Hello E63 person!
This is a document describing the process I used for the setup, running, and testing of CryptDB. Please excuse the fact that I wrote down my process in second person, I find it's easier to convey what I did that way, and easier for others to repeat the same steps.

This document serves as a report for what I did, and in some area elaborates more than my presentation, in some areas less.

CryptDB is a novel method of encrypting databases, which sits between the user and a database-management server (DBMS) as a proxy server. The user authenticates with a password to CryptDB, decrypting a keychain, which is used by the proxy to encrypt the user's database queries, enabling querying on encrypted data on the DBMS. This has the advantage of as little information leaking outside the closed system as possible, with the DBMS having no way of accessing the encrypted data. When a user is logged out, there is absolutely no information leakage as their keychain is fully encrypted, and even when logged in, CryptDB automatically peels off layers of encryption on the columns of the DBMS (referred to as 'onions' due to the many layers and varieties of encryption), selecting the most secure encryption available that still allows information to be processed in the requested way without ever fully decrypting the data.

I. SETING UP CRYPTDB

System requirements:
- An i386 or x86_64 based computer with either Ubuntu 12.04 or 13.04 installed, either directly on the machine or in a virtual machine (VirtualBox, VMWare...)
- Internet connection

Note: I prefer to use the Ubuntu server edition of 12.04.4 LTS (http://releases.ubuntu.com/12.04/), and then install a desktop environment for a more minimalistic, less bloated experience. I prefer xfce4, and you can install it by doing (without quotes) "sudo apt-get install xubuntu-desktop".

Once Ubuntu is installed, open up a terminal and update and upgrade the system:
```
sudo apt-get update
sudo apt-get upgrade
```

In a terminal, navigate to the directory that this document is in (or was in when you downloaded and extracted it), and run:
```
echo "export DEMOHOME=`pwd`" >> ~/.bashrc && source ~/.bashrc
```

Install git and ruby:
```
sudo apt-get install git ruby
```

Make sure you're in your home directory:

```
cd $HOME
```

Clone the MIT CSAIL CryptDB git repository:

```
git clone -b public git://g.csail.mit.edu/cryptdb && cd cryptdb
```

Set the $EDBDIR variable:

```
echo "export EDBDIR=$HOME/cryptdb" >> ~/.bashrc && source ~/.bashrc
```

Now, run the official install script, written in Ruby (for a reason I can not divine...):

```
$EBDIR/scripts/install.rb
```

The script is installing all dependencies for CryptDB, and compiling from source. At some point you will get asked to set the MySQL root password, this demo uses "letmein" (without quotes), so you should set that as the root password, too. The rest of this install could take hours on an older machine. It took at least an hour on my semi-modern machine.

Once the install script completes, you're golden.

2. RUNNING CRYPTDB

Now, in the scripts/ directory of the folder you found this document in (THIS document, the one you're reading now), there is a script "startcryptdb.sh" you can run, to start the CryptDB proxy server:

```
cd $DEMOHOME/ && ./scripts/startcryptdb.sh
```

3. TESTING CRYPTDB

Now that the proxy server is running, open up another terminal (I'll refer to this one as TERM2, and the one with the proxy as TERM1). Connect to CryptDB!:

```
cd $DEMOHOME/ && ./scripts/connecttocryptdb.sh
```

Now, open up another terminal (I'll refer to this one as TERM3) as well so you can do side-by-side comparison connecting directly to MySQL, and run this:

```
cd $DEMOHOME && ./scripts/connecttomysql.sh
```

In the mysql shell in TERM2 do:

```
create database cryptdbtest;
use cryptdbtest;
```

In the mysql shell in TERM3 do:

```
create database notencrypted;
use notencrypted;
```

Note how in TERM1, the proxy processes the query from TERM2 only, as that is the only one connected to mysql through the proxy.

Now, in both TERM2 and TERM3 do the following, observe the time it takes to process the queries:

```
source data/table.sql;
source data/DATA.sql;

select * from USERDATA where first='Aurora' and height=153 and last='Hensley';
```

In TERM2 the times should have been substantially longer, like this:

```
mysql> source data/table.sql;
Query OK, 0 rows affected (1.08 sec)
mysql> source data/DATA.sql
Query OK, 1 row affected (0.37 sec)
...
Query OK, 1 row affected (0.43 sec)
mysql>   select * from USERDATA where first='Aurora' and height=153 and last='Hensley';
+--------+---------+---------+--------------------------------------+--------+---------+
| first  | last    | balance | userid                               | height | zipcode |
+--------+---------+---------+--------------------------------------+--------+---------+
| Aurora | Hensley |    5064 | B48C6854-3B67-78C3-9295-8F52085B6CE9 |    153 |   62370 |
+--------+---------+---------+--------------------------------------+--------+---------+
1 row in set (3.94 sec)
```

Whereas in TERM3, the times should be substantially shorter, like this:

```
mysql> source data/table.sql
Query OK, 0 rows affected (0.29 sec)
mysql> source data/DATA.sql
Query OK, 1 row affected (0.10 sec)
...
Query OK, 1 row affected (0.06 sec)
mysql>   select * from USERDATA where first='Aurora' and height=153 and last='Hensley';
+--------+---------+---------+--------------------------------------+--------+---------+
| first  | last    | balance | userid                               | height | zipcode |
+--------+---------+---------+--------------------------------------+--------+---------+
| Aurora | Hensley |    5064 | B48C6854-3B67-78C3-9295-8F52085B6CE9 |    153 |   62370 |
+--------+---------+---------+--------------------------------------+--------+---------+
1 row in set (0.00 sec)
```

Hmm. What an annoying wait...I wonder if 4 seconds is really a standard query time? That really wouldn't be very nice...let's try to look up Aurora in TERM2 again:

```
mysql>   select * from USERDATA where first='Aurora' and height=153 and last='Hensley';
+--------+---------+---------+--------------------------------------+--------+---------+
| first  | last    | balance | userid                               | height | zipcode |
+--------+---------+---------+--------------------------------------+--------+---------+
| Aurora | Hensley | 5064    | B48C6854-3B67-78C3-9295-8F52085B6CE9 | 153    | 62370   |
+--------+---------+---------+--------------------------------------+--------+---------+
1 row in set (0.14 sec)
```

Wow, that was ~1/32 the time of the first lookup! Wonder what happened...
Taking a look at the proxy server's output (TERM1) yields this for the first query (right side is cut off due to formatting issues):

```
QUERY: select * from USERDATA where first='Aurora' and height=153 and last='Hensley'
Adjusting onion!
onion: oEq
ADJUST:
UPDATE `cryptdbtest`.table_YUCTXLMCMZ    SET OSVDNSJKVUoEq = cryptdb_decrypt_text_sem(`cryptdbtest`.`table_
...[redacted due to length, this part consists of a series encryption adjustment phases]
NEW QUERY: select `cryptdbtest`.`table_YUCTXLMCMZ`.`OSVDNSJKVUoEq`,`cryptdbtest`.`table_YUCTXLMCMZ`.`NEUKHH
ENCRYPTED RESULTS:
+--------------------+--------------------+--------------------+--------------------+--------------------+-+
|OSVDNSJKVUoEq       |NEUKHHEQRZoEq       |EITYUBHYTBoEq       |cdb_saltJYOEVIZEVP  |TOWZKIAWNToEq       |c
+--------------------+--------------------+--------------------+--------------------+--------------------+-+
|??(Y??????e?f?x??6R?K?????k9??K?|??A????;??qH??_/??6??D?ny??]^5??|13525511947258797221|8185390532925406636
+--------------------+--------------------+--------------------+--------------------+--------------------+-+
```

And this for the second:

```
QUERY: select * from USERDATA where first='Aurora' and height=153 and last='Hensley'
NEW QUERY: select `cryptdbtest`.`table_YUCTXLMCMZ`.`OSVDNSJKVUoEq`,`cryptdbtest`.`table_YUCTXLMCMZ`.`NEUKHHE
ENCRYPTED RESULTS:
+--------------------+--------------------+--------------------+--------------------+--------------------+--
|OSVDNSJKVUoEq       |NEUKHHEQRZoEq       |EITYUBHYTBoEq       |cdb_saltJYOEVIZEVP  |TOWZKIAWNToEq       |cd
+--------------------+--------------------+--------------------+--------------------+--------------------+--
|??(Y??????e?f?x??6R?K?????k9??K?|??A????;??qH??_/??6??D?ny??]^5??|13525511947258797221|8185390532925406636
+--------------------+--------------------+--------------------+--------------------+--------------------+--
================================================
```

See what happened? The proxy server realized it had to peel back some encryption layers from the onion ("Adjusting onion!") so that it could perform the operations necessary to look up Aurora by her first and last names, and her height (using the DET layer from the equality onion). Peeling back the encryption is what took all that time, and the proxy doesn't put back the top level encryption once it's been removed, to save time. If you read the query the proxy sends to the DBMS, and the results the DBMS sends back, they are the same for each query - it's just that in the second query, the proxy doesn't have to translate the query first, it already knows how to say it!

Now, let's take a look at another interesting fact. We haven't used the HOM (homomorphic) layer yet. Let's do that. Let's sum all of the balances of all these users together to see how much money there is total.
In TERM2:

```
mysql> select sum(balance) from USERDATA;
+--------------+
| sum(balance) |
+--------------+
| 449827       |
+--------------+
1 row in set (0.16 sec)
```

In TERM3:

```
mysql> select sum(balance) from USERDATA;
+--------------+
| sum(balance) |
+--------------+
|       449827 |
+--------------+
1 row in set (0.00 sec)
```

OK, the result from TERM3 is to be expected because there's no encryption. TERM2 also seems fast, which is awesome. Let's try it again:

In TERM2:

```
mysql> select sum(balance) from USERDATA;
+--------------+
| sum(balance) |
+--------------+
| 449827       |
+--------------+
1 row in set (0.15 sec)
```

Huh, that's strange. At least at first. But what's really happening here is that there is only one layer on the addition onion, so it is never 'peeled' off. It just so happens that this algorithm is also really efficient, so it seems fast for a good reason.

To really enforce the point of CryptDB, let's take a look at what happens when we pretend to be a nosy server admin try to snoop on the encrypted database:

In TERM3:

```
mysql> use cryptdbtest;
Database changed
mysql> show tables;
+----------------------+
| Tables_in_cryptdbtest |
+----------------------+
| table_YUCTXLMCMZ     |
+----------------------+
1 row in set (0.00 sec)
```

Oh, cool, CryptDB even encrypted the table name of USERDATA!

```
mysql> describe table_YUCTXLMCMZ;
+--------------------+--------------------+------+-----+---------+-------+
| Field              | Type               | Null | Key | Default | Extra |
+--------------------+--------------------+------+-----+---------+-------+
| OSVDNSJKVUoEq      | varbinary(288)     | YES  |     | NULL    |       |
| LVQOROBDSPoOrder   | bigint(20) unsigned | YES  |     | NULL    |       |
| cdb_saltHDQOBMSSQA | bigint(8) unsigned | YES  |     | NULL    |       |
| NEUKHHEQRZoEq      | varbinary(288)     | YES  |     | NULL    |       |
| NDWDTNTUTMoOrder   | bigint(20) unsigned | YES  |     | NULL    |       |
| cdb_saltFCSVSVHLOA | bigint(8) unsigned | YES  |     | NULL    |       |
| EITYUBHYTBoEq      | bigint(20) unsigned | YES  |     | NULL    |       |
| XGGWKZJDTEoOrder   | bigint(20) unsigned | YES  |     | NULL    |       |
| RECHIXQKELoADD     | varbinary(256)     | YES  |     | NULL    |       |
| cdb_saltJYOEVIZEVP | bigint(8) unsigned | YES  |     | NULL    |       |
| TOWZKIAWNToEq      | varbinary(288)     | YES  |     | NULL    |       |
| AFAFOIEESIoOrder   | bigint(20) unsigned | YES  |     | NULL    |       |
| cdb_saltZIWTJQWSPP | bigint(8) unsigned | YES  |     | NULL    |       |
| YSMWRYEUUUoEq      | bigint(20) unsigned | YES  |     | NULL    |       |
| NXZNCNYGUSoOrder   | bigint(20) unsigned | YES  |     | NULL    |       |
| ROMKEBOKKKoADD     | varbinary(256)     | YES  |     | NULL    |       |
| cdb_saltSUTONEGJWV | bigint(8) unsigned | YES  |     | NULL    |       |
| KFOLVQAVYWoEq      | bigint(20) unsigned | YES  |     | NULL    |       |
| NPKCZODPBQoOrder   | bigint(20) unsigned | YES  |     | NULL    |       |
| AJDEHMCBEUoADD     | varbinary(256)     | YES  |     | NULL    |       |
| cdb_saltILCWJWUQQR | bigint(8) unsigned | YES  |     | NULL    |       |
+--------------------+--------------------+------+-----+---------+-------+
21 rows in set (0.04 sec)
```

And of course, nothing much can be divined from this encrypted data. There are so many columns here, all an 'onion' for one of the encrypted columns. Let the nosy admin try one last thing - try to view all entries for all column:

Well, that seems like an effective encryption scheme!

Now, after I had explored CryptDB like this, I got curious how easy it would be to interface to CryptDB with Java. Knowing nothing about DBMSes, I didn't even know how to interface Java with MySQL. Some research led me to an open-source driver for MySQL, written by the MySQL team, for Java. It is a Jarfile that you can download from http://dev.mysql.com/downloads/connector/j/.

I wrote a quick-and-dirty java program that interfaced with MySQL and dumped the contents of a CSV file into a database. After some trial and error I managed to get it to connect with CryptDB perfectly! All I had to do was change the connection port in the Connection String from 3306 to 3307, and localhost to the loopback connection. Looking back it's kind of obvious.

In the scripts folder in the $DEMOHOME directory, is CDBExample.java, compile.sh, and run.sh. These assume you have a working JDK on your Ubuntu installation, which you can get by

running, without the quotes "sudo apt-get install default-jdk". In the folder "jars", in the scripts directory, is the MySQL driver, which is placed into the compile-time and run-time classed path, in the compile and run scripts. So, in a terminal, run:

```
$DEMOHOME/scripts/compile.sh
$DEMOHOME/scripts/run.sh
Connection to CryptDB successful!
Creating database cdbexample...
CREATE DATABASE cdbexample

success
Selecting databasecdbexample...
USE cdbexample

success
Creating table with name USERDATA and columns [first, last, balance, userid, height, zi
CREATE TABLE USERDATA (
first VARCHAR(255),
last VARCHAR(255),
balance INT,
userid VARCHAR(255),
height INT,
zipcode INT
)
[redacted for length]
INSERT INTO USERDATA (first, last, balance, userid, height, zipcode) VALUES('Kennan',
'Gibbs',
9946,
'D15C0244-F4C5-1E83-85E4-9C7125A8790D',
183,
61495
)
```

Aha, it works! So, this means you could easily write any sort of Java program to interface with CryptDB, which means that Hadoop is totally possible! Whether it would be worth it and under what circumstances is a different story, but it is definitely 'possible' to perform mapreduce operations with a DBMS without exposing unencrypted data to the database server!

To conclude, CryptDB is a novel solution to many of the problems of in database security, and the few operations supported by the proxy (equality, summation, like, range, equality join and range join) are more than enough for many applications. CryptDB's speed is impressive, especially for what it accomplishes; the security it features it offers are very robust, and the dynamic 'onion' encryption is very effective and quite efficient. All that being said, CryptDB was developed as a proof-of-concept, so there are many ways it can be improved (such as more mathematical operations, with the option for more 'onions' in order to keep the more complex layers away from the simpler ones). Hopefully, CryptDB will lead to a paradigm shift in industry, allowing customers more control of their data - and hopefully spark more interest in developing new homomorphic encryption strategies.