Bachelor Thesis

Analysis of Encrypted Databases with CryptDB

Michael Skiba

Date:	09.07.2015
Supervisor:	Prof. Jörg Schwenk
Advisor:	Dr.-Ing. Christoph Bader
	M.Sc. Christian Mainka
	Dipl.-Ing. Vladislav Mladenov

Ruhr-University Bochum, Germany

Chair for Network and Data Security
Prof. Dr. Jörg Schwenk
Homepage: `www.nds.rub.de`

Erklärung

Acknowledgements

Writing this thesis was a time consuming process and the end result has benefited greatly from the input of a lot of different people whom I would like to thank on this page. First of all I would like to thank my parents for enabling me to be in the position to write this thesis in the first place. Secondly I would like to thank my three advisors who have been directly involved in writing this thesis: There is Dr.-Ing. Christoph Bader that initially set me up with this particular interesting topic. And then there is M.Sc. Christian Mainka and Dipl.-Ing. Vladislav Mladenov whom I have to thank especially for their valuable suggestions and remarks, for both the presentations as well as the actual thesis. I also have to thank both of them for setting up and maintaining the virtual machine that was used for the experiments during this thesis. Additionally I would like to thank everyone that I have not mentioned here but is still somehow involved in my bachelors thesis, this includes, but is not limited to my professors and their assistants, the people at the registrar's office and pretty much everyone else that is involved in keeping the university running.

Now that the obvious *stakeholders* have been pleased (wink), let me come to a few more personal mentions. I would like to thank Peter Skiba for taking the time and interest to proofread the manuscript of this thesis and correcting many post-midnight mistakes, as well as making some stylistic suggestions.

A whole circle of people that also deserves my recognition is my study group. That consists of Alexander Wichert, Christoph Zallmann, Endres Puschner, Johanna Jupke and Tim Guenther. Not just for the occasional LaTeX induced crisis intervention, but also for the good times (and tasty meals) we had during the basic study period. Feel free to visit `https://lerngruppe-id.de` for a visual representation of each of them.

Actually I wanted to thank my laptop for living just long enough for me to finish this thesis. But since it unexpectedly lost power once again while writing this acknowledgement I wont - there you have it you piece of machinery, you are getting replaced by a ThinkPad soon enough. In fact lets thank the internet instead for providing a secure backup of my work.

Almost last but not least, if you are still reading this, then I would like to thank you - the reader - for taking the time to even read the acknowledgement page of this thesis, where someone you probably do not know thanks a bunch of people and even things that you probably also do not know. But by now you have probably realized that this page is to be taken with a wink in one's eye.

And finally I would like to thank Lena Brühl, who always has the last word in our relationship, so why not have it here too? ;-) May the next sixty years be as happy and successful as the the past six ones.

Abstract

CryptDB is a MySQL proxy that allows SQL aware encryption inside existing database management systems. To offer the best possible protecting while enabling the greatest computational flexibility it relies on a new concept called *onions*, where different layers of encryption are wrapped around each other and are only revealed as necessary. While its concept to improve database security looks fresh and interesting from an academic standpoint we wanted to examine the usability in practical application to determine if a real world productive use is desirable. We have therefore benchmarked the performance of CryptDB and examined how well existing applications can be adapted for the use with a CryptDB setup.

KEYWORDS: CryptDB, Databases, Encryption, Onion Encryption, Usability, Benchmark

Contents

List of Figures

List of Tables

Acronyms

AES	Advanced Encryption Standard
BLOB	Binary Large Object
CBC	Cipher Block Chaining
CMC	CBC-mask-CBC
CMS	Content Managing System
CPU	Central Processing Unit
DBA	Database Administrator
DBaaS	Database as a Service
DBMS	Database Management System
IND-CPA	Indistinguishability under chosen-plaintext attack
IV	Initialization Vector
OLTP	Online Transaction Processing
SPEC	Standard Performance Evaluation Corporation
TPC	Transaction Processing Performance Council
UDF	User Defined Function

1. Introduction

In today's computing environment companies accumulate more and more personal data. Virtually every internet user is registered in at least one database, but usually a lot more. In fact, the information provider `Experian` has conducted a survey that revealed that the average Briton between 25 and 34 years has 40.1 different online accounts [2]. The circumstance that there is a huge amount of highly personal data stored in one place makes these databases a very attractive target for both inside and outside attackers. While it seems to be common practice to try to defend the database against attacks from the outside via DMZs, firewalls and intrusion prevention systems it seems like there is not much that can be done against an inside attacker. An inside attacker in our scenario is someone who has limited or full access to the database and its entries, e.g. there is at least one database administrator, who by nature, has to be able to have full control over all access rights to maintain the database. Until now it seems that the only thing you can do would be to trust him to do his job properly[3] [4]. We see something similar with the increasing trend towards cloud computing and Infrastructure as a Service (IaaS) which means that the company which rightfully possesses the user data might not store the data in their facilities but on some third party site. This third party also needs a certain level of control to administrate and maintain their infrastructure. So how can a company outsource data in a possibly untrusted environment without giving away sensible information about their customers?
One approach is to outsource the data only in an encrypted form and let the database perform its operations only on these encrypted data. This is exactly what the software `CryptDB` claims to provide. CryptDB was developed by the MIT and serves as a proxy, a translator, between the application that communicates with standard SQL and the database that behaves like a regular database. According to the original paper by *Popa et al.* [1] that was published together with the first version of `CryptDB`, both the application and the database require only small changes and should otherwise work transparently, i.e. they are unaware that they are computing with encrypted data. But is this really true?

In this thesis we want to evaluate the actual usability of a `CryptDB` setup in practical application. To see whether the loss in performance and the increased space usage are small enough to justify the use of such a crypto layer. Keeping in mind that big companies often maintain databases with several million entries, so even a small overhead might lead to significant differences in the overall result. Additionally we want to conduct whether adapting existing applications is really as easy as it is claimed to be and if there are noteworthy problems that might need to be addressed before a widespread use.

2. Foundation

In this section we describe the general concepts behind databases and SQL that are necessary to understand this thesis. We will see that a minimal knowledge of both is essential to understand how `CryptDB` works.

2.1. Databases and Database Systems

A database is "[a] structured set of data held in computer storage and typically accessed or manipulated by means of specialized software"[5]. Databases are among the most important aspects of the third industrial revolution, that is the transition from analogous to digital computing that took place somewhere between the 1950s and the 1970s. They allow for an abstracted view on data so that the user can request a subset of the underlying data that is relevant to his current interests without him having to rearrange or care about the data itself. Previously digital data sets had typically been stored in "blocks" (e.g. textfiles) that had to follow a precise structure and could only be understood by applications that were specifically aware of the design of said structure. The new ability to store all the data in one place and access only the relevant portions of it has lead to a widespread adoption of database systems. While digital databases have been used since the midst of the twentieth century to store customer data - among other things - it was the triumphant advance of the internet that lead to even more databases and even more customer data that is now distributed around the world. More important than the number of installations is the fact that almost all of these databases are directly connected to the internet and thus are potentially exposed to the threats of cybercrime. This exposure however is necessary to allow the users to access their information from their own internet connected devices from all over the world and still have their personal settings presented to them: E.g. the webshop remembers who you are and what you like and of course, for convenience sake your credit card number, so that your next purchase is just one click away. This development seems logical from an economy point of view. But it is not without risks. As the past has shown millions of sensitive data sets are leaked every year [6]. This includes voluntarily entered data (e.g. a webshop or social media site) [7] as well as administrative data held by the government or employers [8][9].

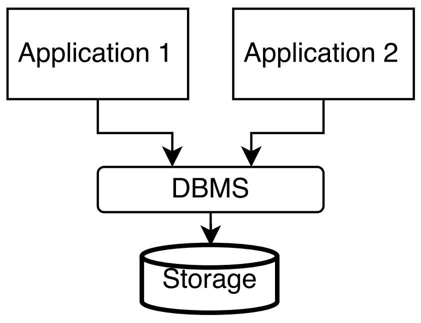


Figure 2.1.: Database storage concept

2.2. Database Management System (DBMS)

The software that coordinates access to the database is called a Database Management System (DBMS). It provides an interface to the user that is usually independent of the platform for both input and output. The input of commands and output of results is usually done via the language SQL (see Sect: 2.3). The DBMS then takes care of the details, such as how it stores the data sets on the filesystem of the used platform or how it handles concurrent access by several applications (see Fig. 2.1).

2.2.1. Structure of a DBMS

In this section we give a quick overview over the structure of a DBMS as well as introducing a few terms that we will use throughout this document. First it has to be stated that a DBMS usually can and will hold more than one database. In fact often the DBMS itself reserves one or more databases to store information about itself and make them accessible to the user, e.g. in `MySQL` one can usually find the databases *mysql* and *information_schema*. Inside such a database there can be several *tables*, that look similar to a spreadsheet. A Database Administrator (DBA) sets *columns* that are supposed to hold a certain data structure (e.g. an integer, a date, a text, ...) and meet specific criteria (e.g. is not allowed to be *NULL*[1]). Afterwards these *columns* can be filled with data. Every successful *INSERT* statement (see Listing 2.4) creates a *new row* in the database table. See Figure 2.2 for a visual representation of these structures inside a DBMS. The single element that you get by combining a *database*, a *table*, a column and a row reference is called a field (e.g. Database 1, Table 3, Column 1, Row 5 would return the field labeled “Value 5”).

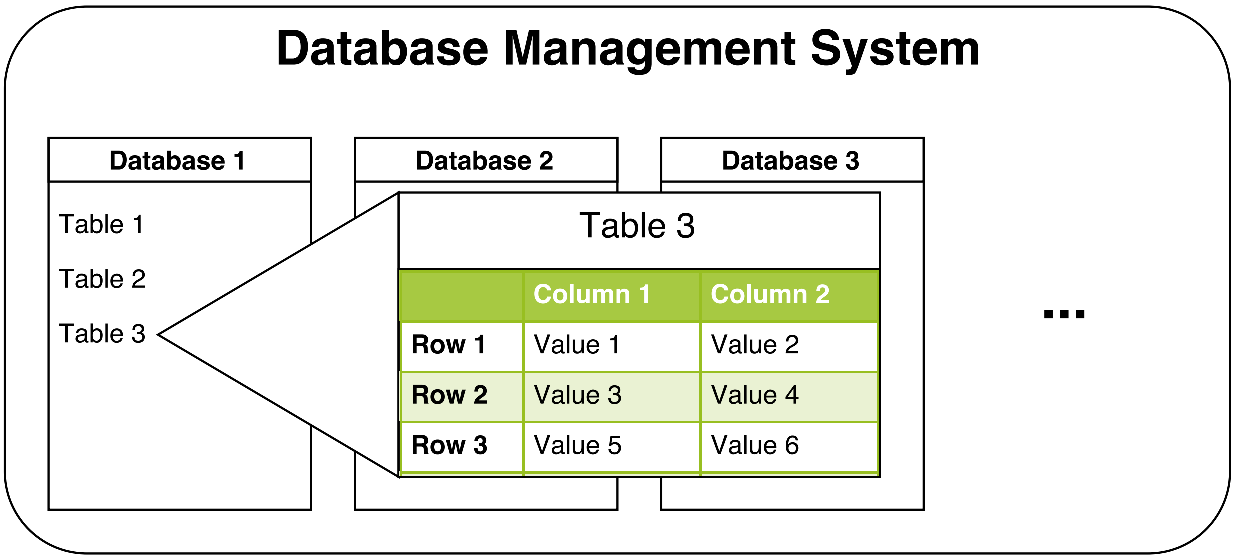


Figure 2.2.: Structure of a Database Management System

[1]SQL distinguishes between an empty string or an integer that is set to 0 and a true *NULL* value. While the first two examples actually have a value, the latter one says that this data field has not been set (yet). There are cases in which it makes sense to prohibit a NULL value.

2.3. The Query Language SQL

Probably the most important language in the brief history of digital databases is SQL which can be classified as a structured descriptive query language. “Structured” because every query has to follow a certain structure, which we will see later. “Descriptive” because the users are intended to enter what they demand (e.g. SUM(COLUMN)) without entering in what way this should be effected (e.g. ROW1 + ROW2 + ROW3 + ROW4). How exactly that sum is calculated is up to the database. SQL has been standardized as ISO/IEC 9075 in 1987 and has been revised several times since then [10]. It lays the foundation for many different database systems, but most of them implement their own additional commands to distinguish themselves from their competitors, thus introducing a certain degree of incompatibility between each other. A few examples of these database systems are `MS SQL`, `Oracle`, `MySQL`, and `PostgreSQL`. The first and second one are examples of commercial database systems which have a high distribution in enterprise environments while the latter two are examples of open source database systems [11]. Due to its free nature `MySQL` has been the standard database system in many webserver packages and features the highest overall installation count (counting company and private users). Because of these two aspects we decided to also use `MySQL` as a base for our tests.

2.3.1. Types of SQL Statements

According to the ISO 9075:2011 definition there are 9 types of SQL Statements:

- SQL-schema statements
- SQL-data statements
- SQL-transaction statements
- SQL-control statements
- SQL-connection statements
- SQL-session statements
- SQL-diagnostics statements
- SQL-dynamic statements
- SQL embedded exception declaration

It is worth mentioning that not all of these statements read from or write to the database. Most of these statements are of an organizational nature and only last temporarily until the session is terminated. The two important classes of statements that can permanently alter the data in the database are the *SQL-schema* and the *SQL-data* statements. This distinction is relevant because - when it comes to concealing sensitive data - only the aforementioned statements do carry a sensitive payload.

2.3.2. Structure of SQL Statements

In this section we briefly explain the structure of the most commonly used *SQL-data* statements that might carry user related data. We will see them again when we discuss `CryptDB`. Generally all SQL statements follow a similar structure: At the beginning it is specified what type of statement should be issued (e.g.

select), depending on the statement there is a further specification (e.g. in case of select: which field should be selected?) and then there is a target (usually a table) which can be restricted by defining certain conditions.

SELECT statement

Probably the most used statement is *SELECT*. It is used to fetch data from the database. One specifies the fields one is interest in and optionally restricts the returned results by certain criteria.

```
<select statement: single row> ::=
SELECT [ <set quantifier> ] <select list>
INTO <select target list>
<table expression>
<select target list> ::=
<target specification> [ { <comma> <target specification> }... ]
```

Figure 2.3.: Formal syntax of a SELECT statement according to ISO/IEC 9075:2011

INSERT statement

The *INSERT* statement is used to insert new data sets into the database.

```
<insert statement> ::=
        INSERT INTO <insertion target> <insert columns and source>

<insertion target> ::=
        <table name>
[...]
```

Figure 2.4.: Formal syntax of an INSERT statement according to ISO/IEC 9075:2011, some optional parts have been omitted due to size restriction and improved readability

UPDATE statement

The *UPDATE* statement is used to modify existing data sets.

```
<update statement: searched> ::=
        UPDATE <target table>
                [ FOR PORTION OF <application time period name>
                        FROM <point in time 1> TO <point in time 2> ]
                [ [ AS ] <correlation name> ]
                SET <set clause list>
                [ WHERE <search condition> ]
```

Figure 2.5.: Formal syntax of a UPDATE statement according to ISO/IEC 9075:2011

DELETE statement

The *DELETE* statement is used to delete a row.

```
<delete statement: searched> ::=
        DELETE FROM <target table>
                [ FOR PORTION OF <application time period name>
                        FROM <point in time 1> TO <point in time 2> ]
                [ [ AS ] <correlation name> ]
                [ WHERE <search condition> ]
```

Figure 2.6.: Formal syntax of a DELETE statement according to ISO/IEC 9075:2011

With the knowledge we have accumulated about the design principles and core mechanics of databases and DBMSs we can now move forward and take a look on how `CryptDB` uses these concepts to interact with the DBMS.

3. CryptDB

In this chapter we take a look at the concepts behind `CryptDB`, similarly to how we looked at databases and SQL in the first chapter. We start with the general setup and then go into details and explain the different encryption methods that are used and what the so called "Onion Layers" are.

3.1. General Setup

`CryptDB` is intended to work as a proxy between the application and the database. An application for example might be a website, an application on a mobile device (a so called "App") or a classic desktop application, basically anything that connects to a database.

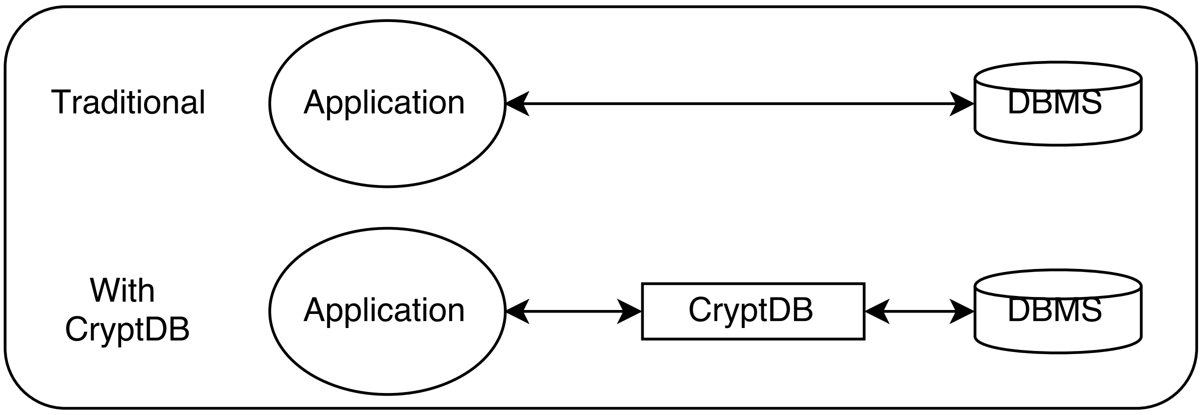


Figure 3.1.: Communication scheme of an application without and with `CryptDB`

3.2. Onion Layers

When it comes to SQL aware encryption there are different aspects of computation that are based on different fundamental principles. For example the operator `GROUP BY` relies on equality checks concerning the encrypted data, other functions like `SUM` rely on the ability to perform additions of the encrypted data. `CryptDB` deals with these different computational aspects by clustering functions by their underlying operations, as mentioned above. Around these different aspects or clusters `CryptDB` builds a construct that the developers have called *onion*: An onion features different layers of encryption from least revealing on the outside to most revealing on the inside (see Fig. 3.2). At the same time the outmost layer is the one with the least functionality while the innermost one offers the greatest functionality. The transformation from one layer into another ("peeling off a layer") happens automatically when the need arises (i.e. when a query with a certain operator/function is issued). In this case `CryptDB` automatically reencrypts the entire column and remembers its state. While technically it is possibly to reencrypt everything to a higher layer of security again it is not recommended by the developers in case of common queries as it would demand a considerable amount of computationpower, besides that the information might have already been revealed.

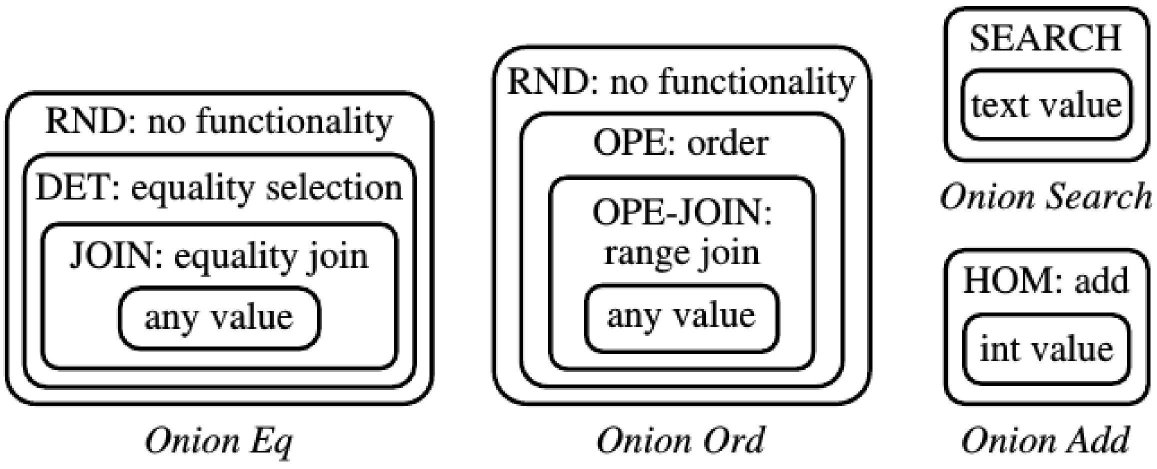


Figure 3.2.: Schematics of the onions construct with various layers that is used in `CryptDB` [1]

3.3. Encryption Types

Each type uses a different algorithm that meets the specified requirements for a certain type and can be exchanged for another algorithm should the need arise, e.g. when a used cipher is broken. In such an event existing encrypted data would have to be decrypted with the old algorithm and reencrypted using the new one. We have listed the different layers from most to least secure. Whereas least secure means that this particular layer does reveal the most information about its encrypted content, please notice that this is sometimes unavoidable in order to perform certain operations and is not automatically insecure.

3.3.1. Random (RND)

The *RND* onion layer provides the strongest security assurances: It is probabilistic, meaning that the same plaintext will be encrypted to a different ciphertext. On the other hand it does not allow any feasible computation in a reasonable amount of time. If someone wants to know something about the content of these fields the encrypted data has to be retrieved as a whole to be decrypted by `CryptDB`. This type seems to be reasonable choice for highly confidential data like medical diagnosis, private messages or credit card numbers that do not need to be compared to other entries for equality.

The current implementation of RND uses the Advanced Encryption Standard (AES) to encrypt strings and Blowfish to encrypt integers[1]. In their paper Popa et al. explain this with the respective block sizes of the two ciphers: Blowfish has a blocksize of 64 bit, which should be large enough to store 99% of all integers, whereas AES is used with a blocksize of 128 bit [1]. This means using Blowfish to store integers only needs half the space that AES needs. Both implementations use the Cipher Block Chaining (CBC) mode with a random Initialization Vector (IV) and are considered to be Indistinguishability under chosen-plaintext attack (IND-CPA) secure by Popa et al. [1].

3.3.2. Homomorphic encryption (HOM)

The *HOM* onion layer provides an equally strong security assurance, as it is considered to be IND-CPA secure too [1]. It is specifically designed for columns of the data type *integer* and allows the database to perform operations of an additive nature. This includes of course the addition of several entries, but also operations like `SUM` or `AVG`. The reason that only addition is supported lies in the fact that fully homomorphic calculations, while mathematically proofen by M. Cooney [12], is unfeasible slow on current hardware. An exception is the homomorphic addition $HOM(x) \cdot HOM(y) = HOM(x + y) \bmod n$, that can be performed in a reasonable amount of time. In `CryptDB` the developers choose to implement the homomorphic addition using the Paillier cryptosystem [13]. Currently the ciphertext of a single integer is stored a `VARBINARY(256)`, this means it uses 256 bytes of space which is 64 times the size of a normal integer that would only use 4 bytes. Considering that integers are among the most used data types in a database this is a huge overhead. Popa et al. indicate that there might be a more efficient way to store the integers with the use of a scheme developed by Ge and Zdonik [14][1]. As of today this has not been implemented.

3.3.3. Word search (SEARCH)

The *SEARCH* onion layer is exclusive for columns of the data type *text*. In the version of `CryptDB` that we used in this thesis (see Appendix A.1) we have been unable to successfully create such an onion. The following explanation is therefore solely of a theoretical nature and based on the paper provided by Popa et al. [1]. This layer uses a modified Version of a cryptographic scheme presented by Song et al.[15] and allows for a keyword level text search with the `LIKE` operator. The implementation splits the string that is to be stored in the database by a specified delimiter (e.g. space or semicolon) and stores each distinct substring in a concatenated and encrypted form in the database. Each substring is padded to a certain size and its position inside the concatenated string is permutated thus obfuscating the position where it appears in the original string. When the user wants to perform a search using the `LIKE` operator `CryptDB` applies the padding to the search term and sends the encrypted version to the DBMS. The DBMS can now search for this specific string and is able to return the results.
This scheme comes with several restrictions: Due to the used scheme it is only able to search for the existence of full words, it does not work with regular expressions or wildcards since they would not be encrypted in the same way.

[1]See line 408 in main/CryptoHandlers.cc

3.3.4. Deterministic (DET)

The *DET* onion layer provides the second strongest security assurance: In contrary to *RND* this layer is deterministic, meaning that the same plaintext will be encrypted to the same ciphertext. This means that the DBMS can identify fields with equal (encrypted) content. This allows us to use functions like `GROUP BY`, to group identical fields together or use `DISTINCT` to only select fields that are different. It does not however reveal whether a certain field is bigger or smaller than another field.
For this type the developers used Blowfish and AES again, although this time they do not distinguish between integers and strings, but choose the cipher depending of the blocksize of the plaintext. Blowfish is used for any plaintext that is smaller than 64 bit and AES for any plaintext that is bigger than 64 bit. In both cases the plaintext is padded up to fit the blocksize. A special situation occurs when the plaintext is longer than the standard 128 bit AES blocksize: In this case the plaintext is split into several blocks which are processed in a variant of AES-CBC-mask-CBC (CMC) mode that uses a zero IV . Popa et al. justifies these special steps because AES in normal CBC mode would reveal prefix equality for the first n blocks in case the first n 128 bit blocks are identical [1].

3.3.5. Order-preserving encryption (OPE)

The *OPE* onion layer is significantly weaker than the *DET* layer as it reveals the order of the different entries. This means that the DBMS knows relations like bigger and smaller, but also equality (without having to look at the *Eq* onion). This means that if $x < y$, then $OPE(x) < OPE(y)$, also if $x = y$, then $OPE(x) = OPE(y)$. This allows us to use ordered operations like `MIN`, `MAX` or `ORDER BY`. To achieve this functionality the developers of `CryptDB` implemented an algorithm that was published by Boldyreva et al. and was inspired by the ideas suggested by Agrawal et al. [16] [17].
In regards to security it is noteworthy that this onion layer is the most revealing one: It can not fulfill the security definition of IND-CPA, as is shown by Boldyreva et al. [16]. Even more important it reveals not just the order but also the proximity of the transformed values to an attacker [16] . This behavior might be acceptable for some values (e.g. text), but might be an issue for others (e.g. financial data).

3.3.6. Join (JOIN, OPE-JOIN)

The *JOIN* and *OPE-JOIN* layers are both “sub layers” of *DET* respective of *OPE*. That means both of them feature the computational abilities of their “parent layer” (i.e. to distinguish whether a plaintext a is equal to plaintext b, respective knowing the order of the entries of a column). In addition to that this type works over multiple columns and allows to determine whether a plaintext in column a is equal to a plaintext in column b for *JOIN* and whether a plaintext in column a is bigger or smaller than a plaintext in column b for *OPE-JOIN*. Both operators work with multiple column allowing for constructs like: *SELECT * FROM test_table WHERE name1=name2 AND name2=name3*.
In this case all 3 name columns will use the same deterministic parameters, with the consequence that the same plaintext will be encrypted in the same ciphertext across all three columns. Therefore it is more

revealing than *DET* or *OPE* alone.

3.4. Related Work

We would like to split the related work part into three different perspectives:

1. CryptDB related papers,
2. CryptDB security related papers and
3. Papers related to similar cryptographic systems

3.4.1. CryptDB related papers

First we would like to mention the original `CryptDB` paper "CryptDB: Protecting Confidentiality with Encrypted Query Processing" [18]. In this paper, released in 2011, `CryptDB` was officially introduced to the public. Besides explaining the elemental concepts of the onion and its layers this paper also features a small section dedicated to performance measurements. However much has changed since then: The multi principal mode (using different keys for different users) has been abandoned and is currently only implemented in a one principal mode. Also there was a general restructuring of the underlying code to improve the overall performance. In this vein the search onion that was used to make encrypted text searchable has not been reimplemented. These findings and the fact that it is not entirely evident how the results were obtained in the first place justify a revalidation of these measurements.

3.4.2. CryptDB security related papers

Even though security is not an official part of this thesis, security is still an important topic when it comes to usability and whether it is worth the additional coasts. One question we had in the beginning was whether a curious database administrator could still draw conclusions from the encrypted data sets and whether he would able to take advantage of that, either by getting interesting insights or by actually being able to manipulate things in a way that would gain him further access to data. For these questions we would like to feature the following two papers: The first one is "*On the Difficulty of Securing Web Applications using CryptDB*" [19] by Ihsan H. Akin and Berk Sunar and the second one is "*Inference Attacks on Property-Preserving Encrypted Databases*" [20] by Muhammad Naveed, Seny Kamara and Charles V. Wright. The First paper shows that the lack of authentication checks for the stored data enables a malicious database administrator to copy and/or exchange row entries so that he could achieve administration privileges in a web application if he manages to identify the correct table. Another interesting aspect, as the the paper points out that as long as the database administrator is able to interact with the web application and is able to "produce" queries whose changes he can log inside the DBMS he will most likely be able to figure out certain relations (e.g. by creating a user/logging in he will most likely be able to figure out the user table and even his users row). At this point he could then try to exchange/copy existing entries to gain further

privileges, while the encryption is technically not broken the web application might be exploited. It is to notice that this might be less of an issue if the application using the database is not publicly accessible.
On the other hand the second paper describes a direct attack against the encryption by way of trying to determine the plaintext value of a ciphertext. To do that they have used two commonly known attacks (frequency analysis and sorting attacks) and also developed two new attacks (l_p-optimization and cumulative attack). All these attacks focus on values encrypted with the DET or OPE layer - the two most revealing layers `CryptDB` has to offer. The proposed attacks have been shown to be very effective when used with *dense* data that is on a limited range (e.g. a scoring from 0-100, or other relatively fixed scales) or the frequency of the data is guessable (e.g. access control groups like administrators, moderators and users). Their findings have however spun up a little controversy between the authors of this paper and Ada Popa, the first author of the `CryptDB` paper who claims that they have wrongfully used the DET and OPE layer for non high entropy values (i.e. values that fulfill the above mentioned criteria). The problem here is that quite a lot operations (like =, !=, count, min, max, group by, order by, ...) require the functionality only offered by one of these two layers. So the question, how these low entropy values should - if at all - be encrypted, still remains open.

3.4.3. Similar cryptographic systems

When talking about databases and cryptography one will probably end up finding an article by Michael Cooney "IBM touts encryption innovation; new technology performs calculations on encrypted data without decrypting it"[12] where he introduces a schema for fully homomorphic encryption for databases in a way that DBMS can perform all operations over fully encrypted data sets. The problem here is that performing most of these operations is extremely slow and therefore is not a valid option for real world applications for the time being. This is the reason why `CryptDB`s middle way approach seems so promising. In fact there are several projects out there that are already based on the core concepts of `CryptDB`, one prominent example is Google's *Encrypted BigQuery Client* [21], which can be used for Google's cloud platform BigQuery. Another example is SAP's *SEEED* implementation for the *SAP HANA* Database Management System (an in memory database), where a paper called "Experiences and observations on the industrial implementation of a system to search over outsourced encrypted data." [22] was published last year. Both systems follow a similar approach like `CryptDB` in the way that both of them implement different onions that consist of different layers of encryption.

4. Benchmark

When evaluating the usability of a new concept like `CryptDB` performance and secondary costs are of great interest to potential investors and users. If a new technology is to be widely adopted it has to have a certain advantage over the old system to justify the coats that are related to switching and running the new system. This becomes all the more interesting since the pricing of cloud services (like Database as a Service (DBaaS)), where `CryptDB` looks most promising, are usually directly related to quantities of usage (often Central Processing Unit (CPU) usage or storage capacities). While the costs are rather obvious in the form of increased storage needs (see Section 3.2, a column is usually padded out and encrypted in several onions) and CPU load (for the underlying cryptography) the advantage of *more security* however is somewhat vague. In this chapter we compare how much additional storage needs and CPU usage are produced by using `CryptDB` in contrast to a normal `MySQL` setup.

4.1. Preliminary Considerations

In our measurements we focused on small to mid sized databases with 1000 to 100 Million rows[23]. This should cover most use cases for web applications. Of course there are applications with a need for larger databases but database scaling is an entirely different topic and is outside of the scope of this thesis. As the database grows `CryptDB` will become more and more of a bottleneck since it is not optimized for simultaneously processing large quantities of queries like the major DBMSs are. Further development would be needed on the side of `CryptDB`.

4.2. Benchmarks

There are several enterprise benchmark processes out there that are used in the industry to measure the performance of Online Transaction Processing (OLTP) systems [24]. OLTP means that the results are computed and evaluated in (near) real-time as opposed to some calculations that run over a longer period of time. The near real-time evaluation is an requirement for most web applications that are supposed to directly display the results produced (e.g. articles on a blog, prices in a webshop, amount of logged in users on a forum, ...). Among the most important OLTP benchmarks are the *SPECjEnterprise2010* by Standard Performance Evaluation Corporation (SPEC)[1] and the *TPC-C* and the *TPC-E* by the Transaction Processing Performance Council (TPC)[2]. All of them feature different suites with a variety of tests to benchmark different scenarios

[1]`https://www.spec.org`
[2]`https://www.tpc.org`

on different hardware. They are designed to measure large enterprise deployments on expensive hardware. However both of these benchmark suites are associated with a fee so we decided to opt for an known and accepted open source solution in the form of the application `SysBench`. This test does not award points for different aspects of the machine and is therefore not usable to identify bottlenecks of the DBMS like other benchmarks do. It rather measures the transactions per second which covers our use case, since we directly compare two different software setups on identical hardware.

4.2.1. SysBench

`SysBench` has been developed by Alexey Kopytov who is a former employee of *MySQL AB* (respective *SUN*, respective *Oracle*), the developers of `MySQL` [25]. Since `SysBench` has been used by *MySQL AB* in several publications it seems to be a reasonable choice for our measurements [26].

4.2.2. Benchmark

When performing a OLTP benchmark with `SysBench`, one usually follows the following three stages, which we will explain in further detail, together with the problems we encountered, later in this section:

1. **Preparation**: We create a test table with a defined amount of rows that is filled with randomly generated data. This is the only table that is being accessed by `SysBench`.
2. **Run**: We run a series of sql queries against the DBMS, respective `CryptDB`
3. **Cleanup**: We delete the previously created test table

4.2.2.1. Scenario

The scenarios differ only in the size of the underlying test database (*PARAM1*, controlled with the *–oltp-table-size* switch) and the number of sequential requests we send to said database (*PARAM2*, controlled with the *–max-requests* switch). Please note that one request is used synonymously to one transaction, which is in fact a block of several SQL commands, so even though there are *only* e.g. 1000 requests the number of actual statements - like select or update - is higher. To simulate concurrent database access we ran the benchmark with 1, 2, 4 and 8 threads each (*PARAM3*, controlled with the *–num-threads* switch). Our originally scenarios included the following parameters:

1. **Small database**: table size: 1.000 rows (PARAM1), 1.000 requests (PARAM2)
2. **Medium database**: table size: 1.000.000 rows (PARAM1), 100.000 requests (PARAM2)

However due to problems we encountered during the benchmarking (see below) we had to reduce the size of these parameters to the following values:

1. **Small database**: table size: 100 rows (PARAM1), 1.000 requests (PARAM2)
2. **Medium database**: table size: 100.000 rows (PARAM1), 500 requests (PARAM2)

We have run both scenarios against our normal MySQL installation as well as against a setup with CryptDB serving as proxy in front of the very same MySQL server to directly compare how using CryptDB influenced the performance. We repeated each tests five times to cancel out load fluctuations on the host system. We restarted CryptDB after every benchmark, because we discovered that after a benchmark ran CryptDB did not properly free the allocated memory, eventually leading to a filled up memory. A nearly filled memory slows down all operations and is eventually being killed by the Kernel *Out of Memory Killer* (oom-killer). We have described this problem below in the preparation section (see Sect. 4.2.2.2).

4.2.2.2. Preparation

In this step we tell SysBench to setup a sample database in which we can perform our SQL commands without destroying currently existing databases. The database is being created with the following command:

```
./sysbench --test=tests/db/oltp.lua --oltp-table-size=PARAM1 --mysql-host
   =127.0.0.1 --mysql-port=3307 --mysql-db=test_with_cryptdb --mysql-user=
   root --mysql-password=rootpassword prepare
```

With this command it invokes the following SQL script (see Fig. 4.1) which adds a new table *sbtest1* with four different columns. The *id* column is an integer that is automatically increased, ranging from 1 to the specified table size (see Sect. 4.2.2.1). The other three fields contain pseudo randomly generated data of different lengths, with the field *k* being an integer and *c* as well as *pad* being of the data type *char*. The latter two contain groups of integers separated by -.

```
CREATE TABLE `sbtest1` (
      `id` int(10) unsigned NOT NULL auto_increment,
      `k` int(10) unsigned NOT NULL default '0',
      `c` char(120) NOT NULL default '',
      `pad` char(60) NOT NULL default '',
       PRIMARY KEY (`id`),
       KEY `k` (`k`));
```

Figure 4.1.: The SQL script responsible for creating the test table on which the benchmark is performed

We ran into problems, when we tried to run the prepare stage with CryptDB with a table size greater than 70.000. The problem was that CryptDB continued to allocate memory for every new insert statement, while not freeing it anymore. This built up to a point where all memory was allocated and the kernel had to kill the CryptDB process to continue operating normally. We solved this problem by exporting the unencrypted test table from our MySQL server into a .sql file. We split this file in the middle, so that we had two files with roughly 50.000 insert statements each. Then we first imported the first file into our CryptDB setup, quickly restarted it and imported the second file. Restarting CryptDB frees the allocated memory and does not come with any negative side effects as the values are safely stored inside the DBMS.

4.2.2.3. Run

In this phase we send the actual queries to CryptDB respective MySQL. The selected ruleset - *OLTP* - consists of the following queries, where x and y stand for different randomly generated numbers that change for each query and are always within the specified limits. c_val and pad_val stand for random, but correct pattern for c and pad:

```
SELECT c FROM sbtest1 WHERE id=x
SELECT c FROM sbtest1 WHERE id BETWEEN x AND y
SELECT SUM(K) FROM sbtest1 WHERE id BETWEEN x AND y
SELECT c FROM sbtest1 WHERE id BETWEEN x AND y ORDER BY c
SELECT DISTINCT c FROM sbtest1 WHERE id BETWEEN x AND y ORDER BY c
UPDATE sbtest1 SET k=k+1 WHERE id=x
UPDATE sbtest1 SET c='c_val' WHERE id=y
DELETE FROM sbtest1 WHERE id=x
INSERT INTO sbtest1 (id, k, c, pad) VALUES (x, y, c_val, pad_val)
```

All of these lie withing the scope of commands that CryptDB is able to process. We confirmed this by manual testing each type of query and cross checking the output for an “ERROR 1105 (07000): unhandled sql command 28” that is thrown, when an certain sql command is not supported. Some of these commands require a certain onion layer, so that CryptDB has to strip away outer layers and reencrypt the data to a inner layer. Since this is a one time only process in addition to the possibility to set the correct onion layer from the start via an annotated schema file we decided to only include the measurements for the second run, where all onions have been encrypted to the correct layer. Since we would consider this the normal usage behavior. We eventually ran the test with the following command:

```
./sysbench --test=tests/db/oltp.lua --oltp-table-size=PARAM1 --mysql-host⤸
   =127.0.0.1 --mysql-port=3307 --mysql-db=test_with_cryptdb --mysql-user=⤸
   root --mysql-password=rootpassword --init-rng=off  --max-requests=PARAM2 ⤸
   --num-threads=PARAM3 run
```

When running the original scenario we encountered the same memory leak problem we have already described in Sect. 4.2.2.2. We therefore had to limit the amount of consecutive requests as described in the scenario section (see Sect. 4.2.2.1).

4.2.2.4. Cleanup

This stage basically deletes the previously created test table and is the only step that always worked without a problem. The command issued was:

```
./sysbench --test=tests/db/oltp.lua --mysql-host=127.0.0.1 --mysql-port=3307 ⤸
   --mysql-db=test_with_cryptdb --mysql-user=root --mysql-password=⤸
   rootpassword cleanup
```

4.3. Results

We have split the results in two subsections, the first one showing the results for the insertion and the second one showing the results for the mixed queries. We choose this separation because it approximates the real world scenarios where an existing database is transferred to a `CryptDB` setup: The initial transfer can be scheduled and might be slow, the more relevant part is certainly the day to day usage that is simulated in the *Run Stage*.

4.3.1. Scenario 1: Small Database

PARAM1=100 rows, PARAM2=1.000 requests
The database size is 64KiB for the unencrypted and 128KiB for the encrypted data. This is an increase of 100% in storage requirements for the 100 rows setup.

4.3.1.1. Preparation stage / INSERT

The preparation stage is always conducted with 1 thread (PARAM3=1).

Run	1	2	3	4	5	Mean	Median
Time	0m0.054s	0m0.050s	0m0.049s	0m0.053s	0m0.050s	0m0.051s	0m0.050s

Table 4.1.: MySQL: Scenario=1, PARAM1=100, PARAM2=1.000, PARAM3=1

Run	1	2	3	4	5	Mean	Median
Time	0m2.203s	0m2.148s	0m2.135s	0m2.208s	0m2.145s	0m2.168s	0m2.148s

Table 4.2.: CryptDB: Scenario=1, PARAM1=100, PARAM2=1.000, PARAM3=1

The mean increased by 2.117s (4151%), the median increased by 2.098s (4196%).

4.3.1.2. Run Stage / Mixed Queries

This stage should reflect the everyday usage behavior of a database and thus contains a mix of various different queries (See Sect. 4.2.2.3). The proportions of these mixed queries stayed the same for each scenario.

1 Thread (PARAM3=1)

Run	1	2	3	4	5	Mean	Median
Time	0m5.462s	0m5.710s	0m5.510s	0m5.827s	0m5.418s	0m5.585s	0m5.510s

Table 4.3.: MySQL: Scenario=1, PARAM1=100, PARAM2=1.000, PARAM3=1

Run	1	2	3	4	5	Mean	Median
Time	6m32.172s	6m34.065s	6m37.230s	6m33.954s	6m32.019s	6m33.888s	6m33.954s

Table 4.4.: CryptDB: Scenario=1, PARAM1=100, PARAM2=1.000, PARAM3=1

The mean increased by 6m28.303s (6953%), the median increased by 6m28.444s (7050%).

2 Threads (PARAM3=2)

Run	1	2	3	4	5	Mean	Median
Time	0m4.003s	0m3.996s	0m4.160s	0m4.075s	0m4.023s	0m4.051s	0m4.023s

Table 4.5.: MySQL: Scenario=1, PARAM1=100, PARAM2=1.000, PARAM3=2

Run	1	2	3	4	5	Mean	Median
Time	6m19.873s	6m31.458s	6m41.970s	6m24.641s	6m29.260s	6m29.440s	6m29.260s

Table 4.6.: CryptDB: Scenario=1, PARAM1=100, PARAM2=1.000, PARAM3=2

The mean increased by 6m25.389s (9513%), the median increased by 6m25.237 (9576%).

4 Threads (PARAM3=4)

Run	1	2	3	4	5	Mean	Median
Time	0m3.078s	0m2.986s	0m2.978s	0m2.957s	0m2.950s	0m2.990	0m2.957s

Table 4.7.: MySQL: Scenario=1, PARAM1=100, PARAM2=1.000, PARAM3=4

Run	1	2	3	4	5	Mean	Median
Time	6m26.908s	6m30.280s	5m24.649s	4m58.225s	4m53.582s	5m38.729s	5m24.649s

Table 4.8.: CryptDB: Scenario=1, PARAM1=100, PARAM2=1.000, PARAM3=4

The mean increased by 5m35.779s (11382%), the median increased by 5m21.692 (10879%).
Note: Beginning with 4 threads we saw very unreliable and inconsistent measures for the `CryptDB` setup. Not only did we see a higher distance between outliers and the mean execution time, but in 14 out of 21 tries the benchmark execution was disrupted and did not finish properly. In those cases `CryptDB` is displaying the following error message:

```
ALERT: mysql_drv_query() for query 'UPDATE sbtest1 SET c⤸
   ='27150449877-35882676096-71614727371-78027787201-37878787463-59418460466-
84621839479-87330447182-06793416020-97484136743' WHERE id=46' failed: 4095
```

```
(main/dml_handler.cc, 1266)
DML query failed against remote database
```

The value for *c* and *id* are different for every error message as they are randomly generated, but the SQL operation associated with the failure has been *UPDATE* 13 out of 14 times, the remaining one can be accounted to a *DELETE* statement. This seems to suggest that there is a problem with handling table/row locks as these are usually required by those two operations. This would also explain why we have not experienced this problem with the *SELECT* or *INSERT* statement which can be simultaneously processed by the DBMS. It is not clear, and we have not investigated it further, whether this is actually a problem on the side of `CryptDB` or on the side of `MySQL`. We did not experience this behavior with the `MySQL` benchmark however.

8 Threads (PARAM3=8)

Run	1	2	3	4	5	Mean	Median
Time	0m2.545s	0m2.616s	0m2.590s	0m2.535s	0m2.568s	0m2.571s	0m2.568s

Table 4.9.: MySQL: Scenario=1, PARAM1=100, PARAM2=1.000, PARAM3=8

Run	1	2	3	4	5	Mean	Median
Time	-	-	-	-	-	-	-

Table 4.10.: CryptDB: Scenario=1, PARAM1=100, PARAM2=1.000, PARAM3=8

Note: At this point we had a zero percent success rate with the `CryptDB` setup. We tried 25 times to run the benchmark, but each time it was aborted in under a minute with the error described above (see Sect. 4.3.1.2, 4 Threads (PARAM3=4)).

4.3.2. Scenario 2: Medium Database

PARAM1=100.000 rows, PARAM2=10.000 requests
The database size is 24.1MiB for the unencrypted and 68.6MiB for the encrypted data. This is an increase of 185% in storage requirements for the 100.000 rows setup.

4.3.2.1. Preparation stage / INSERT

The preparation stage is always conducted with 1 thread (PARAM3=1).

Run	1	2	3	4	5	Mean	Median
Time	0m2.548s	0m2.693s	0m2.674s	0m2.579s	0m2.564s	0m2.612s	0m2.579s

Table 4.11.: MySQL: Scenario=2, PARAM1=100.000, PARAM2=500, PARAM3=1

Run	1	2	3	4	5	Mean	Median
Time	39m25.769s	41m47.200s	40m12.541s	40m20.809s	39m10.930s	40m11.450s	40m12.541s

Table 4.12.: CryptDB: Scenario=2, PARAM1=100.000, PARAM2=500, PARAM3=1

The mean increased by 40m8.838s (92222%), the median increased by 40m9.962s (93446%).

4.3.2.2. Run Stage / Mixed Queries

1 Threads (PARAM3=1)

Run	1	2	3	4	5	Mean	Median
Time	0m3.513s	0m3.378s	0m3.248s	0m3.304s	0m3.259s	0m3.340s	0m3.304s

Table 4.13.: MySQL: Scenario=2, PARAM1=100.000, PARAM2=500, PARAM3=1

Run	1	2	3	4	5	Mean	Median
Time	19m32.590s	19m23.299s	31m53.204s	30m31.576s	19m34.881s	24m11.110s	19m34,881s

Table 4.14.: CryptDB: Scenario=2, PARAM1=100.000, PARAM2=500, PARAM3=1

The mean increased by 24m7.770s (43346%), the median increased by 19m31.577s (35459%).

2 Threads (PARAM3=2)

Run	1	2	3	4	5	Mean	Median
Time	0m2.459s	0m2.438s	0m2.599s	0m2.574s	0m2.559s	0m2.526s	0m2.559

Table 4.15.: MySQL: Scenario=2, PARAM1=100.000, PARAM2=500, PARAM3=2

Run	1	2	3	4	5	Mean	Median
Time	19m53.822s	19m58.848s	30m55.864s	30m21.356s	19m57.951s	24m13.568s	19m58.848s

Table 4.16.: CryptDB: Scenario=2, PARAM1=100.000, PARAM2=500, PARAM3=2

The mean increased by 24m11.042s (57444%), the median increased by 19m56.289s (46748%).

4 Threads (PARAM3=4)

Run	1	2	3	4	5	Mean	Median
Time	0m1.992s	0m1.947s	0m1.888s	0m1.877s	0m2.074s	0m1.956s	0m1.947s

Table 4.17.: MySQL: Scenario=2, PARAM1=100.000, PARAM2=500, PARAM3=4

Run	1	2	3	4	5	Mean	Median
Time	19m35.493s	26m32.328s	19m35.664s	19m9.848s	19m23.405s	20m51.348s	19m35.493s

Table 4.18.: CryptDB: Scenario=2, PARAM1=100.000, PARAM2=500, PARAM3=4

The mean increased by 20m49.392s (63875%), the median increased by 19m33.546s (60275%).

8 Threads (PARAM3=8)

Run	1	2	3	4	5	Mean	Median
Time	0m1.677s	0m1.627s	0m1.683s	0m1.767s	0m1.688s	0m1.688s	0m1.683s

Table 4.19.: MySQL: Scenario=2, PARAM1=100.000, PARAM2=500, PARAM3=8

Run	1	2	3	4	5	Mean	Median
Time	19m29.417s	19m21.851s	19m53.931s	19m54.450s	20m19.593s	19m47.745s	19m53.931s

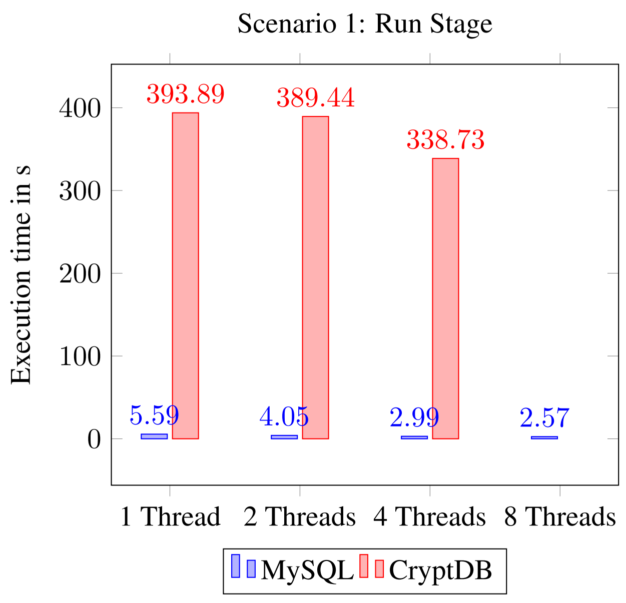
Table 4.20.: CryptDB: Scenario=2, PARAM1=100.000, PARAM2=500, PARAM3=8

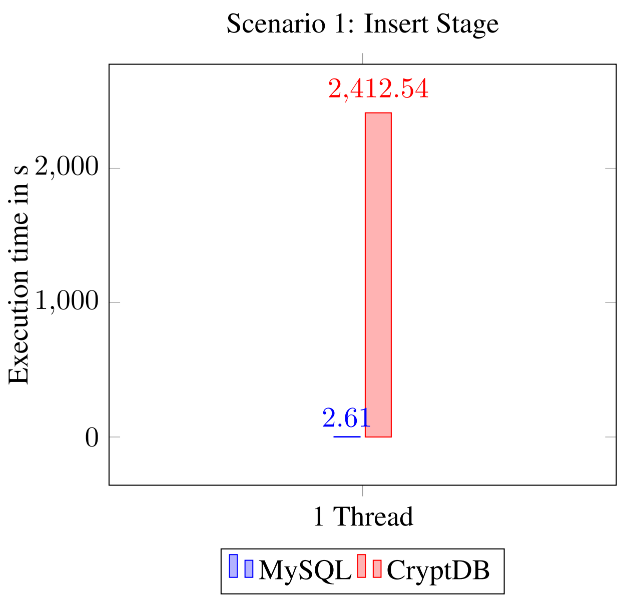
The mean increased by 19m46.057s (70264%), the median increased by 19m52.248s (70841%).

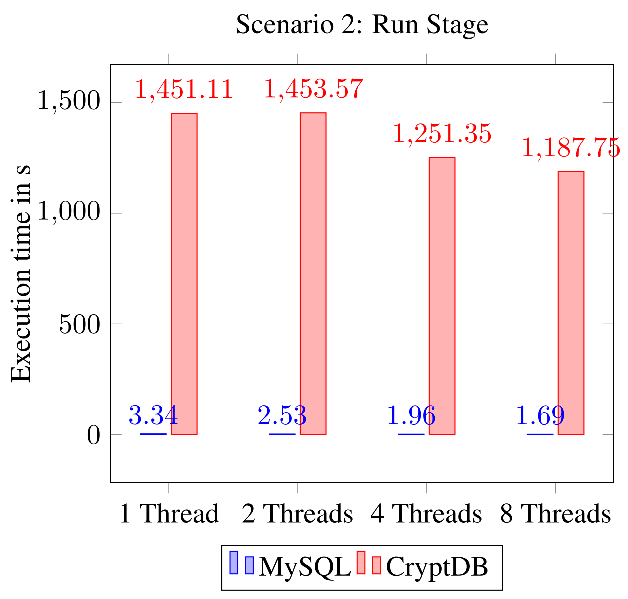
4.4. Conclusion

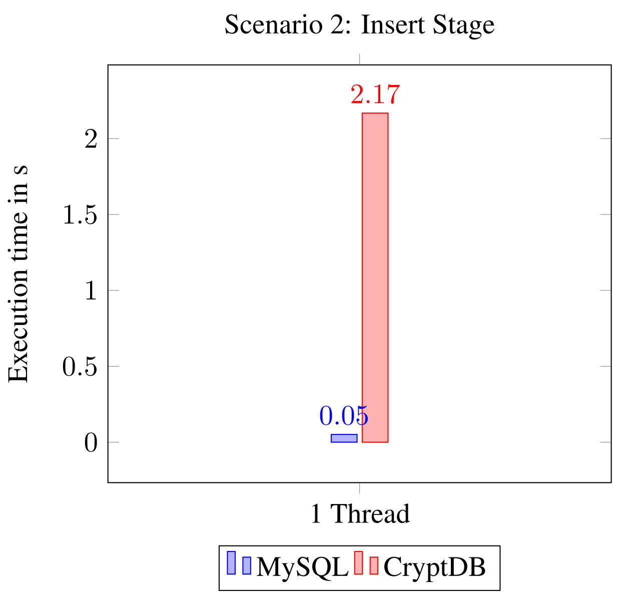
We have ran a multitude of benchmarks with various parameters. We have seen that the `CryptDB` setup is always noticeably slower than the `MySQL` setup. This was fully expected seeing that the `CryptDB` works as a proxy that also performs cryptography and thus has naturally an additional delay compared to a standalone `MySQL` setup. The original `CryptDB` paper [18] thus compared the results of `CryptDB` with SQL queries executed over a MySQL proxy. Since such a proxy would not be the standard setting for small to mid sized databases installations we deliberately did not use one. Our benchmark shows that the total execution time drastically increases with the number of affected rows. Somewhat confusing was that the proportions of the total insert time and the total query execution time are exactly opposite for both scenarios: In the small database scenario inserting is quicker by a small margin, in the medium sized database scenario the percentage increase of the inserting time is more than double of the percentage increase of the query execution time. Another important aspect is the magnitude of the percentage increase: For the small database the increase is of a factor of 10^3 for both times, whereas the increase for the medium sized database is of the factor 10^4. In clearly defined systems that operate in a clearly defined time these two findings would usually indicate some sort of error in the benchmark execution. However since the system we tested does not have fixed but flexible execution times that are influenced by the virtual machine as a whole, we have been looking into other possible causes for these observations. On very likely cause in our opinion could be memory management: During our initial tests we have noticed that the originally planned scenarios are too big to be properly executed by `CryptDB`. `CryptDB` would continuously eat up the available memory and was eventually killed by the kernel *Out-of-memory-killer*. Such a crash also occurred when we first tried to insert 100.000 rows into `CryptDB`, we therefore split the table creation script in two part roughly containing 50.000 rows each. We then loaded the first table creation script, restarted `CryptDB` and loaded the second table creation script, only taking time of the actual script execution. While `CryptDB` worked fine with this routine (in the sense that it did not crash) it might have been slowed down by the still increasing memory load that was aggregated during that process. But even with having a somewhat reasonable explanation for these numbers the fact remained that the current version of `CryptDB` seemed severely flawed and proved unreliable in both memory management and thread handling. Thread handling was only problematic for the first scenario, where we started to experience failing queries with 4 threads and came to a completely unexecutable benchmark with 8 threads. The second scenario did not suffer from these problems, what suggests that the problem might be related with the small amount of rows that might have overlapped each other during queries and a missing or insufficient locking mechanism thereof. Regarding the storage size we have been positively surprised to find that the needs have only doubled in the first scenario and not just tripled in the second scenario. Both seems pretty reasonable seeing that every column has several (usually two - for numbers three) onions. All in all the benchmark revealed the true state `CryptDB` is currently in: An abandoned development state of a research prototype with many unresolved issues, not ready for productive usage.

4.4.1. Time Charts

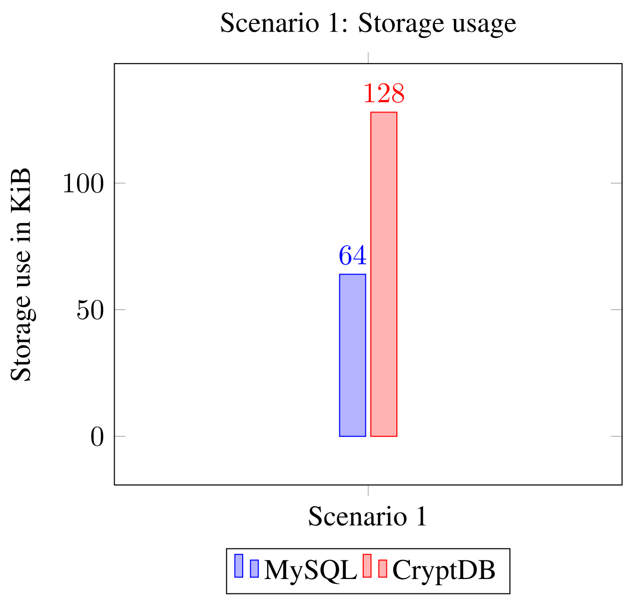


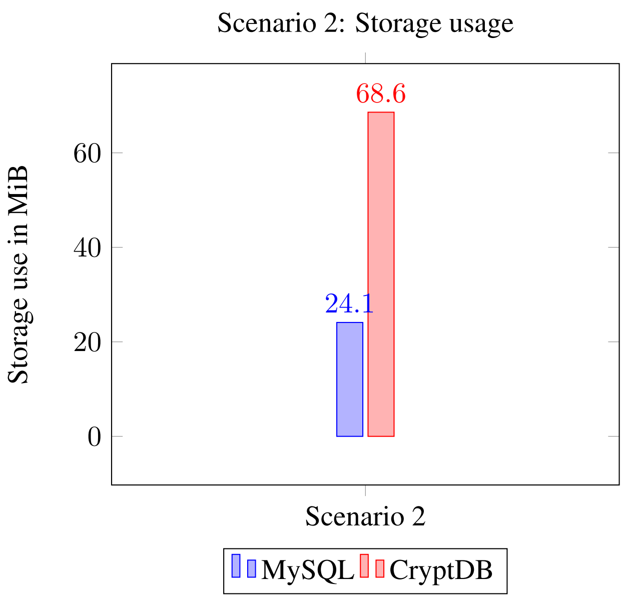






4.4.2. Storage Charts





5. Adapting Applications

After we have measured the performance of a `CryptDB` setup in the previous chapter we now want to test how difficult it is to adjust an existing application to make it usable with `CryptDB`.

5.1. Preliminary Considerations and Restrictions

First off we would like to restrict the scenario to the capabilities of the latest publicly available version of `CryptDB`. The two major points affected by this are:

1. Principals: In the current version of `CryptDB` there is only support for one single principal
2. Onions: In the current version of `CryptDB` there is no search onion and thus no ability to perform a keyword search over encrypted data

These two features were present in an earlier version of the `CryptDB` prototype but have not been reimplemented in the latest version, which features a different structure and was supposedly optimized in other areas (see Appendix A.1 for version numbers). While the single principal restriction does not affect our test cases, the lack of search onions limits our abilities to perform searches over encrypted data. As a result we are left with two choices:

1. We do not care about existing search functions in the application and accept that they will not work properly or
2. We do not encrypt the affected fields (i.e. those that are searchable) at all, leaving existing search functions intact and allow the DBMS to know our search terms as well as all searchable texts.

For our sample applications that do not have any specific security requirements this choice is negligible and we have therefore opted for option two and went with a best effort approach where the strict security setting failed. In fact, recognizing that these problems and options exist is exactly the purpose of these tests. To enable us to have unencrypted fields in the database we have to explicitly set the (undocumented) environmental variable *$SECURE_CRYPTDB* to the value *false* before starting the `CryptDB` proxy. Where incompatible datatypes would have produced an error before they will now be used unencrypted.

5.2. Techniques

One of the challenges when trying to use an application with `CryptDB` is knowing beforehand if all the queries used will be supported or whether the application will run into troubles when in use. We therefore tried to extract all SQL queries from the sourcecode of the scripts to manually check their compliance. There are two problems with that:

1. The function executing the query (e.g. mysql_db_query(), mysqli_query(), pdo->exec(), ...) is the most reliable way to detect a SQL query, but often contains only a variable, not the query itself.
2. Queries are often constructed on a dynamic basis, often with user input. This means that the final query cannot be fully evaluated beforehand. This problem intensifies when dealing with script languages like *PHP* which are not type secure, where even the nature of a variable can be unclear until it is processed (e.g. is it an integer or a string?).

To extract the SQL sequences we slightly adapted an idea published in [27] and used a regular expression (see Listing 5.1) together with `grep`[1] to try and find static parts of an SQL query, namely the operators `SELECT`, `UPDATE`, `INSERT`, `DELETE`, `DROP`, `SHOW`.

```
grep -i -r -n "sql =.*\"\(SELECT\|UPDATE\|INSERT\|DROP\|DELETE\|SHOW\)" . | ⤸
    awk -F: '{print "filename: "$1"\nline: "$2"\nmatch: "$3"\n\n"}'
```

Listing 5.1: Grep command to find sql statements, prettified by awk. Please notice that the backslashes are used to escape characters in bash and are not part of the actual RegEx

The regular expression searches case insensitive for a php variable *$sql* that is somewhere in the same line followed by either of the aforementioned statements. Please notice that it is just a common habit to name the SQL string *$sql*, but in fact it can have any valid variable name whatsoever and needs to be adapted for different applications. We piped the output of the commands from Listing 5.1 into a file for further analysis. Here we could use additional grep commands to explicitly search for known problematic queries.

5.2.1. Problematic Queries

As already mentioned some queries are of a problematic nature and cannot be supported by `CryptDB` due to the underlying cryptography. These problematic queries can be divided into two subclasses:

1. Operator related queries
2. Logic related queries

An example for the first class is the `EXPLAIN` operator, which is not (properly) supported by `CryptDB`. The first class of queries is easy to filter for by a list of keywords. The second class is actually not that easy to filter for, it covers the cases where computation and comparison happens in the same sql query (e.g. `SELECT a FROM b WHERE c = 2*d`). Unfortunately we have not been able to come up with a proper way to detect the latter case since there is a huge variety in the ways such a query might be constructed. We discovered some throughout our experiments with setting up the applications but there are surely more problematic queries out there that we did not catch.

[1] `http://www.gnu.org/software/grep/`

5.3. Sample Applications

For our sample applications we choose web applications that are widely used and targeted at small to mid sized databases. We usually installed the application to the normal `MySQL` installation first, to gather all the relevant SQL data and to avoid running out of (php) script execution time. Because - as we have seen in the benchmark section - inserting larger data structures at the same instant can easily consume some time. Then we exported the whole database from `MySQL` and imported it through our `CryptDB` setup. Afterwards we adapted the application to access `CryptDB` directly.

5.3.1. Wordpress 4.3

Wordpress is arguably the most used blogging software of today, with its website claiming up to 24% of all websites running Wordpress[2].

Installation

We downloaded the latest version from the wordpress homepage[3] and extracted its content in the document root of our `apache2` server. When accessing the site the installer script tries to generate a file `wp-config.php`, if it is not already present. Unfortunately the installer does not allow to set a specific port for our `MySQL` installation, which is needed to access the `CryptDB` proxy. Therefore we had to create the `wp-config.php` file ourself by adapting the provided sample file. After doing that we accessed the installer script again which tried to create the necessary tables. However this script failed to create all the necessary tables and was aborted. To narrow down the issue we repeated the installation with `MySQL` and tried to export it out of `MySQL` and reimport it with `CryptDB`. While doing so we counted 169 sql queries. However 27 out of these queries failed and resulted in an error at the `CryptDB` console. A closer inspection revealed that 21 of these errors are caused by a failing `INSERT` statement because the corresponding table does not exist. The remaining 6 failing queries concerned the creation of said tables. Therefore they induced missing tables in the further process. This leaves us with only 5 out of originally 11 tables. A close comparison of the queries that worked and did not work (See Sect. B.1.1.1 for the full queries) revealed that the only difference was that the queries that did not work featured a length restriction in their KEY assignment (see Listing 5.2, second to last line). After we manually deleted these restrictions by turning e.g. `KEY ‘meta_key‘ (‘meta_key‘(191))` in `KEY ‘meta_key‘ (‘meta_key‘)` only, all eleven tables have been created successfully. This eliminated the reason for the other failing queries. The problematic queries can be identified with the following regular expression “`KEY .*(\d*).*)`” (excluding the parenthesis).

```
CREATE TABLE IF NOT EXISTS ‘wp_usermeta‘ (
  ‘umeta_id‘ bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  ‘user_id‘ bigint(20) unsigned NOT NULL DEFAULT '0',
  ‘meta_key‘ varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  ‘meta_value‘ longtext COLLATE utf8mb4_unicode_ci,
  PRIMARY KEY (‘umeta_id‘),
```

[2]`https://de.wordpress.com`
[3]`https://de.wordpress.org/latest-de_DE.zip`

```
  KEY `user_id` (`user_id`),
  KEY `meta_key` (`meta_key`(191))
) ENGINE=InnoDB  DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci AUTO_INCREMENT=16 ;
```

Listing 5.2: CREATE TABLE query that produces an error

Usage

With the now fixed tables we can fully load the `Wordpress` start page, including the example entry that came with its installation. When first visiting the site we were still logged in as administrator due to a valid authentication cookie from the installation procedure. The first thing we tested was the search function, which worked fine. That means we have been able to produce positive search results (article found) for keywords that are present and we have been able to find empty search results (no article) for keywords that were not present. Also we were able to fully navigate the dashboard (the administration area of `Wordpress`). However, we were not able to create new blog entries or new users. Also when logged out we were unable to log in again. The common denominator of these unsuccessful actions were sql commands that have been replaced by `DO 0` instructions (see Sect. 5.3.2, where we observed the same behavior in regard to `Piwik`). In the case of `Wordpress` we have been able to track it down to a single[4] line of code that is invoking the sql query *SHOW FULL COLUMNS FROM $table* in *wp-includes/wp-db.php:2306*. This line is part of a function that tries to determine the character set that is used by the columns. For testing purposes we simply returned the (known) character set immediately upon calling the function by inserting a `return 'utf8';` in line 2280. This solved the problems mentioned above and allowed us to log in again, create new users, new blog entries and comments. We have not conducted an in-depth test of all `Wordpress` features, which would be outside of the time scope of this thesis, but the basic blogging features appear to work. There might even be a more sophisticated solution by issuing a query against the information_schema table. Something along the lines like Listing 5.3, the variables *$this->dbname* and *$table* are known to the script already, however the information_schema table only knows about the encrypted table names, so one would have to find a way to work around that. One option would be to simply pick the first entry of the database and rely on all tables in this database using the same character set (in which case one would drop the `AND T.table_name` clause and replace it with `LIMIT 1`).

```
SELECT CCSA.character_set_name FROM information_schema.`TABLES` T, ⤸
    information_schema.`COLLATION_CHARACTER_SET_APPLICABILITY` CCSA WHERE CCSA⤸
    .collation_name = T.table_collation AND T.table_schema = "'.$this->dbname⤸
    .'" AND T.table_name = "'.$table.'";
```

Listing 5.3: An idea for a patch to determine the character set of any table

[4]Actually there is a second occurrence in *wp-admin/includes/upgrade.php*. This is the upgrade API and not vital for running the script. Therefore we did not look into that file, but we believe it can be fixed in a similar fashion

5.3.2. Piwik

Piwik is an open source web analytic tool that according to its website has been downloaded over 2.5 million times[5].

Installation

When trying to install we were confronted with two failing queries. The first one was caused by a KEY parameter using prefix lengths (see Sect. 5.3.1), the key is of the Binary Large Object (BLOB) type, where it makes arguably sense to use only a prefix instead of the whole entry as a key. We did not come up with a better way and deleted the whole key instead which allowed us to proceed with the installation but could possibly cause some unwanted side effects later on. The second problem arose when the installer tried to create the *piwik_log_visit* table (see Appendix B.5 for the full query), a table with 64 columns which through `CryptDB` would have been expanded to a table with about 260 columns. Through trial and error we figured out that 234 columns seem to be the maximum number of columns that can be created at the same time. We also checked whether the actual length of the query had any influence by artificially extending the column names. As we could execute queries with more than 3.000 characters successfully, while other queries with less than 3.000 characters (e.g. 2.880) failed, we found out, that the limiting factor was in fact the number of columns. By omitting 8 columns of the original query, with 2.880 characters we were able to make the query work. What is interesting is that we have been able to alter the table directly afterwards to insert these columns by a simple `ALTER TABLE piwik_log_visit ADD [...]` statement. With these two problems more or less solved we have been able to finish the installation.

To allow access to `CryptDB` we added a variable *port* and set it to `port = "3307"` in the file `config/config.ini.php`. It is worth noting that `Piwik` supports two database drivers: Mysqli and PDO, as we will see in the usage section this makes somewhat of a difference. The driver in use can be changed in this file as well.

Usage

When we first opened the site using the *Mysqli* driver we received a message saying "Error: Piwik is already installed". When looking at the `CryptDB` console we see that not all the SQL commands are properly executed. In fact only the first statement is executed correctly (as seen by the corresponding NEW QUERY line). We have not been able to determine why `CryptDB` registers the additional empty queries:

```
QUERY: SET NAMES utf8
NEW QUERY: SET NAMES utf8
==============================================
QUERY: SELECT DATABASE()
unexpected packet type 22
==============================================
QUERY:
unexpected packet type 23
==============================================
QUERY:
```

[5] https://piwik.org

```
unexpected packet type 26
================================================
QUERY:
unexpected packet type 25
================================================
QUERY: SELECT option_value, option_name FROM `piwik_option` WHERE autoload = 1
unexpected packet type 22
================================================
QUERY:
```

What is interesting though is that when using the *PDO* driver the very same commands yield a proper response (output truncated for a better readability):

```
QUERY: SET NAMES 'utf8'
NEW QUERY: SET NAMES 'utf8'
================================================
QUERY: SELECT DATABASE()
NEW QUERY: select database() AS `DATABASE()`
ENCRYPTED RESULTS:
[...]
================================================
QUERY: SELECT option_value, option_name FROM `piwik_option` WHERE autoload = 1
NEW QUERY: select `nocrypt_piwik`.`table_TWWWPXWVZJ`.`PMZXFVWXOQoEq`,`nocrypt_piwik`.`table_TWWWPXWVZJ`.`cdb_saltENOGSDAEQE`,`
      nocrypt_piwik`.`table_TWWWPXWVZJ`.`BIADPUMKCJoEq` from `nocrypt_piwik`.`table_TWWWPXWVZJ` where (`nocrypt_piwik`.`
      table_TWWWPXWVZJ`.`VIAEURAOYGoEq` = 552574328611617537)
ENCRYPTED RESULTS:
[...]
```

As a result there are a few more queries executed, however we still cannot use the page properly as we get another error message (although this time the sites theme is loaded already): "SQLSTATE[HY000]: General error". This error is thrown by the PDO driver and can be traced back to the way `CryptDB` responds to certain commands. Instead of executing commands like `EXPLAIN`, `DESCRIBE` or `SHOW COLUMNS` properly, what would reveal certain properties about the database structure, `CryptDB` rewrites them to a `DO 0` command, that simply returns an empty row, whereas PDO would expects a non empty response as an adequate result. It is outside of the scope of this thesis to analyze the source code of all the applications we tested in order to figure out why it relies on these commands and come up with a possible workaround as these could only be symptomatic and require a different approach each time one wants to preserve the secrecy that `CryptDB` seems to require. We found a workaround for `Wordpress`, where only the character set of a column was required. Piwik however tries to retrieve all columns of a table which seems not to be possible from our current perspective, as the only workaround (considering the application) would be to query the *information_schema* table, which only contains encrypted column names. There might be a possibility to utilize the undocumented `SET @cryptdb='show'` statement that displays the layer status of the different onions and in the same vein reveals which tables contain which columns.The easier solution however (as in one fix fixes all) would be to simply allow these commands in `CryptDB`. However this might cause security implications that would have to be inspected first.

```
QUERY: SHOW COLUMNS FROM . piwik_log_visit
NEW QUERY: DO 0;
```

Listing 5.4: CryptDB replacing a SHOW COLUMNS statement with a DO 0

5.3.3. Joomla

Joomla is another popular open source Content Managing System (CMS). According to a 2011 survey by *w3techs* it is second only to Wordpress[6].

Installation

As before we went ahead and installed the application in our MySQL database and exported the created tables. With the regular expression presented in Sect. 5.3.1 we have been able to directly identify some critical prefix restrained keys that we stripped of their prefix restriction. The second issue we discovered after trying to import the script file was that CryptDB only supports InnoDB as storage engine which conflicted with two tables that should have been created using the MEMORY storage engine. We changed both of them so that they would use the InnoDB storage engine instead of the MEMORY storage engine by replacing the engine name in the last line of Listing 5.5.

```
CREATE TABLE IF NOT EXISTS `fdnag_finder_tokens` (
  `term` varchar(75) NOT NULL,
  KEY `idx_word` (`term`),
) ENGINE=MEMORY DEFAULT CHARSET=utf8;
```

Listing 5.5: (Truncated) table creation script that features the MEMORY storage engine

After these adjustments we have been able to import these tables without further problems.

Usage

When opening the site with the default database driver *mysqli* we are faced with two error messages (please see Appendix B.3, due to their size). Both error messages display a fairly complex failed query, including several CASE WHEN ... THEN ... constructs. When looking in the CryptDB console we have been able to find the corresponding log entry for those two failed queries. We see a behavior that is similar to the one described in the Piwik section, where we found the failing queries followed by a message about unexpected packets:

```
[...]
THEN a.state ELSE 0 END = 1 AND (a.publish_up = '0000-00-00␣00:00:00' OR a.⤸
   publish_up <= '2015-09-16␣13:47:06') AND (a.publish_down = '0000-00-00␣⤸
   00:00:00' OR a.publish_down >= '2015-09-16␣13:47:06')
================================================
QUERY:
unexpected packet type 14
```

Listing 5.6: Joomla: Unexpected package types in CryptDB console (output truncated)

[6]http://w3techs.com/technologies/overview/content_management/all

To verify that this error is not caused by `CASE WHEN ... THEN ...` constructs we performed a few different SELECT statements using this construct. All of them worked fine. So we tried to further narrow the problem down by removing parts from the failing query step by step. It turned out that the second query would eventually work, when we removed the `LEFT OUTER JOIN` and its succeeding subquery. However both the `LEFT OUTER JOIN` and the subquery functioned properly when testing them alone. So the exact cause of this error remains unknown to us. And unfortunately this resulted in an unusable application.

5.4. Conclusion

We have tested some of the most prominent open source web applications to see how much effort it takes to get them to run in a stable manner. Out of all the applications we have tested, none were able to run out of the box. With a few tweaks we were able to get one application to run in a stable manner, whereas the others can not be considered stable or just would not run at all. The reasons for the problems we faced with each applications were quite diverse and ranged from driver related problems to cryptography related timeouts and downright unsupported commands. Another issue, that we have chosen to ignore widely, is the fact that with the current version of `CryptDB` it has not been possible to individually select the sensitivity of each column. Instead we discovered and used an undocumented environmental variable named *$SECURE_CRYPTDB* to switch `CryptDB` from the "secure everything and abort if we can not do that" approach to a "try to secure everything as good as we can and if we can not do that then leave them unencrypted" approach. While this was convenient for us and necessary to test the existing applications, it raises the question whether the additional overhead is worth enduring, when some of the most sensitive fields, i.e. the text fields are not encrypted at all.

6. Conclusions

We know that database security is a difficult topic that becomes increasingly difficult with outsourcing data in the cloud. Regardless of that there is more and more data aggregated every day with increasing sensitive character. And not a single weak passes without some major database leak. So is `CryptDB` the solution we have all been waiting for?
With our tests we have shown that the current version of `CryptDB` suffers from severe memory problems, rendering it totally useless at times where it crashed during a reencryption of an onion layer, leaving us with a unaccessible data. But even when the data sets have been small enough to not cause a memory leak we have seen that accessing them is multiple times slower with `CryptDB`. Though to be fair this is not only overhead resulting from the cryptography but also due to the indirect access with a proxy. So the difference in access times is smaller if your regular `MySQL` setup utilizes a proxy as well. As for the storage requirement we were surprised to see it only increase about 200%, which seems quite acceptable when we consider our scenarios of a small to mid sized database with only a few million rows. Extrapolating our measures from scenario two, this would translate to a database size of below 10 GiB for a database with 10.000.000 rows. This should be quite affordable with today's storage prices. Of course these views can not be applied to large corporate databases which store data of another magnitude and require features like distributed access and load balancing. These features are currently unavailable in `CryptDB` but could certainly be implemented when the different `CryptDB` instances synchronize their internal state after certain operations (e.g. new master key, change of onion layer state, ...). As for security, which was not part of this thesis, but is still relevant when evaluating the usability of a system, we have seen that `CryptDB` is not the final answer to all the problems related to database security. In fact there are still conceptual questions like how one will ever be able to disclose the order of data to the server without revealing to much information. It is however a feasible and justifiable first step in the right direction, leading to more secure databases. Not with `CryptDB` itself, which in its current version is lacking existential features like encrypted text and as of now is nothing more than a research prototype, but rather with the development to follow. We currently see that at least two major database developers are incorporating the core ideas of `CryptDB` in their own enterprise database systems and more research is being done in this area.

A. Appendix General

A.1. Testing Environment

In this section I want to describe the setup we used to run our tests on, as well as explain which software was involved. This is meant as a reference to be able to recreate a similar environment. For any thoughts and conclusions as result of the tests that ran on this setup please refer to the according section in the main thesis.
Due to practical reasons our test system was virtualized with QEMU/Bochs. The node running the test system had a reserved (i.e. fixed) amount of CPU and RAM capacity that is solely available to this system. Network traffic however is shared with other virtual machines.

OS and Hardware

- Operating System: Ubuntu 14.01 LTS (GNU/Linux 3.13.0-57-generic x86_64)
- QEMU Virtual CPU version 2.0.0 (2299.998 MHz)
- 12305888 kB Memory (Ĩ2GB)

Software

Here you see a list of the software we used, along with its version. Please notice that linux distributions occasionally modify their packages to include bug fixes or make adjustments specific to the distribution. Therefore we also included the exact package version in braces.

- Apache2: 2.4.7 (2.4.7-1ubuntu4.4)
- Bison: 2.7.12-4996 (2.7.1.dfsg-1)
- CryptDB (commit c7c7c7748f060011af9e4cf5158ccfc52ae891f6 (Date: Feb 19 00:45:26 2014 -0500))
- MySQL: 14.14 (5.5.44-0ubuntu0.14.04.1)
- UnixBench 5.1.3
- Sysbench 0.5 (commit 0257f50738a9ff5f4ef4ee0ade12b6d902a6e88c (Date Jul 8 08:52:17 2015 +0300))

A.2. Installing CryptDB

Before installing `CryptDB` we installed the software that we intended to run `CryptDB` with first. We did this because the readme file for `CryptDB` hinted that it would install some User Defined Functions (UDFs) into an existing `MySQL` installation. Therefore we installed `Apache2` first, followed by `PHP` and `MySQL`. Afterwards we started with the actual `CryptDB` installation (see Listing A.1 for a copyable version): The current version of `CryptDB` features a script that installs all necessary dependencies when run with the

appropriate privileges (i.e. root), therefore it is only necessary to install two components to start with: The first one is `git`, to download the source code and the second one is `ruby` to run the installation script itself. We install both by issuing the command *sudo aptitude install git ruby*. Then we use the freshly installed `git` to "clone" (i.e. download (if necessary) and copy to a new (local) destination) the CryptDB source code, the `-b public` switch tells git to fetch the files from the "public" branch [28]. Now we switch inside the newly downloaded folder *cryptdb* with the *cd* command. In here we started the installation script with elevated privileges with *sudo ./scripts/install.rb*. At the end of the installation we are told to set an environmental variable called *$EDBDIR*, to point to the full path of `CryptDB`. We do so by adding the following line to our `.bashrc` file, as recommended by the installer, in order to automatically set this variable every time we log into the system: *export EDBDIR=/home/mskiba/git/cryptdb/cryptdb*.
Note: During the compilation of some `MySQL` related files the installation script was aborted with the following error message: "*error: 'yythd' was not declared in this scope*". After some research on the internet we were able to associate that problem with our `bison` installation. Apparently the error message was the result of some incompatibilities between version 2 and version 3. Since we had the latter one installed we tried to downgrade our installed version of bison to version 2. However the installation script kept updating this version 2 to the most recent available version 3. Therefore we modified the installation script, by removing bison from the list of software to install/update and additionally to that we locked it in the distributions package manager with *aptitude hold bison libbison-dev* to prevent the system to update this package. After these changes the compilation and installation went through without further problems.

Listing A.1: Steps to install the current version of CryptDB

```
sudo aptitude install git ruby
git clone -b public git://g.csail.mit.edu/cryptdb
cd cryptdb
sudo ./scripts/install.rb .
```

B. Appendix Logs and Errors

This appendix chapter should serve as a place to reference errors or logs that would require to much space in their respective chapter and are therefore separated here. Please refer to the corresponding text if you have any questions, as this section is not meant to explain anything.

B.1. Adapting Applications

B.1.1. Sample Applications

B.1.1.1. Wordpress 4.3

Listing B.1: Working CREATE TABLE queries

```
CREATE TABLE IF NOT EXISTS `wp_links` (
  `link_id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  `link_url` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  `link_name` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  `link_image` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  `link_target` varchar(25) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  `link_description` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  `link_visible` varchar(20) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT 'Y',
  `link_owner` bigint(20) unsigned NOT NULL DEFAULT '1',
  `link_rating` int(11) NOT NULL DEFAULT '0',
  `link_updated` datetime NOT NULL DEFAULT '0000-00-00 00:00:00',
  `link_rel` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  `link_notes` mediumtext COLLATE utf8mb4_unicode_ci NOT NULL,
  `link_rss` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  PRIMARY KEY (`link_id`),
  KEY `link_visible` (`link_visible`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci AUTO_INCREMENT=1 ;

CREATE TABLE IF NOT EXISTS `wp_options` (
  `option_id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  `option_name` varchar(64) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  `option_value` longtext COLLATE utf8mb4_unicode_ci NOT NULL,
  `autoload` varchar(20) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT 'yes',
  PRIMARY KEY (`option_id`),
  UNIQUE KEY `option_name` (`option_name`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci AUTO_INCREMENT=127 ;

CREATE TABLE IF NOT EXISTS `wp_term_relationships` (
  `object_id` bigint(20) unsigned NOT NULL DEFAULT '0',
  `term_taxonomy_id` bigint(20) unsigned NOT NULL DEFAULT '0',
  `term_order` int(11) NOT NULL DEFAULT '0',
  PRIMARY KEY (`object_id`,`term_taxonomy_id`),
  KEY `term_taxonomy_id` (`term_taxonomy_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;

CREATE TABLE IF NOT EXISTS `wp_term_taxonomy` (
  `term_taxonomy_id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  `term_id` bigint(20) unsigned NOT NULL DEFAULT '0',
  `taxonomy` varchar(32) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  `description` longtext COLLATE utf8mb4_unicode_ci NOT NULL,
  `parent` bigint(20) unsigned NOT NULL DEFAULT '0',
  `count` bigint(20) NOT NULL DEFAULT '0',
  PRIMARY KEY (`term_taxonomy_id`),
  UNIQUE KEY `term_id_taxonomy` (`term_id`,`taxonomy`),
  KEY `taxonomy` (`taxonomy`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci AUTO_INCREMENT=2 ;

CREATE TABLE IF NOT EXISTS `wp_users` (
  `ID` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  `user_login` varchar(60) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  `user_pass` varchar(64) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  `user_nicename` varchar(50) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  `user_email` varchar(100) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  `user_url` varchar(100) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  `user_registered` datetime NOT NULL DEFAULT '0000-00-00 00:00:00',
  `user_activation_key` varchar(60) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  `user_status` int(11) NOT NULL DEFAULT '0',
  `display_name` varchar(250) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  PRIMARY KEY (`ID`),
  KEY `user_login_key` (`user_login`),
  KEY `user_nicename` (`user_nicename`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci AUTO_INCREMENT=2 ;
```

Listing B.2: Not working CREATE TABLE queries

```
CREATE TABLE IF NOT EXISTS `wp_commentmeta` (
  `meta_id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  `comment_id` bigint(20) unsigned NOT NULL DEFAULT '0',
  `meta_key` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `meta_value` longtext COLLATE utf8mb4_unicode_ci,
  PRIMARY KEY (`meta_id`),
  KEY `comment_id` (`comment_id`),
  KEY `meta_key` (`meta_key`(191))
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci AUTO_INCREMENT=1 ;

CREATE TABLE IF NOT EXISTS `wp_comments` (
  `comment_ID` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  `comment_post_ID` bigint(20) unsigned NOT NULL DEFAULT '0',
  `comment_author` tinytext COLLATE utf8mb4_unicode_ci NOT NULL,
  `comment_author_email` varchar(100) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  `comment_author_url` varchar(200) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  `comment_author_IP` varchar(100) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  `comment_date` datetime NOT NULL DEFAULT '0000-00-00␣00:00:00',
  `comment_date_gmt` datetime NOT NULL DEFAULT '0000-00-00␣00:00:00',
  `comment_content` text COLLATE utf8mb4_unicode_ci NOT NULL,
  `comment_karma` int(11) NOT NULL DEFAULT '0',
  `comment_approved` varchar(20) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '1',
  `comment_agent` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  `comment_type` varchar(20) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  `comment_parent` bigint(20) unsigned NOT NULL DEFAULT '0',
  `user_id` bigint(20) unsigned NOT NULL DEFAULT '0',
  PRIMARY KEY (`comment_ID`),
  KEY `comment_post_ID` (`comment_post_ID`),
  KEY `comment_approved_date_gmt` (`comment_approved`,`comment_date_gmt`),
  KEY `comment_date_gmt` (`comment_date_gmt`),
  KEY `comment_parent` (`comment_parent`),
  KEY `comment_author_email` (`comment_author_email`(10))
) ENGINE=InnoDB  DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci AUTO_INCREMENT=2 ;

CREATE TABLE IF NOT EXISTS `wp_postmeta` (
  `meta_id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  `post_id` bigint(20) unsigned NOT NULL DEFAULT '0',
  `meta_key` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `meta_value` longtext COLLATE utf8mb4_unicode_ci,
  PRIMARY KEY (`meta_id`),
  KEY `post_id` (`post_id`),
  KEY `meta_key` (`meta_key`(191))
) ENGINE=InnoDB  DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci AUTO_INCREMENT=2 ;

CREATE TABLE IF NOT EXISTS `wp_posts` (
  `ID` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  `post_author` bigint(20) unsigned NOT NULL DEFAULT '0',
  `post_date` datetime NOT NULL DEFAULT '0000-00-00␣00:00:00',
  `post_date_gmt` datetime NOT NULL DEFAULT '0000-00-00␣00:00:00',
  `post_content` longtext COLLATE utf8mb4_unicode_ci NOT NULL,
  `post_title` text COLLATE utf8mb4_unicode_ci NOT NULL,
  `post_excerpt` text COLLATE utf8mb4_unicode_ci NOT NULL,
  `post_status` varchar(20) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT 'publish',
  `comment_status` varchar(20) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT 'open',
  `ping_status` varchar(20) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT 'open',
  `post_password` varchar(20) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  `post_name` varchar(200) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  `to_ping` text COLLATE utf8mb4_unicode_ci NOT NULL,
  `pinged` text COLLATE utf8mb4_unicode_ci NOT NULL,
  `post_modified` datetime NOT NULL DEFAULT '0000-00-00␣00:00:00',
  `post_modified_gmt` datetime NOT NULL DEFAULT '0000-00-00␣00:00:00',
  `post_content_filtered` longtext COLLATE utf8mb4_unicode_ci NOT NULL,
  `post_parent` bigint(20) unsigned NOT NULL DEFAULT '0',
  `guid` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  `menu_order` int(11) NOT NULL DEFAULT '0',
  `post_type` varchar(20) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT 'post',
  `post_mime_type` varchar(100) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  `comment_count` bigint(20) NOT NULL DEFAULT '0',
  PRIMARY KEY (`ID`),
  KEY `post_name` (`post_name`(191)),
  KEY `type_status_date` (`post_type`,`post_status`,`post_date`,`ID`),
  KEY `post_parent` (`post_parent`),
  KEY `post_author` (`post_author`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci AUTO_INCREMENT=4 ;

CREATE TABLE IF NOT EXISTS `wp_terms` (
  `term_id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(200) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  `slug` varchar(200) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
  `term_group` bigint(10) NOT NULL DEFAULT '0',
  PRIMARY KEY (`term_id`),
  KEY `slug` (`slug`(191)),
  KEY `name` (`name`(191))
) ENGINE=InnoDB  DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci AUTO_INCREMENT=2 ;

CREATE TABLE IF NOT EXISTS `wp_usermeta` (
  `umeta_id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  `user_id` bigint(20) unsigned NOT NULL DEFAULT ’0’,
  `meta_key` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `meta_value` longtext COLLATE utf8mb4_unicode_ci,
  PRIMARY KEY (`umeta_id`),
  KEY `user_id` (`user_id`),
  KEY `meta_key` (`meta_key`(191))
) ENGINE=InnoDB  DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci AUTO_INCREMENT=16 ;
```

B.1.1.2. Joomla

Listing B.3: Error displayed after first opening the page

```
Error: Database discrepancry! FILE: main/dml_handler.cc LINE: 729 SQL=SELECT a.id, a.title, a.alias, a.introtext, a.fulltext, a.
      checked_out, a.checked_out_time, a.catid, a.created, a.created_by, a.created_by_alias, CASE WHEN a.modified = ’0000-00-00 
      00:00:00’ THEN a.created ELSE a.modified END as modified, a.modified_by, uam.name as modified_by_name,CASE WHEN a.publish_up
       = ’0000-00-00 00:00:00’ THEN a.created ELSE a.publish_up END as publish_up,a.publish_down, a.images, a.urls, a.attribs, a.
      metadata, a.metakey, a.metadesc, a.access, a.hits, a.xreference, a.featured, a.language, LENGTH(a.fulltext) AS readmore,CASE
       WHEN badcats.id is not null THEN 0 ELSE a.state END AS state,c.title AS category_title, c.path AS category_route, c.access 
      AS category_access, c.alias AS category_alias,CASE WHEN a.created_by_alias > ’ ’ THEN a.created_by_alias ELSE ua.name END AS
       author,ua.email AS author_email,parent.title as parent_title, parent.id as parent_id, parent.path as parent_route, parent.
      alias as parent_alias,ROUND(v.rating_sum / v.rating_count, 0) AS rating, v.rating_count as rating_count,c.published, CASE 
      WHEN badcats.id is null THEN c.published ELSE 0 END AS parents_published FROM fdnag_content AS a LEFT JOIN fdnag_categories 
      AS c ON c.id = a.catid LEFT JOIN fdnag_users AS ua ON ua.id = a.created_by LEFT JOIN fdnag_users AS uam ON uam.id = a.
      modified_by LEFT JOIN fdnag_categories as parent ON parent.id = c.parent_id LEFT JOIN fdnag_content_rating AS v ON a.id = v.
      content_id LEFT OUTER JOIN (SELECT cat.id as id FROM fdnag_categories AS cat JOIN fdnag_categories AS parent ON cat.lft 
      BETWEEN parent.lft AND parent.rgt WHERE parent.extension = ’com_content’ AND parent.published != 1 GROUP BY cat.id ) AS 
      badcats ON badcats.id = c.id INNER JOIN fdnag_content_frontpage AS fp ON fp.content_id = a.id WHERE a.access IN (1,1,5) AND 
      c.access IN (1,1,5) AND CASE WHEN badcats.id is null THEN a.state ELSE 0 END = 1 AND (a.publish_up = ’0000-00-00 00:00:00’ 
      OR a.publish_up <= ’2015-09-16 13:47:06’) AND (a.publish_down = ’0000-00-00 00:00:00’ OR a.publish_down >= ’2015-09-16 
      13:47:06’) ORDER BY c.lft, a.featured DESC, fp.ordering, CASE WHEN a.publish_up = ’0000-00-00 00:00:00’ THEN a.created ELSE 
      a.publish_up END DESC , a.created DESC LIMIT 0, 4
Error: Database discrepancry! FILE: main/dml_handler.cc LINE: 729 SQL=SELECT COUNT(*) FROM fdnag_content AS a LEFT JOIN 
      fdnag_categories AS c ON c.id = a.catid LEFT JOIN fdnag_users AS ua ON ua.id = a.created_by LEFT JOIN fdnag_users AS uam ON 
      uam.id = a.modified_by LEFT JOIN fdnag_categories as parent ON parent.id = c.parent_id LEFT JOIN fdnag_content_rating AS v 
      ON a.id = v.content_id LEFT OUTER JOIN (SELECT cat.id as id FROM fdnag_categories AS cat JOIN fdnag_categories AS parent ON 
      cat.lft BETWEEN parent.lft AND parent.rgt WHERE parent.extension = ’com_content’ AND parent.published != 1 GROUP BY cat.id )
       AS badcats ON badcats.id = c.id INNER JOIN fdnag_content_frontpage AS fp ON fp.content_id = a.id WHERE a.access IN (1,1,5) 
      AND c.access IN (1,1,5) AND CASE WHEN badcats.id is null THEN a.state ELSE 0 END = 1 AND (a.publish_up = ’0000-00-00 
      00:00:00’ OR a.publish_up <= ’2015-09-16 13:47:06’) AND (a.publish_down = ’0000-00-00 00:00:00’ OR a.publish_down >= 
      ’2015-09-16 13:47:06’)
```

Listing B.4: Working CREATE TABLE queries

```
CREATE TABLE IF NOT EXISTS `piwik_log_visit` (
  `idvisit` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `idsite` int(10) unsigned NOT NULL,
  `idvisitor` binary(8) NOT NULL,
  `visit_last_action_time` datetime NOT NULL,
  `config_id` binary(8) NOT NULL,
  `location_ip` varbinary(16) NOT NULL,
  `location_longitude` float(10,6) DEFAULT NULL,
  `location_latitude` float(10,6) DEFAULT NULL,
  `location_region` char(2) DEFAULT NULL,
  `visitor_localtime` time NOT NULL,
  `location_country` char(3) NOT NULL,
  `location_city` varchar(255) DEFAULT NULL,
  `config_device_type` tinyint(100) DEFAULT NULL,
  `config_device_model` varchar(100) DEFAULT NULL,
  `config_os` char(3) NOT NULL,
  `config_os_version` varchar(100) DEFAULT NULL,
  `visit_total_events` smallint(5) unsigned NOT NULL,
  `visitor_days_since_last` smallint(5) unsigned NOT NULL,
  `config_quicktime` tinyint(1) NOT NULL,
  `config_pdf` tinyint(1) NOT NULL,
  `config_realplayer` tinyint(1) NOT NULL,
  `config_silverlight` tinyint(1) NOT NULL,
  `config_windowsmedia` tinyint(1) NOT NULL,
  `config_java` tinyint(1) NOT NULL,
  `config_gears` tinyint(1) NOT NULL,
  `config_resolution` varchar(9) NOT NULL,
  `config_cookie` tinyint(1) NOT NULL,
  `config_director` tinyint(1) NOT NULL,
  `config_flash` tinyint(1) NOT NULL,
  `config_device_brand` varchar(100) DEFAULT NULL,
  `config_browser_version` varchar(20) NOT NULL,
  `visitor_returning` tinyint(1) NOT NULL,
  `visitor_days_since_order` smallint(5) unsigned NOT NULL,
  `visitor_count_visits` smallint(5) unsigned NOT NULL,
  `visit_entry_idaction_name` int(11) unsigned NOT NULL,
  `visit_entry_idaction_url` int(11) unsigned NOT NULL,
  `visit_first_action_time` datetime NOT NULL,
  `visitor_days_since_first` smallint(5) unsigned NOT NULL,
  `visit_total_time` smallint(5) unsigned NOT NULL,
  `user_id` varchar(200) DEFAULT NULL,
  `visit_goal_buyer` tinyint(1) NOT NULL,
  `visit_goal_converted` tinyint(1) NOT NULL,
  `visit_exit_idaction_name` int(11) unsigned NOT NULL,
  `visit_exit_idaction_url` int(11) unsigned DEFAULT '0',
  `referer_url` text NOT NULL,
  `location_browser_lang` varchar(20) NOT NULL,
  `config_browser_engine` varchar(10) NOT NULL,
  `config_browser_name` varchar(10) NOT NULL,
  `referer_type` tinyint(1) unsigned DEFAULT NULL,
  `referer_name` varchar(70) DEFAULT NULL,
  `visit_total_actions` smallint(5) unsigned NOT NULL,
  `visit_total_searches` smallint(5) unsigned NOT NULL,
  `referer_keyword` varchar(255) DEFAULT NULL,
  `location_provider` varchar(100) DEFAULT NULL,
  `custom_var_k1` varchar(200) DEFAULT NULL,
  `custom_var_v1` varchar(200) DEFAULT NULL,
  PRIMARY KEY (`idvisit`),
  KEY `index_idsite_config_datetime` (`idsite`,`config_id`,`visit_last_action_time`),
  KEY `index_idsite_datetime` (`idsite`,`visit_last_action_time`),
  KEY `index_idsite_idvisitor` (`idsite`,`idvisitor`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=1 ;

ALTER TABLE piwik_log_visit ADD `custom_var_k2` varchar(200) DEFAULT NULL;
ALTER TABLE piwik_log_visit ADD `custom_var_v2` varchar(200) DEFAULT NULL;
ALTER TABLE piwik_log_visit ADD `custom_var_k3` varchar(200) DEFAULT NULL;
ALTER TABLE piwik_log_visit ADD `custom_var_v3` varchar(200) DEFAULT NULL;
ALTER TABLE piwik_log_visit ADD `custom_var_k4` varchar(200) DEFAULT NULL;
ALTER TABLE piwik_log_visit ADD `custom_var_v4` varchar(200) DEFAULT NULL;
ALTER TABLE piwik_log_visit ADD `custom_var_k5` varchar(200) DEFAULT NULL;
ALTER TABLE piwik_log_visit ADD `custom_var_v5` varchar(200) DEFAULT NULL;
```

Listing B.5: Not working CREATE TABLE queries

```
CREATE TABLE IF NOT EXISTS `piwik_log_visit` (
  `idvisit` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `idsite` int(10) unsigned NOT NULL,
  `idvisitor` binary(8) NOT NULL,
  `visit_last_action_time` datetime NOT NULL,
  `config_id` binary(8) NOT NULL,
  `location_ip` varbinary(16) NOT NULL,
  `location_longitude` float(10,6) DEFAULT NULL,
  `location_latitude` float(10,6) DEFAULT NULL,
  `location_region` char(2) DEFAULT NULL,
  `visitor_localtime` time NOT NULL,
  `location_country` char(3) NOT NULL,
  `location_city` varchar(255) DEFAULT NULL,
  `config_device_type` tinyint(100) DEFAULT NULL,
  `config_device_model` varchar(100) DEFAULT NULL,
  `config_os` char(3) NOT NULL,
  `config_os_version` varchar(100) DEFAULT NULL,
  `visit_total_events` smallint(5) unsigned NOT NULL,
  `visitor_days_since_last` smallint(5) unsigned NOT NULL,
  `config_quicktime` tinyint(1) NOT NULL,
  `config_pdf` tinyint(1) NOT NULL,
  `config_realplayer` tinyint(1) NOT NULL,
  `config_silverlight` tinyint(1) NOT NULL,
  `config_windowsmedia` tinyint(1) NOT NULL,
  `config_java` tinyint(1) NOT NULL,
  `config_gears` tinyint(1) NOT NULL,
  `config_resolution` varchar(9) NOT NULL,
  `config_cookie` tinyint(1) NOT NULL,
  `config_director` tinyint(1) NOT NULL,
  `config_flash` tinyint(1) NOT NULL,
  `config_device_brand` varchar(100) DEFAULT NULL,
  `config_browser_version` varchar(20) NOT NULL,
  `visitor_returning` tinyint(1) NOT NULL,
  `visitor_days_since_order` smallint(5) unsigned NOT NULL,
  `visitor_count_visits` smallint(5) unsigned NOT NULL,
  `visit_entry_idaction_name` int(11) unsigned NOT NULL,
  `visit_entry_idaction_url` int(11) unsigned NOT NULL,
  `visit_first_action_time` datetime NOT NULL,
  `visitor_days_since_first` smallint(5) unsigned NOT NULL,
  `visit_total_time` smallint(5) unsigned NOT NULL,
  `user_id` varchar(200) DEFAULT NULL,
  `visit_goal_buyer` tinyint(1) NOT NULL,
  `visit_goal_converted` tinyint(1) NOT NULL,
  `visit_exit_idaction_name` int(11) unsigned NOT NULL,
  `visit_exit_idaction_url` int(11) unsigned DEFAULT '0',
  `referer_url` text NOT NULL,
  `location_browser_lang` varchar(20) NOT NULL,
  `config_browser_engine` varchar(10) NOT NULL,
  `config_browser_name` varchar(10) NOT NULL,
  `referer_type` tinyint(1) unsigned DEFAULT NULL,
  `referer_name` varchar(70) DEFAULT NULL,
  `visit_total_actions` smallint(5) unsigned NOT NULL,
  `visit_total_searches` smallint(5) unsigned NOT NULL,
  `referer_keyword` varchar(255) DEFAULT NULL,
  `location_provider` varchar(100) DEFAULT NULL,
  `custom_var_k1` varchar(200) DEFAULT NULL,
  `custom_var_v1` varchar(200) DEFAULT NULL,
  `custom_var_k2` varchar(200) DEFAULT NULL,
  `custom_var_v2` varchar(200) DEFAULT NULL,
  `custom_var_k3` varchar(200) DEFAULT NULL,
  `custom_var_v3` varchar(200) DEFAULT NULL,
  `custom_var_k4` varchar(200) DEFAULT NULL,
  `custom_var_v4` varchar(200) DEFAULT NULL,
  `custom_var_k5` varchar(200) DEFAULT NULL,
  `custom_var_v5` varchar(200) DEFAULT NULL,
  PRIMARY KEY (`idvisit`),
  KEY `index_idsite_config_datetime` (`idsite`,`config_id`,`visit_last_action_time`),
  KEY `index_idsite_datetime` (`idsite`,`visit_last_action_time`),
  KEY `index_idsite_idvisitor` (`idsite`,`idvisitor`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=1 ;
```

Bibliography

[1] R. Popa, N. Zeldovich, and H. Balakrishnan, “Cryptdb: A practical encrypted relational dbms. technical report mit-csail-tr-2011-005,” 2011.

[2] Experian, “Online id od: illegal web trade in personal information soars (accessed 2015-09-25),” 2012. [Online]. Available: https://www.experianplc.com/media/news/2012/illegal-web-trade-in-personal-information-soars/

[3] M. Bishop, “The insider problem revisited,” in *Proceedings of the 2005 workshop on New security paradigms*, ser. NSPW ’05. New York, NY, USA: ACM, 2005, pp. 75–76. [Online]. Available: http://doi.acm.org/10.1145/1146269.1146287

[4] B. Schneier, *Secrets and lies: digital security in a networked world*, ser. Wiley computer publishing. John Wiley, 2000. [Online]. Available: https://books.google.de/books?id=eNhQAAAAMAAJ

[5] Oxford English Dictionary, “"database, n.".” accessed: 2015-07-13. [Online]. Available: http://www.oed.com/view/Entry/47411?redirectedFrom=Database&amp;

[6] breachalert.com, “Breachalert (accessed 2015-09-25),” 2015. [Online]. Available: https://breachalarm.com/

[7] “Online cheating site ashleymadison hacked (accessed 2015-09-25),” http://krebsonsecurity.com/2015/07/online-cheating-site-ashleymadison-hacked/, 2015. [Online]. Available: http://krebsonsecurity.com/2015/07/online-cheating-site-ashleymadison-hacked/

[8] WallStreetJournal, “Irs says cyberattacks more extensive than previously reported (accessed 2015-09-25),” 2015. [Online]. Available: http://www.wsj.com/articles/irs-says-cyberattacks-more-extensive-than-previously-reported-1439834639

[9] ——, “U.s. suspects hackers in china breached about 4 million people’s records, officials say,” 2015. [Online]. Available: http://www.wsj.com/articles/u-s-suspects-hackers-in-china-behind-government-data-breach-sources-say-1433451888

[10] I. ISO, “Iec 9075: 2011 information technology, database languages,” 2011.

[11] “Db-engines ranking,” 2015. [Online]. Available: http://db-engines.com/en/ranking/relational+dbms

[12] M. Cooney, “Ibm touts encryption innovation; new technology performs calculations on encrypted data without decrypting it,” *Computer World, June*, 2009.

[13] P. Paillier, “Public-key cryptosystems based on composite degree residuosity classes,” in *Advances in cryptology (EUROCRYPT) 99*. Springer, 1999, pp. 223–238.

[14] T. Ge and S. Zdonik, “Answering aggregation queries in a secure system model,” in *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 519–530.

[15] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on.* IEEE, 2000, pp. 44–55.

[16] A. Boldyreva, N. Chenette, Y. Lee, and A. O'neill, "Order-preserving symmetric encryption," in *Advances in Cryptology-EUROCRYPT 2009.* Springer, 2009, pp. 224–241.

[17] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Order preserving encryption for numeric data," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data.* ACM, 2004, pp. 563–574.

[18] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: protecting confidentiality with encrypted query processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles.* ACM, 2011, pp. 85–100.

[19] I. H. Akin and B. Sunar, "On the difficulty of securing web applications using cryptdb," in *Big Data and Cloud Computing (BdCloud), 2014 IEEE Fourth International Conference on.* IEEE, 2014, pp. 745–752.

[20] C. V. W. Muhammad Naveed, Seny Kamara, "Inference attacks on property-preserving encrypted databases," 2015. [Online]. Available: http://research.microsoft.com/en-us/um/people/senyk/pubs/edb.pdf

[21] Google, "Encrypted bigquery client," https://github.com/google/encrypted-bigquery-client, 2015.

[22] P. Grofig, M. Haerterich, I. Hang, F. Kerschbaum, M. Kohler, A. Schaad, A. Schroepfer, and W. Tighzert, "Experiences and observations on the industrial implementation of a system to search over outsourced encrypted data." in *Sicherheit*, 2014, pp. 115–125.

[23] dev.mysql.com, *MySQL Documentation (accessed 2015-08-05)*, 2015. [Online]. Available: https://dev.mysql.com/doc/refman/5.0/en/compatibility.html

[24] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker, "Oltp through the looking glass, and what we found there," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08. New York, NY, USA: ACM, 2008, pp. 981–992. [Online]. Available: http://doi.acm.org/10.1145/1376616.1376713

[25] A. Kopytov, "Sysbench: a system performance benchmark," *URL: http://sysbench.sourceforge.net*, 2004.

[26] MySQL AB, "Mysql performance benchmarks," *A MySQL Technical White Paper*, 2005. [Online]. Available: http://www.jonahharris.com/osdb/mysql/mysql-performance-whitepaper.pdf

[27] J. Clarke, *SQL injection attacks and defense.* Elsevier, 2009.

[28] git scm.com, *Git - git-clone Documentation (accessed 2015-07-25)*, 2015. [Online]. Available: http://git-scm.com/docs/git-clone