacDlwhdi	ZD5tGlBcM	Qbq5LRmY
	Georgia	9946	Public Relations	""	raBTsdUS	AkVN8r	OpDxJmdCRm	""	ACfh6BRz4
	Nero	7205	Quality Assurance	02CADFD4	Dszce7nb	TqutoJC	vzYzoakQcBSI	C3fDPYZcD	cduPeU6gQ
	Kato	8235	""	""	K2botfwgz	LokbHp	""	""	JJwwBJRCtF
	Keely	4624	Public Relations	FB521AF5	xepOTFw7	TDCPrd	DAAoZotjLTHR	oyMYU7fkg	FBlwOBbj5i
	Roary	2138	""	7AA0B465	VozkbtWJs	wYdJmd	""	rjSQ7eNczH	r4sJe9NiNq
	Anthony	9575	Quality Assurance	""	XEmXRRhp	RAitOzB	dbrefJYdppjRA	""	hhr4Ks899A
No.Values	9	8	7	5	9	8	7	5	

Figure 2: Case study sample data.

Adjustment type	Release-Aware Encryption Adjustment				Simple Encryption Adjustment				
Feild	*Name*	*Salary*	*Department*	*ID*	*Name*	*Salary*	*Department*	*ID*	IV
Values	Tfgfnkjqc	**8wqlk7**	YCKKuFYeFlk	**JCFDERT**	**vdefghtyi**	**8wqlk7**	**ohdnbsjkaatr**	**JCFDERT**	wabtnb3M
	FGZXQTP	**3vbmg5**	pfhpfwkgjwW	""	**lkjdnvywm**	**3vbmg5**	**Ryjmfiolmnbqe**	""	PxcrchUN7
	SBFJlwuy	""	JGipacDlwhdi	**AntTUJND**	**Sewadhue**	""	**zsequiopSffjji**	**AntTUJND**	Qbq5LRmY
	ESAKVNy	**9ewlf6**	OpDxJmdCRm	""	**ESAKVNy**	**9ewlf6**	**Nhgfrtyuiown**	""	ACfh6BRz4
	Dszce7nb	**7dfhgj5**	**kPEYzxVNFYTgu**	**OPESDFb**	**Nbfwqohy**	**7dfhgj5**	**kPEYzxVNFYTgu**	**OPESDFb**	cduPeU6gQ
	K2botfwg	**8ghjhj5**	""	""	**VSEmbnjd**	**8ghjhj5**	""	""	JJwwBJRCtF
	xepOTFw	**4jkncg4**	DAAoZotjLTHR	**QWCBNMR**	**weutjvmb**	**4jkncg4**	**Quyythkjlnbhg**	**QWCBNMR**	FBlwOBbj5i
	VozkbtWJ	**2uygjh8**	""	**XDOGRVU**	**Qtreingvfl**	**2uygjh8**	""	**XDOGRVU**	r4sJe9NiNq
	FBlWobgj	**9hghgh6**	dbrefJYdppjRA	""	**FBlWobgjW**	**9hghgh6**	**Luyytyrbnbhfg**	""	hhr4Ks899A
No.feild values Adjustment	2	8	1	5	9	8	7	5	-

Figure 3: Inward Encryption Adjustment.

Adjustment type	Release-Aware Encryption Adjustment				Simple Encryption Adjustment				
Feild	*Name*	*Salary*	*Department*	*ID*	*Name*	*Salary*	*Department*	*ID*	IV
Values	Tfgfnkjqc	**tES9AK**	YCKKuFYeFlk	**bbePArciX**	**Tfgfnkjqc**	**tES9AK**	**YCKKuFYeFlk**	**bbePArciX**	wabtnb3M
	FGZXQTP	**TGRsED**	pfhpfwkgjwW	""	**FGZXQTPP**	**TGRsED**	**pfhpfwkgjwW**	""	PxcrchUN7
	SBFJlwuy	""	JGipacDlwhdi	**ZD5tGlBcM**	**SBFJlwuy**	""	**JGipacDlwhdi**	**ZD5tGlBcM**	Qbq5LRmY
	raBTsdUS	**AkVN8r**	OpDxJmdCRm	""	**raBTsdUS**	**AkVN8r**	**OpDxJmdCRm**	""	ACfh6BRz4
	Dszce7nb	**TqutoJC**	**vzYzoakQcBSI**	**C3fDPYZcD**	**Dszce7nb**	**TqutoJC**	**vzYzoakQcBSI**	**C3fDPYZcD**	cduPeU6gQ
	K2botfwg	**LokbHp**	""	""	**K2botfwgz**	**LokbHp**	""	""	JJwwBJRCtF
	xepOTFw	**TDCPrd**	DAAoZotjLTHR	**oyMYU7fkg**	**xepOTFw7**	**TDCPrd**	**DAAoZotjLTHR**	**oyMYU7fkg**	FBlwOBbj5i
	VozkbtWJ	**wYdJmd**	""	**rjSQ7eNczH**	**VozkbtWJs**	**wYdJmd**	""	**rjSQ7eNczH**	r4sJe9NiNq
	XEmXRRh	**RAitOzB**	dbrefJYdppjRA	""	**XEmXRRhp**	**RAitOzB**	**dbrefJYdppjRA**	""	hhr4Ks899A
No.feild values Adjustment	2	8	1	5	9	8	7	5	-

Figure 4: Outward Encryption Adjustment.

in bold red in Figure 3, 4 in the right part. In contrast, *selected* values (i.e. *16*) are adjusted in IEA and are restored in OEA for the RAIO case's, as shown in bold red in Figures 3, 4 in the left part. Therefore for this case RAIO provides better than SEA protection of the data, as fewer values are exposed at the lower protection levels.

```
$db.collection.find({$or:$and:{Name:$eq
"Anthony",Salary:$gt 9000}], $and:
{Department:$eq"Quality Assurance",
$ID:$eq:"02CADED4"}]}); Q3
```

4 EVALUATION

The evaluation of RAIO policy was carried out using a workflow from Figure 1 extended by Data Owner entity. The scheme contains four entities: Data Owner, User, Proxy, and Database. The data owner encrypts and uploads data to the database, while the proxy is responsible for handling queries over encrypted data between the database and the user.

Experiments Setup. Three layers of data encryption were used, CP-ABE(Bethencourt et al., 2007), AES(Halevi and Rogaway, 2003), and OPE(Boldyreva et al., 2011), with code derived from, respectively, (Wang,),(Kamble,), and (Savvides,). This implementation was based on Java 8 and NetBeans 8.2 . The experiments were conducted using a desktop PC running Windows 10 with an Intel Core

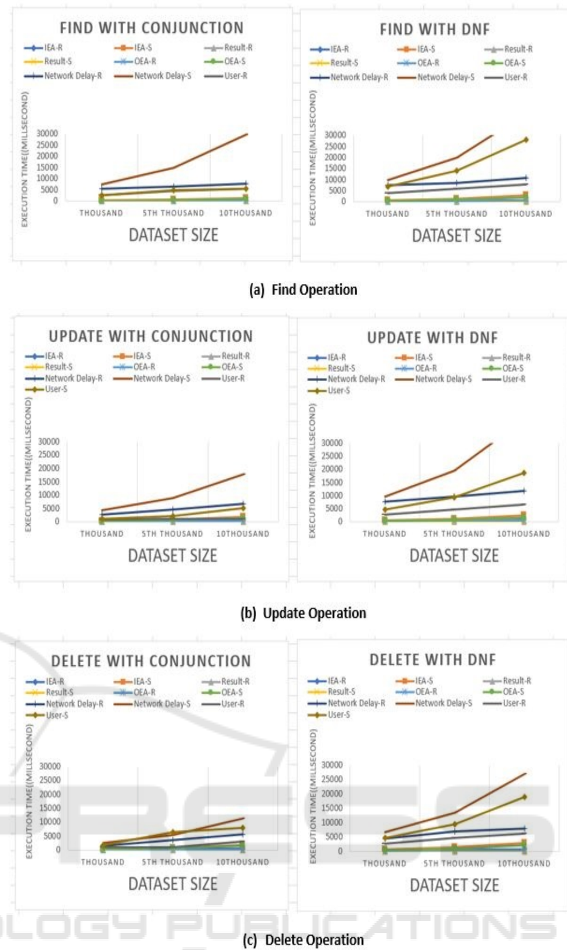
Table 1: Test Queries.

Q	*Criteria*	*Fields Query*	*Fields Change*
Find	Conj.	First-name,Salary	-
Find	DNF	First-name,Salary,Department,City	-
Update	Conj.	First-name,Salary	Salary,City
Update	DNF	First-name,Salary, Department,City	Salary,City,Department
Delete	Conj.	First-name,Salary	-
Delete	DNF	First-name,Salary,Department,City	-

1.8 GHz processor and 8.00 GB of RAM. A document can include First-Name, Last-Name, Credit-Card-Number, Salary, Department, Country, and City fields. Document data was generated using the API Mocking tool. For all experiments, user, data owner, and proxy were run in the local environment. In contrast, MongoDB was run either as Local, on the MongoDB community, or in the Cloud, using three different cloud providers, Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP).

Experiments Design. The main objective of the experiments conducted is to test the RAIO's efficiency as compared to SAE. We have carried out two types of experiments using different storage providers and different sizes of datasets. Conjunctive and Disjunctive Normal Form (DNF) queries were chosen for experiments with Find, Update and Delete operations. The experimental query's specifics are shown in the table 1. Each query ran 30 times, and the user, proxy, and database entities were evaluated for their average execution times. User time is calculated between issuing the query and the retrieval of results or executing data changes to update and delete queries. The proxy time, on the other hand, is the sum of the IEA and OEA time, plus the database communication time, which is the time needed for data to be retrieved, changed, or removed, and the result's decryption time. The first experiment measured the scalability of the proposed approach to query processing, and it involved increasing the number of database documents (1,000, 5,000, and 10,000 documents). In the second experiment the performance of RAIO was also tested for MongoDB deployed in the cloud. The second experiment calculated the proposed approach's execution time for three cloud providers supporting MongoDB and compared it to the local MongoDB's execution time.

Results. In Table 2, the mean execution time for various stages of processing for both RAIO and SEA (Experiment 1) is shown. According to these results RAIO policy consistently outperforms SEA. While IEA and OEA times appear to grow linearly in the size of the databases for both policies, in all cases the time taken for these stages by RAIO is considerably less than by SAE. The difference is even more dramatic for Network-Delay stage. For simple policy the growth of Network-Delay time is superlinear, while for RAIO it is still close to be linear, and for



(a) Find Operation

(b) Update Operation

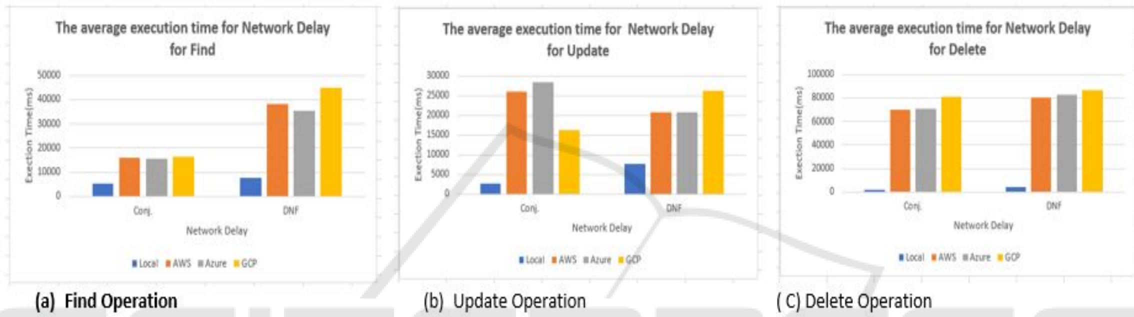
(c) Delete Operation

Figure 5: Experiment 1 Result.

databases of size 10000 Network-Delay time is ˜ 2-3 times less for RAIO. There is no significant difference in Result times, as this is for decryption of the results of the query, and the number of result values is the same, no matter under which policy a query is executed. We notice that Network-Delays(ND) have a considerable effect on total proxy time and user time due to the need for round-trip connections for adjustment between the DB and proxy. Table 3 displays the average times obtained for Experiment 2. Figure 6 displays the details of the average execution time for local MongoDB vs. cloud providers. Experiment 2 revealed, as predicted that the ND time has a significant effect on execution time due to the round-trip connection. Due to Internet speeds or the performance of the service services, an increase in time on the cloud could also have been incurred.

Table 2: Comparison of Simple and Release-Aware In-Out Encryption Adjustment on Local MongoDB Community.

	Q	Criteria	RAIO (ms)					SEA (ms)				
			User	Proxy				User	Proxy			
				IEA	Result	OEA	ND		IEA	Result	OEA	ND
Thousand	Find	Conj.	2784	165	2.13	56	5398.13	4145	321	2.66	200	7398.13
		DNF	4086.40	482.40	2.74	209.67	7471.60	6987.50	694.20	2.12	474.01	9914.90
	Update	Conj.	825.87	201.27	0.67	177.40	266.6	1236.74	436.75	1.74	321.90	4365.10
		DNF	2901.33	403.93	3.67	227	7671.6	4648.95	631.88	3.05	432.10	9748.70
	Delete	Conj.	961.73	262.13	0	149.80	1612.67	1745.21	410.75	0	394.30	2745.97
		DNF	2734.20	842.67	0	223.20	4455.33	4745.10	696.75	0	531.30	6745.12
Five-Thousand	Find	Conj.	4784	277	2.45	156	6398.13	8364	662	2.01	419	14826.27
		DNF	6074.70	671.20	2.15	409.67	8471.60	14031	1411.40	2.33	982.02	19874.80
	Update	Conj.	1025.15	422.75	1.14	274.70	4745.80	2529.48	907.49	1.45	737.80	8827.20
		DNF	4784.12	611.75	3.79	474	9745.10	9320.90	1275.75	3.80	864.20	19573.40
	Delete	Conj.	1171.79	474.24	0	214.10	3745.12	3502.42	877.49	0	809.60	5588.94
		DNF	4741.80	674.75	0	497.80	6954.78	9557.20	1464.49	0	1094.60	13531.24
Ten-Thousand	Find	Conj.	5412	474	2.95	278	7745.12	16760	1398	2.14	892	29695.53
		DNF	7748.90	874.90	2.89	614.78	10963.20	28076	2920.8	2.24	1976.05	39794.60
	Update	Conj.	1230.75	674.75	1.80	441.20	6741.90	5068.96	1837.98	1.97	1529.60	177752.4
		DNF	6741.32	811.44	3.33	601	11964.20	18706.80	2596.50	3.10	1815.40	39180.80
	Delete	Conj.	3174.91	647.03	0	432.50	5745.12	7038.84	1766.98	0	1503.60	11275.88
		DNF	6397.7	874.91	0	641.90	8012.98	19126.40	2995.98	0	2200.20	27085.48



(a) Find Operation (b) Update Operation (C) Delete Operation

Figure 6: Experiment 2 Result.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

Table 3: Comparison of Release-Aware In-Out Encryption Adjustment on Local MongoDB Community vs. Clouds Providers.

DB	Q	Criteria	RAIO (ms)				
			User	Proxy			
				IEA	Result	OEA	ND
Local	Find	Conj.	2784	165	2.133	56	5398.13
		DNF	4086.4	482.4	2.73	209.67	7471.60
	Update	Conj.	825.87	201.27	0.67	177.40	2662.60
		DNF	2901.33	403.93	3.67	227	7671.60
	Delete	Conj.	961.73	262.13	0	149.80	1612.67
		DNF	2737.20	482.67	0	223.20	4455.33
Azure	Find	Conj.	10864.53	183.27	2.87	65.53	15530.87
		DNF	15255.13	483.33	3.73	215.73	35430.13
	Update	Conj.	13316.87	244.40	1.72	139.27	28419.07
		DNF	10673	417.20	3.27	214.80	20666.80
	Delete	Conj.	17301.07	275.60	0	151.40	71060.87
		DNF	44357.93	481	0	238.47	82342.33
AWS	Find	Conj.	8592.13	184.73	2.73	60.47	16031.73
		DNF	17975.73	497.53	3.47	216.80	38353.53
	Update	Conj.	16405.10	261.13	1.07	185	26065.53
		DNF	10988.13	458.40	3.20	256	20707.80
	Delete	Conj.	40155.2	261.87	0	141.60	70170.13
		DNF	55.30.20	480.13	0	220.33	80167.20
GCP	Find	Conj.	10223.87	185.33	2.87	63.20	16467.03
		DNF	20400.73	459.20	3.53	214	45010.67
	Update	Conj.	10125.33	208.47	1.13	145.07	16231.52
		DNF	14469.20	408.47	3.33	245.07	26311.47
	Delete	Conj.	55906.07	296.60	0	153.73	81041.53
		DNF	47901	484.33	0	224.20	86517.07

5 RELATED WORK

CryptDB (Popa et al., 2011) was the first to suggest multi-layer encryption, encrypting each data element into one or more onions, with onion layers representing increasingly stronger encryption. CryptDB is a secure system for relational databases related to SQL queries over encrypted data, and it uses SEA. CryptMDB (Xu et al., 2017) is an efficient encryption system on MongoDB that uses a single additional cryptographic asymmetric encryption system encrypt data, thus performing only one additional operation, with no need for adjustment of techniques. SDDB (Almarwani et al., 2020a), CryptGraphDB (Aburawi et al., 2018b), and PPE schemes on WCSs (Waage and Wiese, 2017) apply the concept of CryptDB transfer to document databases (MongoDB), graph databases (Neo4j database), and Wide-column databases (Cassandra) respectively to execute queries on encrypted data through SEA. In (Shih and Chang, 2017) and (Wiese et al., 2020), CryptDB is moved to a wide column store; Crypt-NoSQL uses multiple proxies to distribute overhead; and CloudDBGuard removes the proxy functions to the client-side. (Aburawi et al., 2018a) proposes a traversal-aware encryption adjustment synchronized with query execution, thus improving security for the Simple technique on a Graph database.

6 CONCLUSION

This work introduced a Release-Aware In-Out encryption adjustment, expanding on previous work on Release-Aware Encryption Adjustment to improve performance by dynamically adjusting selective fields during query processing. Compared to SEA, RAIO reduces overhead in decryption costs and provides more security by exposing less information to the database servers. The proposal also supports backward adjustment to provide protection post query processing. RAIO can also be extended to outsourced databases such as cloud database providers. Our proposal overhead is most significantly affected by exchange communication between the database and proxy caused by MongoDB's lack of support for UDF. Future work includes extending the proposed policy to the larger classes of queries and further optimization, including reduction of communication cost and execution time. We intend to refine our policy's query behavior; we expect to use recent advances in cryptographic algorithms and find the best trade-off between protection and efficiency to integrate them into our future work policy.

REFERENCES

Aburawi, N., Coenen, F., and Lisitsa, A. (2018a). Traversal-aware encryption adjustment for graph databases. In *DATA*, pages 381–387.

Aburawi, N., Lisitsa, A., and Coenen, F. (2018b). Querying encrypted graph databases. In *Proceedings of the 4th International Conference on Information Systems Security and Privacy, ICISSP 2018, Funchal, Madeira - Portugal, January 22-24, 2018.*, pages 447–451.

Almarwani., M., Konev., B., and Lisitsa., A. (2019). Flexible access control and confidentiality over encrypted data for document-based database. In *Proceedings of the 5th International Conference on Information Systems Security and Privacy - Volume 1: ICISSP,*, pages 606–614. INSTICC, SciTePress.

Almarwani, M., Konev, B., and Lisitsa, A. (2020a). *Fine-Grained Access Control for Querying Over Encrypted Document-Oriented Database*, pages 403–425.

Almarwani, M., Konev, B., and Lisitsa, A. (2020b). Release-aware encryption adjustment query processing for document database. In *Proceedings of the 2020 4th International Conference on Cloud and Big Data Computing*, pages 48–51.

Bethencourt, J., Sahai, A., and Waters, B. (2007). Ciphertext-policy attribute-based encryption. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 321–334. IEEE.

Boldyreva, A., Chenette, N., and O'Neill, A. (2011). Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Annual Cryptology Conference*, pages 578–595. Springer.

Halevi, S. and Rogaway, P. (2003). A tweakable enciphering mode. In *Annual International Cryptology Conference*, pages 482–499. Springer.

Kamble, A. Basic aes and des implementation in java. https://github.com/AjitTK/JAVAEncryption. Accessed: 2020-9-1.

Popa, R. A., Redfield, C., Zeldovich, N., and Balakrishnan, H. (2011). Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100. ACM.

Savvides, S. Order-preserving encryption in java. https://github.com/ssavvides/jope. Accessed: 2020-9-11.

Shih, M.-H. and Chang, J. M. (2017). Design and analysis of high performance crypt-nosql. In *2017 IEEE Conference on Dependable and Secure Computing*, pages 52–59. IEEE.

Tutorial, A. (2017). Mongodb - java - tutorialspoint.

Waage, T. and Wiese, L. (2017). Property preserving encryption in nosql wide column stores. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 3–21. Springer.

Wang, J. Java realization for ciphertext-policy attribute-based encryption. https://github.com/junwei-wang/cpabe. Accessed: 2020-9-1.

Wiese, L., Waage, T., and Brenner, M. (2020). Clouddbguard: A framework for encrypted data storage in nosql wide column stores. *Data & Knowledge Engineering*, 126:101732.

Xu, G., Ren, Y., Li, H., Liu, D., Dai, Y., and Yang, K. (2017). Cryptmdb: A practical encrypted mongodb over big data. In *Communications (ICC), 2017 IEEE International Conference on*, pages 1–6. IEEE.