CRYPTDB: PROTECTING CONFIDENTIALITY WITH ENCRYPTED QUERY PROCESSING

RALUCA ADA POPA, CATHERINE M. S. REDFIELD, NICKOLAI ZELDOVICH, AND HARI BALAKRISHNAN
MIT CSAIL
Why

**Equifax**
- Sep, 2017
- “Equifax security breach leaks personal info of 143 million US consumers”

**Yahoo!**
- Dec, 2016
- “1 Billion User Accounts of Yahoo Were Hacked”

**LinkedIn**
- Jun, 2012
- “Hackers extracted 6.5 million hashed passwords from the DB of LinkedIn”

**PlayStation**
- Apr, 2011
- “Sony PlayStation Network, impacted 77 million personal information profiles”
Possible attacks

- **Cloud Admin**: Try to access your private data
- **Hacker**: Use bugs or exploits to access databases
- **Government**: May have physical access to databases
Goal

- Fast Real-world performance
- Safe Meaningful security
- Easy Large class of real application
## Challenge

**Employee Table**

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>70,000</td>
<td>23</td>
</tr>
<tr>
<td>Bob</td>
<td>50,000</td>
<td>25</td>
</tr>
</tbody>
</table>

**Encrypted Employee Table**

<table>
<thead>
<tr>
<th>gd58i9</th>
<th>s9i4j3e</th>
<th>2ki9o0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x638e5</td>
<td>x1eab8</td>
<td>x98f73</td>
</tr>
<tr>
<td>x922eb</td>
<td>x638e5</td>
<td>x73b41</td>
</tr>
</tbody>
</table>

- Query all the employees whose salary is greater than $60,000
Challenge

- For original database
  - SELECT * FROM Employee WHERE salary > 60000

- For encrypted database
  - SELECT * FROM Employee WHERE s9i4j3e > ?%#$&

- Sum
- Equality
- Order
- ......
Challenge

- Using Fully homomorphic encryption (FHE) [Gentry’09]
  - For any $op(i_1, i_2, ..., i_n) = r \iff op(fhe(i_1), fhe(i_2), ..., fhe(i_n)) = fhe(r)$
  - Prohibitively slow, e.g., slowdown X1,000,000,000
- Using strong and efficient cryptosystem such as AES
- Need to give the DBMS server access to the decryption key
Challenge

- Using Fully homomorphic encryption (FHE) [Gentry’09]
  - For any $op(i_1, i_2, ..., i_n) = r \iff op(fhe(i_1), fhe(i_2), ..., fhe(i_n)) = fhe(r)$
  - Prohibitively slow, e.g., slowdown X1,000,000,000
- Using strong and efficient cryptosystem such as AES
- Need to give the DBMS server access to the decryption key
Challenge

- How to minimize the amount of data leaked in such cases?
- How to ensure that a compromised application can obtain only a limited amount of decrypted data?
Two Threats

An original architecture
Threat 1: DBMS Compromise

An original architecture - passive DB server attacks
Threat 1: DBMS Compromise

Users Computer -> Application Server -> CryptDB Proxy Server -> DBMS Server

Users: User 1, User 2, User 3

Application: SQL input

Proxy: Plain query transformed to encrypted query

DB Server: Encrypted data

Architectural diagram with CryptDB Proxy Server
Threat 1: DBMS Compromise

- CryptDB Proxy: Master Key, Schema, How DB is encrypted
- Proxy transforms plain query, and decrypts the encrypted text from DB
Threat 1: DBMS Compromise
Threat 1: DBMS Compromise

Guarantee:
1. Confidentiality for data content and names of columns, tables.
2. Does not hide overall table structure, #row, type of columns, etc.
Threat1: DBMS Compromise

**Guarantee:**
1. Confidentiality for data content and names of columns, tables.
2. Does not hide overall table structure, #row, type of columns, etc

**Confidentiality Level? Depends on Application:**
1. If application requests no relational predicate filtering on a column: nothing leaks😃
2. If application requests equality check: reveals histogram😐
3. If application requests order check: reveals order😢
Threat 1: DBMS Compromise

**Guarantee:**
1. Confidentiality for data content and names of columns, tables.
2. Does not hide overall table structure, #row, type of columns, etc

**Confidentiality Level? Depends on Application:**
1. If application requests no relational predicate filtering on a column: nothing leaks 😊
2. If application requests equality check: reveals histogram 😐
3. If application requests order check: reveals order 😟

DROP DATABASE Midterm_Grades;
Threat 2: Arbitrary Threats

Threat 2: any attacks on all servers

Users Computer → Application → Proxy → DB Server → CryptDB UDFs

- Threat 1: passive DB server attacks
- Threat 2: any attacks on all servers
Threat 2: Arbitrary Threats

Threat 2: any attacks on all servers

Threat 1: passive DB server attacks

Architecture with CryptDB Proxy Server
Threat 2: Arbitrary Threats

Express finer-grained confidentiality policies: Encrypt data also with user keys!
... So only user1 can be attacked now.
Threat 2: Arbitrary Threats

Express finer-grained confidentiality policies: Encrypt data also with user keys!

... So only user1 can be attacked now.

Still do not protect User side.
**Threat 2: Arbitrary Threats**

**Threat 2: any attacks on all servers**

**Architecture with CryptDB Proxy Server**

- Express finer-grained confidentiality policies: Encrypt data also with user keys!
  
  ... So only user1 can be attacked now.

**Still do not protect User side.**
SQL-aware Encryption
SQL-aware Encryption

- CryptDB uses a number of existing cryptosystems:
SQL-aware Encryption

- CryptDB uses a number of existing cryptosystems:
- Random (RND). RND provides the maximum security. Even two equal values are mapped to different ciphertexts with overwhelming probability.
SQL-aware Encryption

- CryptDB uses a number of existing cryptosystems:
- Random (RND). RND provides the maximum security. Even two equal values are mapped to different ciphertexts with overwhelming probability.
- Deterministic (DET): DET has a slightly weaker guarantee. This encryption layer allows the server to perform equality checks.
SQL-aware Encryption

- CryptDB uses a number of existing cryptosystems:

- Random (RND). RND provides the maximum security. Even two equal values are mapped to different ciphertexts with overwhelming probability.

- Deterministic (DET): DET has a slightly weaker guarantee. This encryption layer allows the server to perform equality checks

- Order-preserving encryption (OPE): OPE allows order relations between data items to be established based on their encrypted values, without revealing the data itself.
SQL-aware Encryption

- CryptDB uses a number of existing cryptosystems:
  - Random (RND). RND provides the maximum security. Even two equal values are mapped to different ciphertexts with overwhelming probability.
  - Deterministic (DET): DET has a slightly weaker guarantee. This encryption layer allows the server to perform equality checks.
  - Order-preserving encryption (OPE): OPE allows order relations between data items to be established based on their encrypted values, without revealing the data itself.
  - Homomorphic encryption (HOM), Join (JOIN and OPE-JOIN), and Word search (SEARCH), etc.
Adjustable Query-based Encryption
Adjustable Query-based Encryption
Adjustable Query-based Encryption

- Each value is dressed in layers of increasingly stronger encryption
Adjustable Query-based Encryption

- Each value is dressed in layers of increasingly stronger encryption
- Each layer of each onion enables certain kinds of functionality
Adjustable Query-based Encryption

- Each value is dressed in layers of increasingly stronger encryption
- Each layer of each onion enables certain kinds of functionality
- Multiple onions are needed for compatibility and performance
Adjustable Query-based Encryption
Adjustable Query-based Encryption
Adjustable Query-based Encryption

- The proxy strips off the onion layers to allow different operations
Adjustable Query-based Encryption

- The proxy strips off the onion layers to allow different operations
- The proxy never decrypts the data past the least-secure encryption onion layer
EXAMPLE (FROM THE AUTHORS’ SLIDES):
EXAMPLE (FROM THE AUTHORS’ SLIDES):

<table>
<thead>
<tr>
<th>rank</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘CEO’</td>
<td></td>
<td></td>
</tr>
<tr>
<td>‘worker’</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
EXAMPLE (FROM THE AUTHORS’ SLIDES):

<table>
<thead>
<tr>
<th>emp</th>
<th>rank</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>‘CEO’</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>‘worker’</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**table 1:**

<table>
<thead>
<tr>
<th>col1-OnionEq</th>
<th>col1-OnionOrder</th>
<th>col1-OnionSearch</th>
<th>col2-OnionEq</th>
</tr>
</thead>
<tbody>
<tr>
<td>RND</td>
<td>RND</td>
<td>SEARCH</td>
<td>RND</td>
</tr>
<tr>
<td>RND</td>
<td>RND</td>
<td>SEARCH</td>
<td>RND</td>
</tr>
</tbody>
</table>

...
EXAMPLE (FROM THE AUTHORS’ SLIDES):

<table>
<thead>
<tr>
<th>rank</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>'CEO'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'worker'</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>table 1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>col1-OnionEq</td>
</tr>
<tr>
<td>RND</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Onion Equality

RND

JOIN

'CEO'

DET
SELECT * FROM emp WHERE rank = 'CEO';
### Onion Equality

**JOIN**

<table>
<thead>
<tr>
<th>col1-OnionEq</th>
<th>col1-OnionOrder</th>
<th>col1-OnionSearch</th>
<th>col2-OnionEq</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>RND</td>
<td>RND</td>
<td>SEARCH</td>
<td>RND</td>
<td>...</td>
</tr>
<tr>
<td>RND</td>
<td>RND</td>
<td>SEARCH</td>
<td>RND</td>
<td></td>
</tr>
</tbody>
</table>
SELECT * FROM emp WHERE rank = 'CEO';
```
SELECT * FROM emp WHERE rank = 'CEO';
```
EXAMPLE (CONT’D)

SELECT * FROM emp WHERE rank = ‘CEO’;

UPDATE table1 SET col1-OnionEq = Decrypt_RND(key, col1-OnionEq);
### Example (Cont’d)

**SELECT** * FROM `emp` **WHERE** `rank` = ‘CEO’;

**UPDATE** `table1` **SET** `col1-OnionEq` = 
* Decrypt_RND*(key, `col1-OnionEq`);

**SELECT** * FROM `table1` **WHERE** `col1-OnionEq` = xda5c0407;
SELECT * FROM emp WHERE rank = 'CEO';

UPDATE table1 SET col1-OnionEq = Decrypt_RND(key, col1-OnionEq);

SELECT * FROM table1 WHERE col1-OnionEq = xda5c0407;
SELECT * FROM emp WHERE rank = 'CEO';

UPDATE table1 SET col1-OnionEq = Decrypt_RND(key, col1-OnionEq);

SELECT * FROM table1 WHERE col1-OnionEq = xda5c0407;
SELECT * FROM emp WHERE rank = 'CEO';

UPDATE table1 SET col1-OnionEq = Decrypt_RND(key, col1-OnionEq);

SELECT * FROM table1 WHERE col1-OnionEq = xda5c0407;
SELECT * FROM emp WHERE rank = 'CEO';

UPDATE table1 SET col1-OnionEq = Decrypt_RND(key, col1-OnionEq);

SELECT * FROM table1 WHERE col1-OnionEq = xda5c0407;
SELECT * FROM emp WHERE rank = 'CEO';

UPDATE table1 SET col1-OnionEq = Decrypt_RND(key, col1-OnionEq);

SELECT * FROM table1 WHERE col1-OnionEq = xda5c0407;
SELECT * FROM emp WHERE rank = 'CEO';

UPDATE table1 SET col1-OnionEq = Decrypt_RND(key, col1-OnionEq);

SELECT * FROM table1 WHERE col1-OnionEq = xda5c0407;
SELECT * FROM emp WHERE rank = 'CEO';

UPDATE table1 SET col1-OnionEq = Decrypt_RND(key, col1-OnionEq);

SELECT * FROM table1 WHERE col1-OnionEq = xda5c0407;
SELECT * FROM emp WHERE rank = 'CEO';

UPDATE table1 SET col1-OnionEq = Decrypt_RND(key, col1-OnionEq);

SELECT * FROM table1 WHERE col1-OnionEq = xda5c0407;
System design

Application → SQL Interface → CryptDB Proxy → transformed query → encrypted results → Unmodified DBMS → Server → CryptDB SQL UDFs (user-defined functions) → results
System design

- Application
- SQL Interface
- CryptDB Proxy
- Unmodified DBMS
- Server
- CryptDB SQL UDFs (user-defined functions)

Flow:
- Application sends query to SQL Interface
- SQL Interface sends query to CryptDB Proxy
- CryptDB Proxy transforms query and sends it to Unmodified DBMS
- Unmodified DBMS performs query and sends encrypted results to CryptDB Proxy
- CryptDB Proxy decrypts results and sends it to SQL Interface
- SQL Interface sends results to Application
So far the first threat is solved.
System design

- So far the first threat is solved.
- What if the proxy and application are also untrusted?
System design

- So far the first threat is solved.
- What if the proxy and application are also untrusted?
Application protection

User 1
User 2
User 3
Application
Proxy
SQL
DB Server
Application protection

User 1
User 2
User 3

Application
Proxy

SQL

DB Server
Application protection

User 1

Application

Proxy

User 2

User 3

SQL

DB Server
Application protection

User 1
User 2
User 3

Application
Proxy
SQL
DB Server
Application protection

User 1 → Application → Proxy → SQL → DB Server

User 2

User 3
Application protection

- Protect data of logged-out users.
Application protection

- Protect data of logged-out users.
- Leaking data of active users is unavoidable.
Data sharing
Data sharing

➢ Access control is easy if proxy has all the keys
Data sharing

➢ Access control is easy if proxy has all the keys
Data sharing

➢ Access control is easy if proxy has all the keys

➢ But we want to protect the data of logged out users
Data sharing

➢ Access control is easy if proxy has all the keys

➢ But we want to protect the data of logged out users
Policy Annotations

——— Define Privileges and Access Controls
Policy Annotations

——— Define Privileges and Access Controls

Principal: entities such as users, groups, or messages
Policy Annotations

Define Privileges and Access Controls

Principal: entities such as users, groups, or messages

• Internal: Delegation
Policy Annotations

--- Define Privileges and Access Controls

**Principal**: entities such as users, groups, or messages

- **Internal**: Delegation

  Privileges are restricted by the delegation rules in the **DB table**
Policy Annotations

--- Define Privileges and Access Controls

**Principal**: entities such as users, groups, or messages

- Internal: Delegation
  Privileges are restricted by the delegation rules in DB table
- External: End user who logs in with password
Policy Annotations

—— Define Privileges and Access Controls

Principal: entities such as users, groups, or messages
- Internal: Delegation
  Privileges are restricted by the delegation rules in DB table
- External: End user who logs in with password
  Privileges are obtained through proxy after providing password.
Annotation: developer specified

ENC_FOR: which column has secret and what principals have access to those secret.

SPEAKS_FOR: if A delegates B, then A has access to all keys B has access to

```sql
CREATE TABLE privmsgs (  
  msgid int,  
  subject varchar(255) ENC_FOR (msgid msg),  
  msgtext text  
  ENC_FOR (msgid msg) );

CREATE TABLE users (  
  userid int,  
  username varchar(255),  
  (username physical_user) SPEAKS_FOR (userid user) );
```
Key Chaining
— handling the access control keys
Key Chaining
—–handling the access control keys

Four special tables in DB for access control
Key Chaining
—–handling the access control keys

Four special tables in DB for access control

Access_keys table
• Common symmetric key for principals that are all active
Key Chaining
—–handling the access control keys

Four special tables in DB for access control

Access_keys table
  • Common symmetric key for principals that are all active

Public_keys table
  • Asymmetric key for inactive principals
Key Chaining
—handling the access control keys

Four special tables in DB for access control

Access_keys table
  • Common symmetric key for principals that are all active

Public_keys table
  • Asymmetric key for inactive principals

External_keys table
  • Random key generated by principal password indicating its privilege
Key Chaining  
— handling the access control keys

Four special tables in DB for access control

Access_keys table
  • Common symmetric key for principals that are all active

Public_keys table
  • Asymmetric key for inactive principals

External_keys table
  • Random key generated by principal password indicating its privilege

Cryptdb_active table
  • Indicating whether principal is active, remove its key if not
Key Chaining
— handling the access control keys

Four special tables in DB for access control

Access_keys table
• Common symmetric key for principals that are all active

Public_keys table
• Asymmetric key for inactive principals

External_keys table
• Random key generated by principal password indicating its privilege

Cryptdb_active table
• Indicating whether principal is active, remove its key if not
Key Chaining
—— handling the access control keys

Four special tables in DB for access control

Internal Principal

Access_keys table
• Common symmetric key for principals that are all active

Public_keys table
• Asymmetric key for inactive principals

External_keys table
• Random key generated by principal password indicating its privilege

Cryptdb_active table
• Indicating whether principal is active, remove its key if not
Key Chaining
—–handling the access control keys

Four special tables in DB for access control

- **Access_keys table**
  - Common symmetric key for principals that are all active

- **Public_keys table**
  - Asymmetric key for inactive principals

- **External_keys table**
  - Random key generated by principal password indicating its privilege

- **Cryptdb_active table**
  - Indicating whether principal is active, remove its key if not
Key Chaining
—- handling the access control keys

Four special tables in DB for access control

- **Access_keys table**
  - Common symmetric key for principals that are all active

- **Public_keys table**
  - Asymmetric key for inactive principals

- **External_keys table**
  - Random key generated by principal password indicating its privilege

- **Cryptdb_active table**
  - Indicating whether principal is active, remove its key if not
• Internal Principal
• **Internal Principal**
  1. Symmetric key
     Says A speaks for B and B is active, then B’s symmetric key is encrypted using A’s symmetric key
  2. Asymmetric key
     Says A send a message to B, but B is offline. So CryptDB looks up the table for B’s public key, which can only be decrypted by its private key.
**Internal Principal**

1. Symmetric key
   Says A speaks for B and B is active, then B’s symmetric key is encrypted using A’s symmetric key.

2. Asymmetric key
   Says A send a message to B, but B is offline. So CryptDB looks up the table for B’s public key, which can only be decrypted by its private key.

**External Principal**

1. Random key
   When logged in, external principals are assigned a random key. When logged out, the related keys to that principals are removed.
Chaining Behavior

- A speaks for B: B’s key is encrypted by A’s key and stored in a DB table
- B speaks for C: C’s key is encrypted by B’s key and stored in a DB table
Chaining Behavior

- A speaks for B: B’s key is encrypted by A’s key and stored in a DB table
- B speaks for C: C’s key is encrypted by B’s key and stored in a DB table

When A wants to get C’s key and retrieve its principal (sensitive message)
Chaining Behavior

- A speaks for B: B’s key is encrypted by A’s key and stored in a DB table
- B speaks for C: C’s key is encrypted by B’s key and stored in a DB table

When A wants to get C’s key and retrieve its principal (sensitive message)
EXPERIMENTAL EVALUATION
EXPERIMENTAL EVALUATION

- Application Changes
EXPERIMENTAL EVALUATION

- Application Changes
- Functional Evaluation
EXPERIMENTAL EVALUATION

- Application Changes
- Functional Evaluation
- Security Evaluation
EXPERIMENTAL EVALUATION

- Application Changes
- Functional Evaluation
- Security Evaluation
- Performance Evaluation
Application Changes
## Application Changes

<table>
<thead>
<tr>
<th>Application</th>
<th>Annotations</th>
<th>Login/logout code</th>
<th>Sensitive fields secured, and examples of such fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>phpBB</td>
<td>31 (11 unique)</td>
<td>7 lines</td>
<td>23: private messages (content, subject), posts, forums</td>
</tr>
<tr>
<td>HotCRP</td>
<td>29 (12 unique)</td>
<td>2 lines</td>
<td>22: paper content and paper information, reviews</td>
</tr>
<tr>
<td>grad-apply</td>
<td>111 (13 unique)</td>
<td>2 lines</td>
<td>103: student grades (61), scores (17), recommendations, reviews</td>
</tr>
<tr>
<td>TPC-C (single princ.)</td>
<td>0</td>
<td>0</td>
<td>92: all the fields in all the tables encrypted</td>
</tr>
</tbody>
</table>
Functional Evaluation
## Functional Evaluation

<table>
<thead>
<tr>
<th>Application</th>
<th>Total cols.</th>
<th>Consider for enc.</th>
<th>Needs plaintext</th>
</tr>
</thead>
<tbody>
<tr>
<td>phpBB</td>
<td>563</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>HotCRP</td>
<td>204</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>grad-apply</td>
<td>706</td>
<td>103</td>
<td>0</td>
</tr>
<tr>
<td>OpenEMR</td>
<td>1297</td>
<td>566</td>
<td>7</td>
</tr>
<tr>
<td>MIT 6.02</td>
<td>15</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>PHP-calendar</td>
<td>25</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>TPC-C</td>
<td>92</td>
<td>92</td>
<td>0</td>
</tr>
<tr>
<td>Trace from sql.mit.edu</td>
<td>128840</td>
<td>128840</td>
<td>1094</td>
</tr>
<tr>
<td>... with in-proxy processing</td>
<td>128840</td>
<td>128840</td>
<td>571</td>
</tr>
<tr>
<td>... col. name contains pass</td>
<td>2029</td>
<td>2029</td>
<td>2</td>
</tr>
<tr>
<td>... col. name contains content</td>
<td>2521</td>
<td>2521</td>
<td>0</td>
</tr>
<tr>
<td>... col. name contains priv</td>
<td>173</td>
<td>173</td>
<td>0</td>
</tr>
</tbody>
</table>
Non-plaintext cols. with MinEnc
## Non-plaintext cols. with MinEnc

<table>
<thead>
<tr>
<th>Application</th>
<th>RND</th>
<th>SEARCH</th>
<th>DET</th>
<th>OPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>phpBB</td>
<td>21</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>HotCRP</td>
<td>18</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>grad-apply</td>
<td>95</td>
<td>0</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>OpenEMR</td>
<td>526</td>
<td>2</td>
<td>12</td>
<td>19</td>
</tr>
<tr>
<td>MIT 6.02</td>
<td>7</td>
<td>0</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>PHP-calendar</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>TPC-C</td>
<td>65</td>
<td>0</td>
<td>19</td>
<td>8</td>
</tr>
<tr>
<td>Trace from sql.mit.edu</td>
<td>80053</td>
<td>350</td>
<td>34212</td>
<td>13131</td>
</tr>
<tr>
<td>... with in-proxy processing</td>
<td>84008</td>
<td>398</td>
<td>35350</td>
<td>8513</td>
</tr>
<tr>
<td>... col. name contains pass</td>
<td>1936</td>
<td>0</td>
<td>91</td>
<td>0</td>
</tr>
<tr>
<td>... col. name contains</td>
<td>2215</td>
<td>52</td>
<td>251</td>
<td>3</td>
</tr>
<tr>
<td>... col. name contains priv</td>
<td>159</td>
<td>0</td>
<td>12</td>
<td>2</td>
</tr>
</tbody>
</table>
Performance Evaluation
Performance Evaluation

- 2.4GHz Intel Xeon E5620 4-core processor
Performance Evaluation

- 2.4GHz Intel Xeon E5620 4-core processor
- 12 GB RAM
EXPERIMENT ENVIRONMENT

Performance Evaluation

- 2.4GHz Intel Xeon E5620 4-core processor
- 12 GB RAM
- MySQL 5.1.54 server
EXPERIMENT ENVIRONMENT

Performance Evaluation

- 2.4GHz Intel Xeon E5620 4-core processor
- 12 GB RAM
- MySQL 5.1.54 server
Performance Evaluation

- 2.4GHz Intel Xeon E5620 4-core processor
- 12 GB RAM
- MySQL 5.1.54 server
- eight 2.4GHz AMD Opteron 8431 6-core processors
EXPERIMENT ENVIRONMENT

Performance Evaluation

- 2.4GHz Intel Xeon E5620 4-core processor
- 12 GB RAM
- MySQL 5.1.54 server
- eight 2.4GHz AMD Opteron 8431 6-core processors
- 64GB RAM
Performance Evaluation
Performance Evaluation

![Graph showing the performance evaluation of MySQL and CryptDB.

The x-axis represents the number of server cores, ranging from 1 to 8.

The y-axis represents queries per second, ranging from 0 to 50000.

- MySQL is represented by blue circles.
- CryptDB is represented by red circles.

The graph shows a clear upward trend for both MySQL and CryptDB as the number of server cores increases. MySQL consistently outperforms CryptDB across all core counts.

From the graph, it is evident that adding more server cores significantly increases the number of queries processed per second for both systems, with MySQL consistently achieving higher query rates compared to CryptDB.

This indicates that MySQL is more scalable and efficient in handling increased load compared to CryptDB.
DRAWBACKS
DRAWBACKS

- More storage
DRAWBACKS

- More storage
  - Multiple onions for the same field
DRAWBACKS

- More storage
  - Multiple onions for the same field
  - Ciphertexts are larger than plaintexts for some encryption schemes
DRAWBACKS

- More storage
  - Multiple onions for the same field
  - Ciphertexts are larger than plaintexts for some encryption schemes
- CryptDB cannot perform server-side computations on values encrypted for different principals because the ciphertexts are encrypted with different keys.
**DRAWBACKS**

- More storage
  - Multiple onions for the same field
  - Ciphertexts are larger than plaintexts for some encryption schemes
- CryptDB cannot perform server-side computations on values encrypted for different principals because the ciphertexts are encrypted with different keys.
- There are certain computations CryptDB cannot support on encrypted data
**DRAWBACKS**

- More storage
  - Multiple onions for the same field
  - Ciphertexts are larger than plaintexts for some encryption schemes
- CryptDB cannot perform server-side computations on values encrypted for different principals because the ciphertexts are encrypted with different keys.
- There are certain computations CryptDB cannot support on encrypted data
  - For example, it does not support both computation and comparison on the same column, such as WHERE salary > age*2+10.
DRAWBACKS

- More storage
  - Multiple onions for the same field
  - Ciphertexts are larger than plaintexts for some encryption schemes
- CryptDB cannot perform server-side computations on values encrypted for different principals because the ciphertexts are encrypted with different keys.
- There are certain computations CryptDB cannot support on encrypted data
  - For example, it does not support both computation and comparison on the same column, such as WHERE salary > age*2+10.
- Removing an onion layer is bottlenecked by the speed at which the DBMS server can copy a column from disk for disk-bound databases.