University of Amsterdam

Security of Systems and Networks

Security and Performance Analysis of Encrypted NoSQL Databases

February 12, 2017

Authors:
M.W. Grim BSc
A.T. Wiersma BSc

Supervisors:
F. Turkmen PhD

{Max.Grim, Abe.Wiersma}@os3.nl, F.Turkmen@uva.nl

Abstract

This research evaluates the performance and security of NoSQL databases, specifically MongoDB, implementing two types of encryption. These types of encryption are at rest encryption performed at the server and end-to-end encryption done by the client, where the server is able to perform queries over the encrypted data using Order Revealing and Partially Homomorphic encryption schemes. For the latter case we extended earlier work done by Alves et al. by adding functionality to their Python MongoDB connector wrapper. This research shows that enabling encryption introduces overhead in both the case of encryption at rest and the case of end-to-end encryption. Additionally this research shows that enabling end-to-end encryption prevents numerous attack vectors at the server side, albeit with the tradeoff of introducing a significant overhead in performance and limiting the number of supported queries.

Acknowledgements

We would like to thank Fatih Turkmen for his guidance and supervision as well as providing necessary information on the project. Also we would like to thank Pedro Alves for providing us help with setting up his proof of concept framework.

Contents

Chapter 1

Introduction

The security of private information is a challenge the IT branch slowly has been getting to terms with. Securing data residing in databases however is not as trivial. Attackers have a plethora of ways to gain unauthorised access to servers, either by using software vulnerabilities or by human error. These are not the only types of attack that pose a threat; Honest but curious individuals such as system administrators or developers may breach a database's confidentiality. This especially concerns databases hosted on Cloud infrastructures. Though popularity of these services increases, one should always consider the security risks that arise when hosting applications at Cloud providers.

These attacks can be countered by encryption the data stored in these databases. Encryption at rest is where the file system, directory or separate rows of a database are encrypted and decrypted by the server when moved from memory to storage. Full disk encryption has shown to be circumventable by using a cold boot attack on a servers memory [16]. This illustrates the fundamental problem with encryption at rest, namely that the server must be aware of the secret key. Additionally one should trust the Cloud provider hosting the database software to properly encrypt the plain data that is sent to them. Advantages of having the infrastructure in the Cloud is that it has a good infrastructure and often is cheaper to place all the data there.

Another option that might turn out to be safer to do is encrypting at the client side. This means that only the client is in possession of the secret key, ensuring end-to-end encryption. End-to-end encryption is growing in popularity, examples can be seen in for example chat applications as WhatsApp and Signal that promise to perform end-to-end encryption between clients. This however would mean losing query capabilities as the server is not able to read the data anymore.

In an effort to improve security and performance Popa et al. proposed a way of securing databases by using (No)SQL-aware encryption schemes, allowing (search) operations and computation on encrypted data that reside in the database while retaining confidentiality [25, 26]. Doing so allows clients to still query the database, without the need for the server to know the secret key.

While relational SQL databases still dominate the database market, the more efficient NoSQL databases are eating away at the SQL market share [3]. This transition is most apparent in big-data applications. As such this paper aims to evaluate encrypted NoSQL databases in terms of performance and security.

1.1 Research questions

- How is SQL-aware encryption realised in NoSQL database engines?
  - What kind of security does it provide?
  - How do (end-to-end) NoSQL-aware encryption schemes compare to regular encryption at rest schemes in terms of security?
- What is the performance impact of enabling encryption schemes on database engines under different types of workloads such as query processing over encrypted data?
  - What limitations are their in terms of functionality when enabling these encryption schemes?

1.2 Related work

This section presents an overview of relevant work done in the database security research field.

Previous research on the "*Security issues in NoSQL databases*": MongoDB and Cassandra, by Okman et al. determined the lack of on-disk encryption of the data-store to be problematic [23].

A later research by Noiumkar & Chomsiri "*A Comparison the Level of Security on Top 5 Open Source NoSQL Databases*" confirms again the lack of data encryption on disk by NoSQL databases [22]. Since then both Cassandra as well as MongoDB released encryption at rest for their enterprise product line.

Research on encrypting SQL databases was published in Popa et al. (2011) [26]. "*CryptDB: Protecting Confidentiality with Encrypted Query Processing*", which uses SQL-aware encryption to reduce security loss and decrease overhead in doing encryption for SQL databases.

Recently Poddar et al. (2016) continued the work done in CryptDB by adapting towards NoSQL-aware encryption with "*Arx: A Strongly Encrypted Database System*" [25]. Arx uses a similar infrastructure set-up as the previous CryptDB research, but uses AES as its strong basis instead of the onion encryption of CryptDB [26]. As the paper for Arx is still under review the publishers of the Arx database system chose to not disclose the system at this time.

1.2.1 A framework for searching encrypted databases

Using a smaller instruction set Alves & Aranha (2016) structured "*A Framework for Searching Encrypted Databases*" [6]. Alves & Aranha propose the use of Order Revealing Encryption over the use of Order Preserving Encryption which is used under Arx. As with Arx the encryption used is NoSQL-aware allowing for computation over encrypted data.

Bosch et al. (2015) [8] provides with "*A survey of provably secure searchable encryption*" which does evaluations for the numerous types of Searchable Encryption (SE). Authors of the aforementioned publications all reviewed their encryption using the Bosch survey.

1.3 Project outline

This report will continue with a summary on relevant subjects such as encryption at rest and homomorphic encryption in chapter 2. Chapter 3 describes the methodology used to test the performance of MongoDB with different types of encryption and how its security is evaluated. Results of these evaluations are presented in chapter 4. Finally, chapter 5 will draw conclusions from the collected data and chapter 6 will discuss the research done and propose future work to be done.

Chapter 2

Background

2.1 Encryption at rest

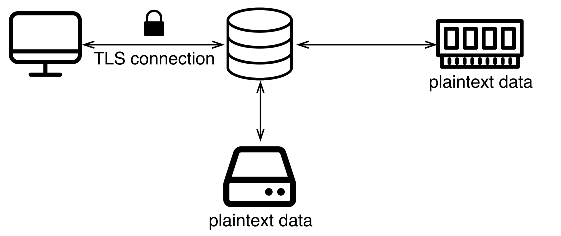
Encryption at rest is the term used to describe the encryption of the inactive data in a database. Similar technology is prevalent in the encryption of hard-disks and file-systems. When the inactive data is needed for operating on, the data is decrypted by the database application and stored as plaintext. In terms of change in infrastructure this equates in a change from Figure 2.1a to Figure 2.1b.

This type of encryption is usually done using proven encryption methods like AES or RSA [11, 17]. To ensure the best possible security the encryption keys should be stored separately from the encrypted data and be updated regularly. For larger amounts of data it is recommended to avoid using ECB mode AES as identical blocks of plaintext are encrypted into identical blocks of ciphertext. This would result in visible data patterns which is undesirable in regards to optimal security.

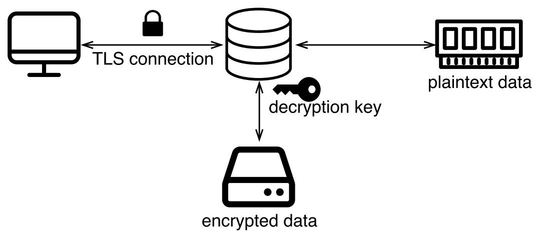
When the inactive data is stored it is as secure as your AES key, but the active data is as secure as its volatility. Because of the data remanence properties of computer memory its very possible to extract the un-encrypted contents of the memory by doing a "cold boot attack" [15].

2.1.1 MongoDB

MongoLabs natively supports encryption at rest since version 3.2 of their MongoDB Enterprise Advanced edition[1]. It uses the OpenSSL library to encrypt pages at application level using AES256-CBC. This improves performance as only modified pages need to be encrypted or decrypted [2].



(a) Encryption in flight using TLS. Data is only encrypted when in transit, and handled as plaintext on both the client and the server side.



(b) Encryption at rest on the server side. Once the data is written to disk it is encrypted. In the memory, data is still stored as plaintext.

[1]At the moment of writing MongoDB Enterprise Advanced is at version 3.4.1.

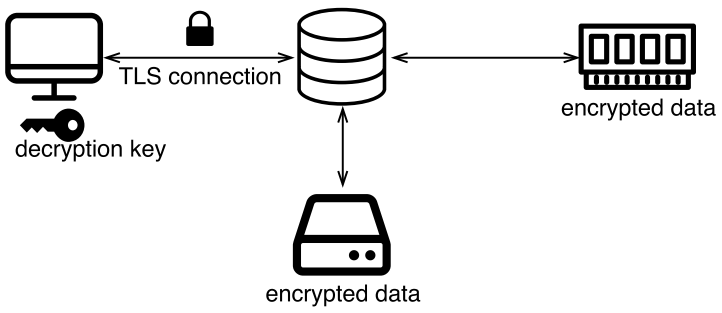


Figure 2.2: Diagram of database setup where the data is encrypted and decrypted by the client. This facilitates end-to-end encryption, where all the data in the database, whether it is stored on the disk or loaded into the memory, is encrypted.

Each node has its own key for encrypting and decrypting data, which is wrapped by an external master key which is supplied at the startup of the node. Keys can be stored at disk or on an external keyserver. Note that this means the server always needs access to the key by storing it in the memory. Memory is prohibited from being written to disk in unencrypted form using operating system calls [12, 20]. This makes sure that keys are never written to disk in unencrypted form.

2.2 Computation over encrypted data

A more secure solution would be to delay decryption of data to the moment it reaches the client. This ensures that sensitive data can not be compromised on the server, even when loaded into the memory. In terms of change required to the infrastructure this equates from a change from Figure 2.1b to Figure 2.2.

Consequently the database is not able anymore to read the data it stores for the client. But how can the database perform queries on its data if it can not even read its contents? This is where computation over encrypted data comes in. With computation over encrypted data the database server is able to do operations on encrypted data with the security that as little as possible data is leaking. To preserve data confidentiality and computability several encryption schemes are available. This ranges from encryption allowing for addition two encrypted values, multiplication of encrypted values (homomorphic encryption) or encryption that allows comparing two encrypted values and revealing order (ORE).

2.2.1 Order Revealing Encryption

Order revealing encryption (ORE) reveals order when fed into a publicly available function. Not to confuse with Order Preserving Encryption (OPE), for which Naveed et al. shown that it is vulnerable for “inference attacks” [21].

A sample is given in Figure 2.3 where two values, in this case 2003 and 2016, are encrypted using an ORE encryption scheme and a secret key. Once encrypted, a publicly available compare function can be used to evaluate $x > y$. The possible outcome of this function is illustrated in Equation 2.1.

When order can be evaluated, one can perform range queries, perform sorting, and can apply filtering on encrypted data [19].

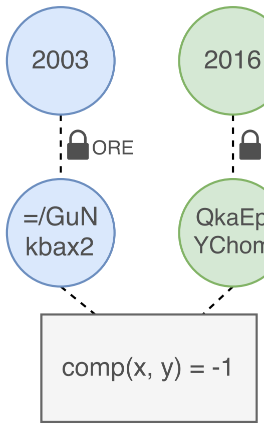


Figure 2.3: Order Revealing Encryption example with two values, 2003 and 2016 marked as green. Once encrypted using an ORE encryption scheme (marked as blue) they can be fed into a public compare function that will tell which of the two values is bigger or that they are equal.

$$comp(x,y) = \begin{cases} \text{-1} & \text{if } x < y \\ 0 & \text{if } x = y \\ +1 & \text{if } x > y \end{cases} \tag{2.1}$$

2.2.2 Homomorphic encryption

Homomorphic encryption schemes are a type of encryption schemes that allow computation on encrypted values. Operations can be executed on ciphertexts, and once its resulting ciphertext is decrypted it has the same outcome as if the operation would have been performed on the plaintext variant of the values [28].

In other words, an encryption scheme has the homomorphic property if Equation 2.2 holds for all secret keys k and for all values a and b. That is, each operation executed on the ciphertexts should (once decrypted) result in the same value as when the operation was executed on the plaintext values of a and b.

$$E_k(a) \circ E_k(b) = E_k(a \circ b) \tag{2.2}$$

Examples of homomorphic encryption schemes are: unpadded RSA [27], Goldwasser-Micali [14], Benaloh [7], ElGamal [13] and Pallier [24].

2.3 The Secure Mongo framework for querying encrypted databases

The Secure Mongo Framework applies one of five attributes to the fields in a MongoDB document:

1. static
2. index
3. h_add
4. h_mul
5. do_nothing

2.3.1 static

AES encryption is used to encrypt non-indexable fields in a MongoDB document. To do this the framework stores a SHA256 hashed plaintext password as its key. To encrypt the field field is padded to fill a multiple of 128-bits and an IV is generated. The padded field together with the key and IV are passed to the AES submodule of the PyCrypto library[2]. The encrypted data together with the prepended IV are stored in the database.

2.3.2 index

To encrypt an index in the Secure Mongo framework the Order-Revealing Encryption (ORE) described in "Order-revealing encryption: New constructions, applications, and lower bounds" [19] is used. A field with the index attribute is stored both in the regular collection using AES as well as in a separate index-collection using ORE. The ORE from the aforementioned paper was implemented in C in the FastORE[3] framework. To make its functionality available in Python, a C to Python binding was made by the authors of the Secure Mongo framework. To start doing ORE encryption a secret key is generated using AES key expansion. The ORE secret key is actually two AES keys, the PRF-key for deriving the keys from each prefix and the PRP-key for permuting the slots within a block [19]. The framework then uses this secret key to encrypt plaintext[4] into a left and right encryption. A left encryption combined with a right encryption allow for an evaluation based of off a pseudo random function of the two:

- -1, for when the left encryption evaluates to a smaller number than the right encryption.
- 0, for when the left encryption evaluates to the same number than the right encryption.
- 1, for when the left encryption evaluates to a bigger number than the right encryption.

This comparison function allows for the traversal of a serialised AVL-tree in the database. A node in the serialised AVL-tree stores the right encryption, the id's of its left/right child and the indexes of the documents that contain the AES-encrypted value corresponding with the ORE right encryption ciphertext in the node. This structure allows index queries over encrypted data, by first doing a lookup using ORE for the correct node in the index-collection and then doing a lookup for the indexes found in this node in the "regular" collection.

[2]PyCrypto library: https://pypi.python.org/pypi/pycrypto
[3]https://github.com/kevinlewi/fastore
[4]Currently only a size up to 64-bit integers are supported

2.3.3 h_add

To be able to do addition and subtraction over an encrypted field in a MongoDB document the h_add attribute can be specified for that field. The Secure Mongo framework which has generated a public / private key-pair for use with Paillier homomorphic encryption, can then encrypt the field using the Paillier public key. When values need to be added to a Paillier encrypted field this can be done with minimal information leakage. This operation can be formalised as follows:

$$D(c_1 \cdot c_2 \bmod n^2) = m_1 + m_2 \bmod n \tag{2.3}$$

The multiplication of two ciphertexts $\bmod n^2$ evaluates to the same value as the addition of the two plaintexts.

2.3.4 h_mul

To complete the basic set of arithmetic operations the h_mul attribute can be specified to allow multiplication for a field in a MongoDB document. To achieve this the homomorphic properties of the ElGamal cryptosystem are used. First of all the framework needs to generate the keys, a public and private component, which are derived from a cyclic group G of order q with generator g. A random x is then chosen with $x \in \{1...q-1\}$ and h is calculated as follows: $h := g^x$. The group of G, q and g together with h are the public key and the x is used as public key. The encryption of plaintext m is formalised as follows: $E(m) = (g^r, m \cdot h^r)$ with r a random with $r \in \{1...q-1\}$. The homomorphic property of ElGamal can then be defined as follows:

$$E(m_1) \cdot E(m_2) = (g^{r_1}, m_1 \cdot h^{r_1})(g^{r_2}, m_2 \cdot h^{r_2}) \tag{2.4}$$
$$(g^{r_1}, m_1 \cdot h^{r_1})(g^{r_2}, m_2 \cdot h^{r_2}) = (g^{r_1+r_2}, (m_1 \cdot m_2)h^{r_1+r_2}) \tag{2.5}$$
$$(g^{r_1+r_2}, (m_1 \cdot m_2)h^{r_1+r_2}) = E(m_1 \cdot m_2) \tag{2.6}$$

The homomorphic properties of ElGamal extend into additive homomorphism, as discussed in [6], by encrypting m as the exponent of g, like: g^m. In the case of a small m this would be considerable, but calculation of the discrete log on decryption gets more expensive as m increases.

2.3.5 do_nothing

Finally the attribute to not encrypt a field, "do_nothing" is defined to store the field as plaintext.

Chapter 3

Approach & Methods

This chapter will start by describing the changes made to the Secure Mongo framework. Followed by the description of the benchmarking frameworks used to load-test the different MongoDB set-ups and the experiments run.

3.1 Extending the Secure Mongo framework

The first half of the BenchmarkDB benchmark consists of doing timed sequential inserts, but this was impossible to do in the initial Secure Mongo framework. Because an overhaul of the bulk insert and client side AVL-tree generation was possible within the available time, sequential inserts were implemented as part of this research. Not only is the implementation of sequential inserts important for the benchmarking, but also because in real-life applications having an immutable dataset is extremely impractical.

As a first measure the AVL-tree storage was moved from memory based to MongoDB based, this obviously meant moving from low lookup times from memory to high lookup times from the networked database. All AVL-tree operations, namely inserts and balancing, were still being done at the application. To reduce the number of times the network needs to be traversed these operations could be implemented on the server-side. This means to either implement these operations into the MongoDB source code or to define them in the MongoDB stored operations[1].

Because of time-constrains on this project the decision was made to try and implement these operations within the Javascript stored operations mentioned previously. The ORE requires the AES-cryptosystem and because Javascript has no built-in libraries for AES, tree comparisons are still being done on the application. The AVL-tree balancing on-insert however is being done on the server by Javascript defined methods. The changes made to the Secure Mongo framework are illustrated in Figure 3.1.

[1]https://docs.mongodb.com/manual/core/server-side-javascript/

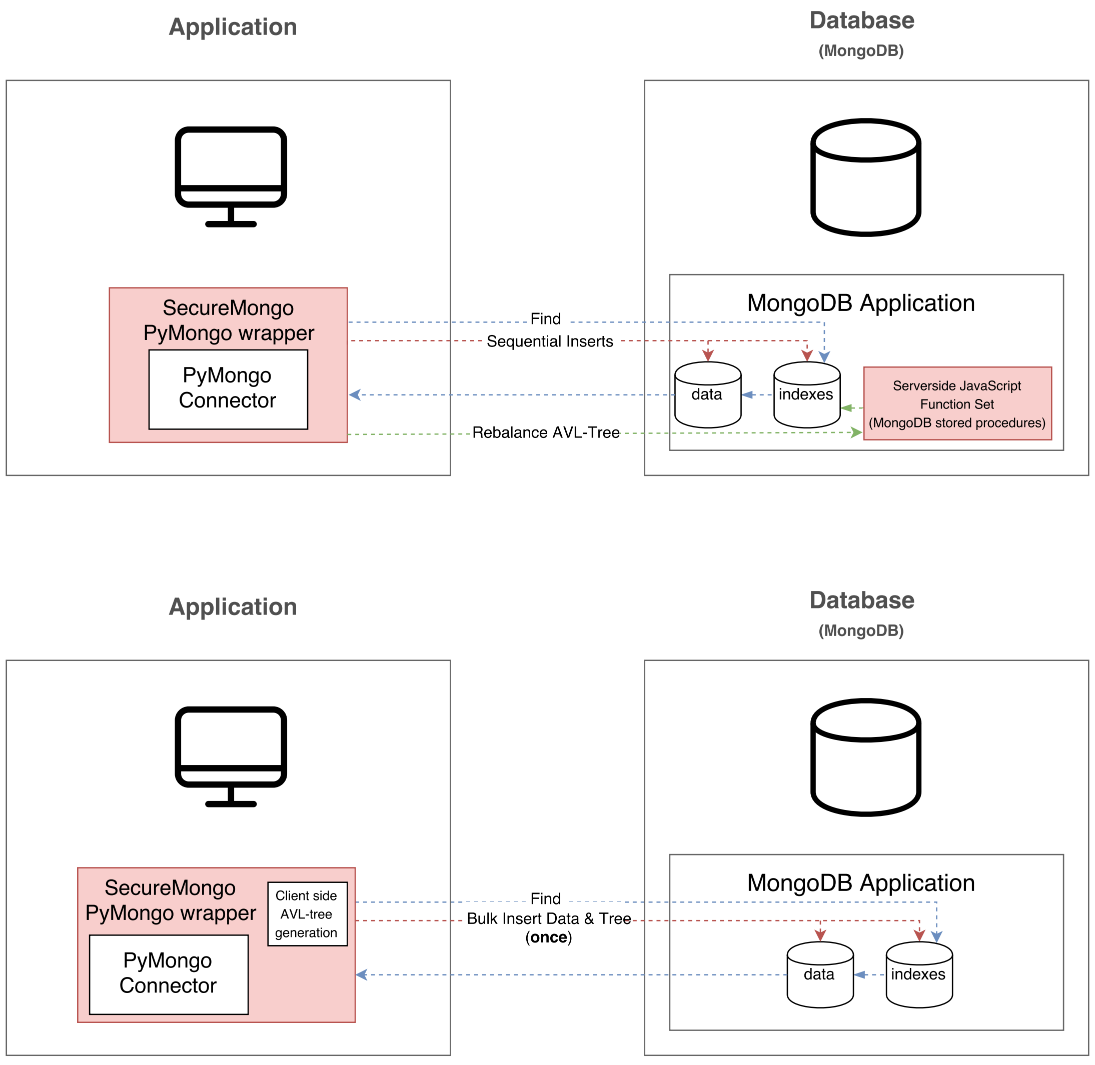


Figure 3.1: Changes made for this research to the Secure Mongo framework [6], with the original work shown in the bottom of the figure and the work for this project shown in the top of figure.

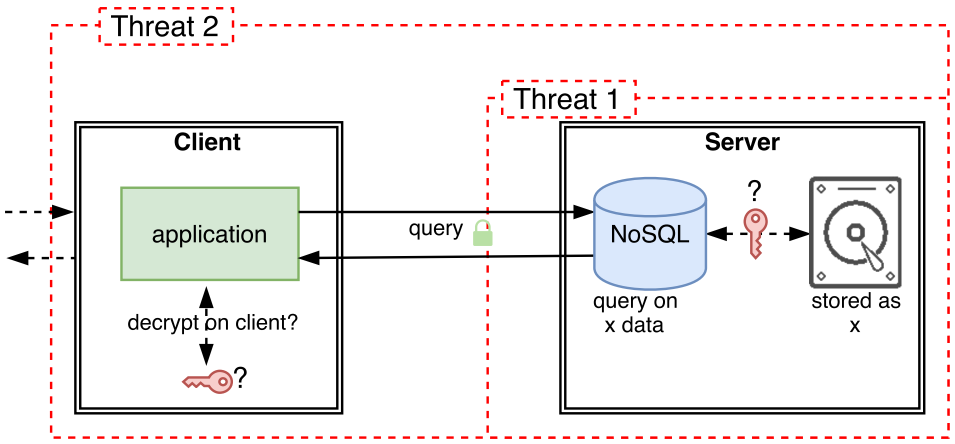


Figure 3.2: Diagram showing two types of threats for a NoSQL infrastructure.

3.2 Evaluating security

We will evaluate the security of both encryption at rest and NoSQL-aware encryption schemes. For this we will evaluate the properties of the encryption schemes and what makes them suitable for certain situations. This includes possible attacks on both encryption at rest and NoSQL aware encryption of both resident and instant adversaries. These attacks can be split up in two threat types (see Figure 3.2): a threat in which the database server is compromised as shown with Threat 1 and an arbitrary threat in which the database server and application server may be compromised arbitrarily as shown with Threat 2.

]b[

Listing 3.1: Movie

```
{
  "title":"Rain Man",
  "Index":1988
}
```

3.3 Evaluating performance

Evaluating performance of databases is generally done by performing benchmarks. To assess the performance of MongoDB under different methods of encryption we ran benchmarks with different parameters. Benchmarks simulate a workload and measure how the database performs.

The encryption at rest configuration and the Secure Mongo configuration use a different benchmarking tool because of several reasons; Firstly the number of operations that is possible in the Secure Mongo framework is limited, therefore allowing less intricate benchmarks to be done, secondly because our implementation of the Secure Mongo framework is limited in types of operations it can perform and because it is written in a specific language, namely Python. Both the encryption at rest configuration and the Secure Mongo framework will be benchmarked in relation to a "default" set-up using benchmarking frameworks.

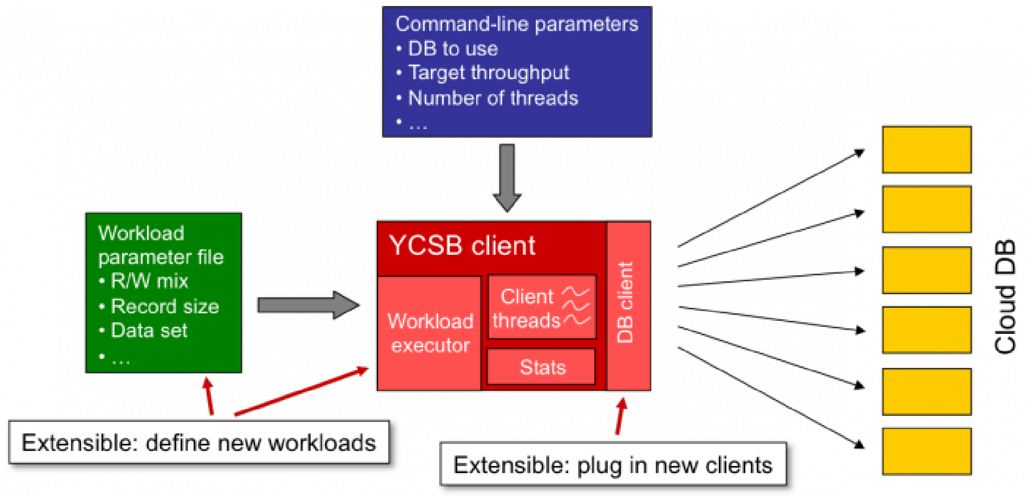


Figure 3.3: Diagram of YCSB [9].

3.3.1 Encryption at rest

As a benchmarking framework for the encryption at rest configuration YCSB was used. YCSB is a generic and extensible framework that is tailored for generating load on (non-relational) data stores while measuring its performance [5, 10]. It has built-in support for many database clients including HBase, Cassandra and MongoDB.

The main component of the YCSB framework is the client (see Figure 3.3). The client consists of a workload executor, multiple client threads, a database client, and a statistical module. This client is called with command-line parameters that specify the database connection, the number of client threads, and the workload parameter file. As YCSB supports different kinds of stores it refers to records and fields, where in MongoDB records are documents and fields are fields inside the document, all stored within one collection.

The YCSB client initiates a specific number of client threads that connect to the data store using the appropriate database client. In our case, this database client will be a MongoDB connector implemented in the same language as the framework, namely Java. After all threads are connected successfully, the YCSB client will start the workload executer.

The framework provides one default workload executer called the "core workload" executer which simulates a user database containing random data. The core workload accepts multiple parameters customising the workload simulation. Parameters such as the number of records and fields, the number of operations, the proportion of read-, write- and update queries, and the random distribution that it will use to simulate the load. Table 3.1 shows the workload used for the benchmarks, which is an update heavy workload with a 50/50 distribution of reads and updates.

workload	com.yahoo.ycsb.workloads.CoreWorkload
recordcount	16,000,000
operationcount	100,000
readproportion	0.5
updateproportion	0.5
requestdistribution	zipfian

Table 3.1: Workload parameters used for the benchmarks.

Listing 3.2: Inserted data record.

```
{
  "_id":"user7456755629803624101",
  "field1":{
    "$binary":"PCU/Kj0sPDcg0jsyKjoxLiAqNiA6Nzs4J...",
    "$type":"00"
  },
  "field0": ...
}
```

Before a workload can be executed first data has to be loaded into the database, for which also the performance can be measured. For this YCSB will generate random user data that it will store into the database. An example of the data that YCSB randomly generates is shown in Listing 3.2.

Once all the random user data is loaded into the database the second step is to run the simulated workload over the data, where both reads and updates are executed. A read is executed by requesting a random document by its ID from the database. An example of such a query is shown in Listing 7.1.

An update is executed by updating a random field in a user record by its user id. An example of such a query is shown in Listing 7.3. These queries are collected by using the profiling mechanism of MongoDB. We measure the throughput of the database in operations per second.

Both benchmarks were done on a MongoDB[2] instance that used encryption at rest, and a MongoDB instance that did not use encryption at rest. For this we used two servers (see Table 3.2 for the machine specifications), one which ran the YCSB client, and one that ran the MongoDB instance. The machines are connected with a 1Gbps link. As YCSB codebase we will use a specific fork of YCSB that MongoLabs also used for their benchmarks [1, 2, 4]. As told by the people at MongoLabs:

"YCSB has not been actively maintained in the recent past and here is where you will find the current version of what we believe is correct version of MongoDB YCSB DB Client."

All experiments ran concurrently on eight threads, one thread per core.

3.3.2 Computation over encrypted data

To evaluate the performance of the Secure Mongo framework, which was written in Python code, we needed a comprehensible Python benchmarking framework that supported the extension of

Machine	PowerEdge R230
Processor	Intel Xeon CPU E3-1240L v5 @ 2.10GHz
OS	Ubuntu 16.04.1 LTS
Memory	2x DDR-4 DIMM 8192MB memory @2133 MHz
Disk	1TB Toshiba HDD (SATA)

Table 3.2: Server specifications.

[2]MongoDB Enterprise v3.4.1

its connectors. Because of the strict time schedule of the project, an effort to re-implement the Secure Mongo framework in Java, YCSB's native language, was deemed unattainable.

For this purpose we found BenchmarkDB [18], a small Python benchmarking framework that allows easy addition of new modules. BenchmarkDB already had a module that allowed for benchmarking MongoDB, which we copied and modified for use with the Secure Mongo framework.

While BenchmarkDB originally used randomly generated fields to be inserted, we wanted to query actual data from the database to visualise the queries. We chose to import a part of the IMDB database, where we inserted a document containing the title and year of X number of movies into the database (see Listing 3.3). Once inserted those same movies were retrieved again from the database, measuring the time it took to insert and retrieve those movies.

Our implementation as will make use of two collections, one for storing the index and one for storing the actual data. We will compare its performance to that of ordinary inserts, that do not perform any encryption and does not make use of encrypted indexes stored in another database, but will make use of a native index that MongoDB provides[3].

Listing 3.3: Movie

```
{
  "title":"Rain Man",
  "Index":1988
}
```

[3]https://docs.mongodb.com/manual/indexes/

Chapter 4

Results

This chapter will start by describing the performance results that are gathered while running benchmarks on MongoDB under different configurations, followed by the results gathered during the security analysis of MongoDB under these same configurations.

4.1 Performance

The results of the performance benchmarks consist, as discussed in the method, out of two parts. As these parts are incompatible in terms of the benchmark software that they use, they will be discussed in separate subsections. This section will start by describing the performance evaluations of encryption at rest, followed by the Secure Mongo implementation that allows queries over encrypted data.

4.1.1 Encryption at rest

For these benchmarks we considered the performance impact of enabling encryption at rest for MongoDB Enterprise Advanced. The benchmark consists out of two phases: the loading phase and the running phase. In the loading phase records are only inserted, whereas in the running phase records are either read or updated. The first benchmark consists of two stages: the loading stage (insert operations) and the running stage (read/update operations), where as discussed earlier in this report reads and updates are executed randomly in a 50/50 zipfian distribution. In the load stage 16 million rows were inserted, and this benchmark was executed 40 times in total. The running stage executed a total of 100000 operations and this benchmark was run 107 times in total. These statistics are shown in Table 4.1.

Figure 4.1 shows the throughput distribution of MongoDB. This figure clearly shows a throughput overhead in the encryption at rest case for both insert operations and for the combination of read and update operations, as their peaks lie more to the left of the chart area. This is further illustrated in Figure 4.2, which clearly shows the median throughput being lower

Type	**Operations**	**Runs**
insert	16000000	40
read / update	100000	107

Table 4.1: Number of operations performed per benchmark and the number of benchmarks performed overall for both insert benchmarks and read and update benchmarks.

in both types of operations for the encryption at rest configuration. The median throughput is illustrated further in Figure 4.2. Enabling encryption at rest introduces an overhead of 4.71% for insert operations and an overhead of 6.74% in terms of throughput for read/update operations. These results are shown in Table 4.2.

The same results can be observed when using the median latency as a measure. The median latency for operations is illustrated further in Figure 4.3. Enabling encryption increases insert latency by 5.24%, read latency by 7.41% and update latency by 7.51%. These results are shown in Table 4.3.

	Insert (ops/s)	**Read / update (ops/s)**
not encrypted	9915	193
encryption at rest	9448	180

Table 4.2: Throughput (operations per second) for both insert operations and the combination of read and update operations.

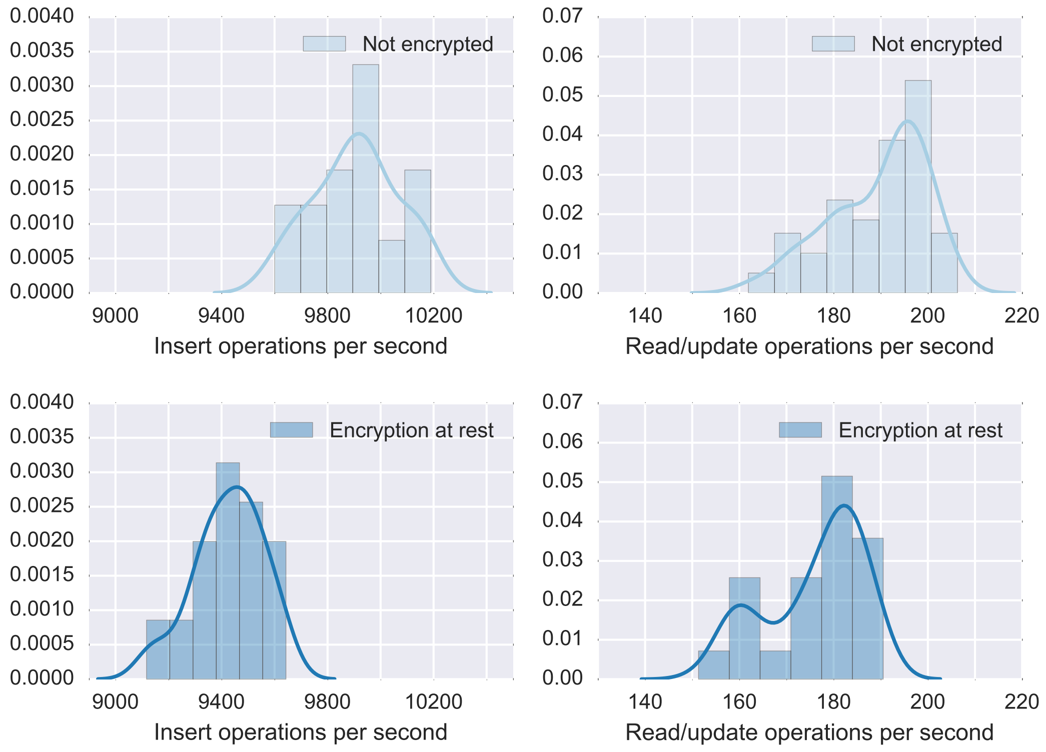


Figure 4.1: Throughput (operations per second) distribution for both insert operations and the combination of read and update operations.

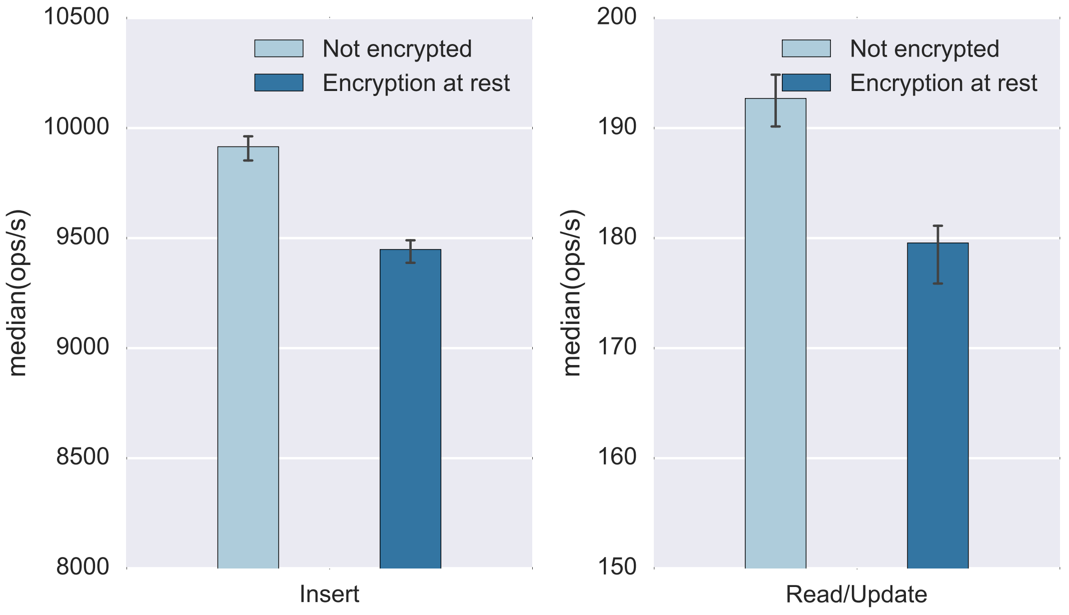


Figure 4.2: Throughput (operations per second) for both insert operations and the combination of read and update operations.

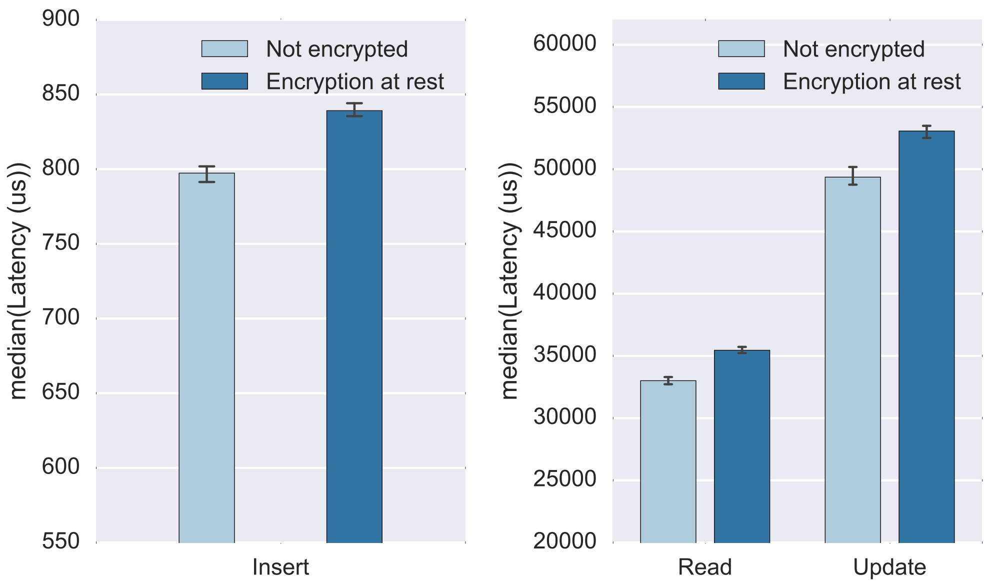


Figure 4.3: Median latency for insert and read/update operations.

		Median (us)	StDev (us)	Min. (us)	Max. (us)
insert					
	not encrypted	839.187	11.965	776.707	824.501
	encrypted	797.371	13.369	822.813	870.777
read					
	not encrypted	33007.012	1634.307	30711.253	38280.328
	encrypted	35452.194	2257.168	33108.312	45610.949
update					
	not encrypted	49360.605	3313.949	45198.531	61627.781
	encrypted	53065.576	3418.262	49201.372	63014.574

Table 4.3: Latency statistics for operations in microseconds.

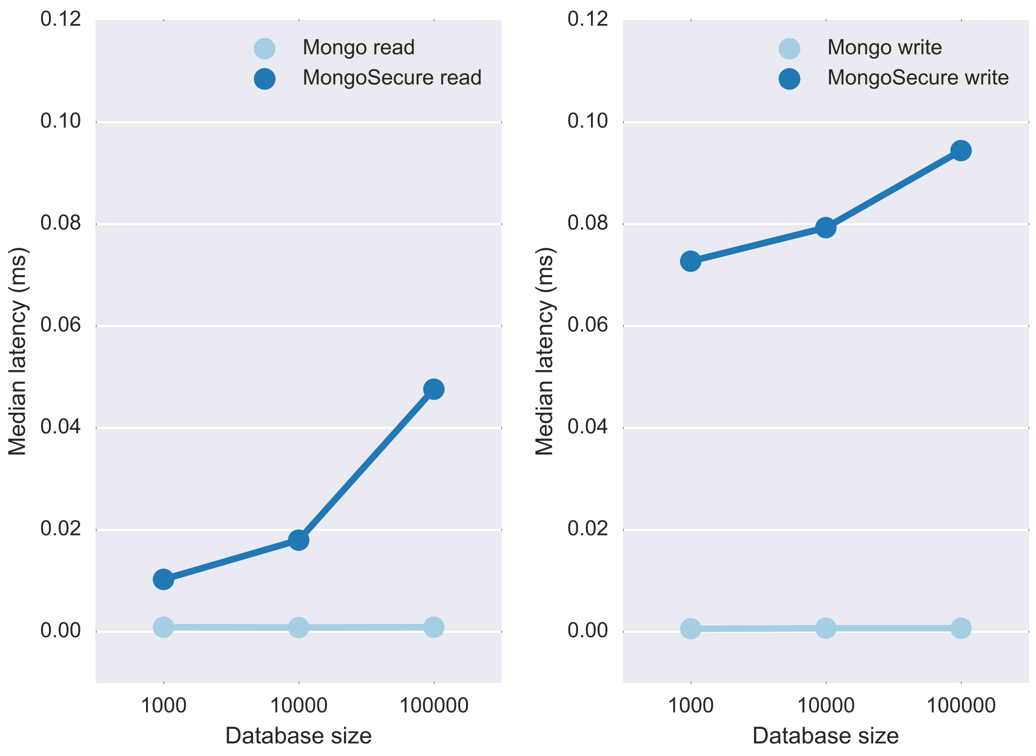


Figure 4.4: Median latency (ms) for read and write operations on a default vs computation over encrypted data MongoDB server.

4.1.2 Secure Mongo

For these benchmarks we considered the performance impact of using the Secure Mongo framework with MongoDB Enterprise Advanced. The benchmark consists of two phases: the writing phase and the reading phase. In the writing phase records are only inserted, whereas in the reading phase, records are either searched by Index and read. In the both phases 1000, 10000 or 100000 documents were written or read. Table 4.4 shows the number of times each of these benchmarks ran. The results of these read and write benchmarks are shown in Table 4.5 and 4.6 respectively.

Figure 4.4 shows the median latency for the read and write operations in the benchmarks. This figure clearly shows a significant overhead over default MongoDB for the Secure Mongo framework in both read and write operations. The largest difference in latency is found for the write operations, but the biggest increase in latency over database size is found with the read operations. This overhead is more clearly specified in the **Overhead** column of Tables 4.5 and 4.6.

Documents	Mongo	Runs
1000		
	Not encrypted	22
	Secure Mongo	58
10000		
	Not encrypted	22
	Secure Mongo	124
100000		
	Not encrypted	26
	Secure Mongo	14

Table 4.4: Number of times each individual benchmark was run.

Documents	Mongo	Median (ms)	Overhead	Min. (ms)	Max. (ms)	StDev (ms)
1000						
	Not encrypted	0.000903	1x	0.000440	0.001093	0.000122
	Secure Mongo	0.010280	11.4x	0.002767	0.013390	0.002116
10000						
	Not encrypted	0.000838	1x	0.000311	0.00185	0.000165
	Secure Mongo	0.017993	21.5x	0.005162	0.024106	0.002318
100000						
	Not encrypted	0.000899	1x	0.000262	0.001992	0.000170
	Secure Mongo	0.047606	53x	0.032493	0.073872	0.004420

Table 4.5: Latency statistics for read operations in milliseconds of a default MongoDB server vs. a MongoDB server doing computation over encrypted data.

Documents	Mongo	Median (ms)	Overhead	Min. (ms)	Max. (ms)	StDev (ms)
1000						
	Not encrypted	0.000587	1x	0.000310	0.082469	0.002597
	MongoSecure	0.072722	123.9x	0.048533	0.382577	0.017212
10000						
	Not encrypted	0.000696	1x	0.000298	0.002177	0.000171
	MongoSecure	0.079310	114.0x	0.048415	0.362210	0.015046
100000						
	Not encrypted	0.000697	1x	0.000223	0.149140	0.000501
	MongoSecure	0.094429	135.5x	0.047828	0.379993	0.016588

Table 4.6: Latency statistics for write operations in milliseconds of a default MongoDB server vs a MongoDB server doing computation over encrypted data.

4.2 Security

4.2.1 Threat 1: Database server compromise

In this threat model, the attacker can be considered a curious database administrator or an external attacker with full access to the data stored by the database and the machine itself. The attackers purpose is to steal data, not to invalidate the integrity of the data or to disrupt availability. The focus is confidentiality, not integrity or availability.

Default MongoDB

By default MongoDB stores its data as plaintext. This means an attacker would have unrestricted access to all of the documents maintained in the MongoDB database when access to the database server is realised.

MongoDB with encryption at rest

When MongoDBs optional encryption at rest is enabled the inactive data is stored as AES encrypted ciphertext, see Figure 2.1b, and the active data is stored as plaintext in memory. This leaves the database server vulnerable to cold-boot attacks, allowing the attacker plaintext values of all active data. Also because decryption is done on the database server the key is required, leaving three possible locations for the attacker to retrieve the key from.

1. Cold-boot extraction from memory (always).
2. Extract from hard-disk (if key is stored on disk).
3. Retrievable from secondary server by posing as the database-server (possibly negated by two factor key retrieval).

The overall benefit of enabling encryption at rest is that the database data on the hard-disk is now useless to the attacker. **The AES** type of encryption used is IND-CPA secure as AES-256CBC via OpenSSL is used to encrypt and decrypt text. OpenSSL is used in compliance with FIPS 140-2.

MongoDB utilising Computation over Encrypted data

When an implementation of computation over encrypted data is combined with MongoDB the server is at no point in time aware of the data it is storing, all data on the server is encrypted with either AES [11], ORE [19], ElGamal [13] or Paillier [24]. **The AES** encryption as with encryption at rest enabled is IND-CPA secure as AES-128CBC is used in combination with a randomly generated IV for every encryption. **The Order-Revealing Encryption** proposed by Lewi and Wu offers IND-OCPA and goes as far as limiting the use of the compare function by decomposing the ciphertext into a left and right component. As only the right component is stored, no two ciphertexts can be compared. The implementation of Alves and Pedro [6] however uses an AVL-tree to allow lookups over indexes, which leaks the information hidden by only storing the right-component. Moreover the ORE by Lewi and Wu leaks the first block of d-bits that differs [19]. In which d is the block size in bits for the ORE scheme. **ElGamal** is proven semantically secure against Chosen Plaintext Attacks, meaning that as long as the attacker doesn't get access to the encryption function for a plaintext, plaintexts cannot be deduced from ciphertexts. Because the encryption in this case is done application-side, this scheme is secure. **Paillier** is proven to provide semantic security against Chosen Plaintext Attacks, which hangs on the decisional composite residuosity assumption. As with ElGamal, this means that as long as the attacker is not able to access the encryption function, which is still the case, the scheme is secure from this threat. The nodes in the **AVL-tree** leak the indexes of the data-collection they point to, allowing the attacker to deduce relationships between encrypted documents in the data-collection.

4.2.2 Threat 2: Arbitrary threats

We now observe a second threat model in which the application server and database server are compromised arbitrarily.

Default MongoDB

Because the attacker already had full access to the data, more cannot be gained.

MongoDB with encryption at rest

Because plaintext active data and key could already be retrieved from the memory, the additional scope of the threat only means the decrypted data can more easily be retrieved. The addition of the application server to the scope enables the attacker to make the database decrypt the encrypted at rest data by impersonating this server.

MongoDB utilising Computation over Encrypted data

The attack-surface for an application using computation over encrypted data is similar to the attack-surface for the database server in threat 1. This means that plaintext data is temporarily stored in the memory for the duration it is used. Moreover the keys for decryption need to be stored in the memory leaving the application server vulnerable to cold-boot key retrieval. Depending on the implementors decision the key could be retrieved from the following locations:

1. Cold-boot extraction from memory (always).
2. Extract from hard-disk (if key is stored on disk).
3. Retrievable from secondary server by posing as the application-server (possibly negated by two factor key retrieval).

Chapter 5

Conclusion

This research has evaluated the performance of encrypting NoSQL databases in particular that of the MongoDB NoSQL database. It has extended the capabilities of the Secure Mongo framework, by adding sequential inserts and tested this extended performance. Using two benchmarking frameworks, YCSB and BenchmarkDB, the overhead of doing two types of encryption on a MongoDB server was estimated. These benchmarks showed a small overhead for doing encryption at rest and a relatively big overhead for using the Secure Mongo framework.

On the other hand the security provided by the Secure Mongo framework displaces the threat of doing cold-boot attacks from the database server to the application server. This means the database server could be a completely untrustworthy entity in the application infrastructure when the Secure Mongo framework is used.

Taken as a whole the results provide an insight into the overall trade-off of performance versus security. The more steps are taken to assure data confidentiality, the more the performance of the database engine decreases.

Chapter 6

Discussion & Future Work

6.1 Discussion

Preferably the complete benchmarking would have been done with only one benchmarking framework, but as the authors of the MongoDB code did their benchmark in this exact same type of set-up, and YCSB could not be used for the benchmarking of the Secure Mongo framework due to incompatibilities between YCSB and the PyMongo driver, this was the compromise.

Due to the fact that MongoDB enterprise (closed-source) is the only MongoDB that has encryption-at-rest available as a feature, source-code inspection of their implementation of the encryption was impossible. This meant no review of the key management and implementation of the AES-cryptosystem was done.

The initial source-code of the Secure Mongo framework had quite a steep learning curve into making it work. In the end the result of the project was also a cleaned up and trimmed version of the Secure Mongo framework with sequential inserts added as a feature.

Due to time restrictions no performance measurements of the partial homomorphic encryption was done. The ORE was not implemented to it's fullest possibilities as ORE also supports range-queries, sorting and filtering. ORE currently only makes it possible to index on integers which for actual database usage would make it considerably impractical. With the benchmarks done on the Secure Mongo framework no parallel writes were done to test consistency, which hypothetically is something the current implementation of the AVL-tree was not designed for.

The MongoDB NoSQL system was presented as a general NoSQL database to test these types of encryption schemes. While most of the encryption schemes proposed here are quite modular, it would be better to test the performance of other NoSQL databases i.e. ones that are not document stores.

6.2 Future work

The feasibility of using a MongoDB Javascript server-side proxy to cut-out network latency in AVL-tree traversal was tested. Conclusion was that implementing AES-CBC encryption with Javascript was beyond the scope of the project and most likely very in-efficient. Implementation of a server-side proxy built-in with the MongoDB source-code however could be quite efficient as the FastORE C-code can be dropped in seamlessly with the MongoDB C++.

Native indexing built into MongoDB. That is, MongoDB already uses B-tree implementations for indexes. If the implementation of an ORE comparator would be written for these indexes the tree-traversal could speed up. A possible downside of this approach would be that MongoDB has to be modified in order to work, whereas our current solution allows to use an arbitrary (Cloud) instance of MongoDB.

Appendix

This appendix contains pieces of code, queries, et cetera that are too long to place in the chapters of the report.

Listing 7.1: Read query.

```
{
  "op":"query",
  "ns":"ycsb.usertable",
  "query":{
    "find":"usertable",
    "filter":{ "_id":"user6047990667576834393" },
    "ntoreturn":-1
  },
  "keysExamined":1,
  "docsExamined":1,
  "cursorExhausted":true,
  "numYield":0,
  "locks":{
    "Global":{
      "acquireCount":{
        "r":NumberLong(2)
      }
    },
    "Database":{
      "acquireCount":{
        "r":NumberLong(1)
      }
    },
    "Collection":{
      "acquireCount":{
        "r":NumberLong(1)
      }
    }
  }, ...
}
```

Listing 7.2: Insert query.

```
{
  "op":"insert",
  "ns":"ycsb.usertable",
  "query":{
    "insert":"usertable",
    "ordered":true,
    "documents":[
      {
        "_id":"user7456755629803624101",
        "field1":{
          "$binary":"PCU/Kj0sPDcg0jsyKjoxLiAqNiA6Nz...",
          "$type":"00"
        },
        "field0": ...
      }
    ]
  },
  "ninserted":1,
  "keysInserted":1,
  "numYield":0,
  "locks":{
    "Global":{
      "acquireCount":{
        "r":NumberLong(1),
        "w":NumberLong(1)
      }
    },
    "Database":{
      "acquireCount":{
        "w":NumberLong(1)
      }
    },
    "Collection":{
      "acquireCount":{
        "w":NumberLong(1)
      }
    }
  }, ...
}
```

Listing 7.3: Update query.

```
{
  "op":"update",
  "ns":"ycsb.usertable",
  "query":{ "_id":"user708853751256151 83" },
  "updateobj":{
    "$set":{
      "field2":{
        "$binary":"JC8wOTQhJyYyKjsqJjAyNiAxICQwNyY...",
        "$type":"00"
      }
    }
  },
  "keysExamined":1,
  "docsExamined":1,
  "nMatched":1,
  "nModified":1,
  "numYield":0,
  "locks":{
    "Global":{
      "acquireCount":{
        "r":NumberLong(1),
        "w":NumberLong(1)
      }
    },
    "Database":{
      "acquireCount":{
        "w":NumberLong(1)
      }
    },
    "Collection":{
      "acquireCount":{
        "w":NumberLong(1)
      }
    }
  },...
}
```

Bibliography

[1] https://www.mongodb.com/partners/partner-program/technology/certification/encryption-best-practices.

[2] At-test encryption in MongoDB 3.2: features and performance. https://www.mongodb.com/blog/post/at-rest-encryption-in-mongodb-3-2-features-and-performance. Accessed: 2017-01-23.

[3] DB-engines ranking. http://db-engines.com/en/ranking. Accessed 2017-01-23.

[4] mongodb-labs/YCSB. https://github.com/mongodb-labs/YCSB. Accessed: 2017-01-23.

[5] YCSB. https://github.com/brianfrankcooper/YCSB. Accessed: 2017-01-11.

[6] Alves, Pedro. A framework for searching encrypted databases. In *Anais do XVI Simpsio Brasileiro em Segurana da Informao e de Sistemas Computacionais (SBSeg 2016)*, 2016.

[7] Josh Benaloh. Dense probabilistic encryption. In *Proceedings of the workshop on selected areas of cryptography*, pages 120–128, 1994.

[8] Christoph Bösch, Pieter Hartel, Willem Jonker, and Andreas Peter. A survey of provably secure searchable encryption. *ACM Computing Surveys (CSUR)*, 47(2):18, 2015.

[9] Brian F. Cooper. Yahoo! Cloud Serving Benchmark. ACM Symposium on Cloud Computing, 2010.

[10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[11] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[12] die.net. mlock(2) - Linux man page. https://linux.die.net/man/2/mlock. Accessed 2017-02-08.

[13] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

[14] Shafi Goldwasser and Silvio Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 365–377. ACM, 1982.

[15] Michael Gruhn and Tilo Muller. On the practicability of cold boot attacks. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 390–397. IEEE, 2013.

[16] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[17] J. Jonsson and B. Kaliski. Public-key cryptography standards (pkcs) #1: Rsa cryptography specifications version 2.1, 2003.

[18] Kurtis Jungersen. BenchmarkDB. `https://github.com/kmjungersen/BenchmarkDB`. Accessed: 2017-01-27.

[19] Kevin Lewi and David J Wu. Order-revealing encryption: New constructions, applications, and lower bounds. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1167–1178. ACM, 2016.

[20] Microsoft. VirtualLock function. `https://msdn.microsoft.com/en-us/library/windows/desktop/aa366895(v=vs.85).aspx`. Accessed 2017-02-08.

[21] Muhammad Naveed, Seny Kamara, and Charles V Wright. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 644–655. ACM, 2015.

[22] Preecha Noiumkar and Tawatchai Chomsiri. A comparison the level of security on top 5 open source NoSQL databases. In *The 9th International Conference on Information Technology and Applications (ICITA2014)*, 2014.

[23] Lior Okman, Nurit Gal-Oz, Yaron Gonen, Ehud Gudes, and Jenny Abramov. Security issues in NoSQL databases. In *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 541–547. IEEE, 2011.

[24] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 223–238. Springer, 1999.

[25] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: a strongly encrypted database system.

[26] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100. ACM, 2011.

[27] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[28] Xun Yi, Russell Paulet, and Elisa Bertino. *Homomorphic encryption and applications*, volume 3. Springer, 2014.