CS573 Data Privacy and Security

CryptDB: Protecting Confidentiality with Encrypted Query Processing

Li Xiong

Slides credit: Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and HariBalakrishnan
MIT CSAIL
Confidential data leaks from databases

- 2012: hackers extracted 6.5 million hashed passwords from the DB of LinkedIn
- Sony Playstation Network, accessed 77 million user profiles

Threat 2: any attacks on all servers

Threat 1: passive DB server attacks
CryptDB in a nutshell

- Goal: protect confidentiality of data

1. Process SQL queries on encrypted data
2. Use fine-grained keys; chain these keys to user passwords based on access control

Threat 1: passive DB server attacks
Threat 2: any attacks on all servers
Contributions

1. First practical DBMS to process most SQL queries on encrypted data
   Hide DB from sys. admins., outsource DB

2. Modest overhead: 26% throughput loss for TPC-C

3. No changes to DBMS (e.g., Postgres, MySQL) and no changes to applications
Most SQL uses a limited set of operations. Security: Reveal only relations among data that are required by classes of queries issued.

Unencrypted databases

CryptDB

FHE

[CryptDB, Gentry’09, BV’11,12, GHS’12,..]

Fast

High degree of security

Insecure

Fast

Strong security

Slow

Query input: $x_{24ab1c}$

Output:

<table>
<thead>
<tr>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>xa32601</td>
</tr>
<tr>
<td>x8199f3</td>
</tr>
<tr>
<td>x62d03b</td>
</tr>
<tr>
<td>xcef3f7</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Unencrypted databases

<table>
<thead>
<tr>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x4be2</td>
</tr>
<tr>
<td>x95c6</td>
</tr>
<tr>
<td>x2ea8</td>
</tr>
<tr>
<td>x17ce</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

query
x98aa = ?

index

CryptDB

FHE

Unencrypted databases
fast
insecure

CryptDB
fast
high degree
of security

FHE
slow
strong security

Other work: weaker security, functionality, and/or efficiency:

- Search on encrypted data (e.g., [Song et al.,’00])
- Systems proposals (e.g., [Hacigumus et al.,’02])
- Rewrite the DBMS, significant client-side processing
System Setup

- **Application**
  - Plain query
  - Decrypted results

- **Proxy**
  - Stores schema, master key
  - No data storage
  - No query execution

- **DB Server**
  - Under attack 🤡
  - Process queries completely at the DBMS, on encrypted database

- **Encrypted DB**
  - Transformed query
  - Encrypted results

- **Process SQL queries on encrypted data**
Application

SELECT * FROM emp
WHERE salary = 100

Proxy

SELECT * FROM table1
WHERE col3 = x5a8c34

Table 1/emp

<table>
<thead>
<tr>
<th>col1/rank</th>
<th>col2/name</th>
<th>col3/salary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>x934b1-c1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x5a8c34</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x84a21c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x5a8c34</td>
</tr>
</tbody>
</table>

Randomized encryption

Deterministic encryption
The diagram illustrates a process where an Application sends a query to a Proxy. The query is to select all rows from the `emp` table where `salary` is greater than or equal to 100. The Proxy then directly queries the `emp` table with the same condition and returns the filtered data to the Application. The data is presented in a table format, showing columns for `col1/rank`, `col2/name`, and `col3/salary`, with some rows filtered out due to the condition.

Additionally, there is a mention of deterministic encryption, which is depicted in the diagram but not directly related to the query depicted above.
Two techniques

1. Use SQL-aware set of encryption schemes

   Most SQL uses a limited set of operations
   Having encryption schemes that covers most common SQL operations

2. Adjust encryption of database based on queries

   • Different queries required that data to be encrypted with different encryption schemes
## Encryption schemes

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Construction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>RND</td>
<td>AES in CBC</td>
<td>none</td>
</tr>
<tr>
<td>HOM</td>
<td>Paillier</td>
<td>+, *</td>
</tr>
<tr>
<td>SEARCH</td>
<td>Song et al., '00</td>
<td>word search</td>
</tr>
<tr>
<td>DET</td>
<td>AES in CMC</td>
<td>equality</td>
</tr>
<tr>
<td>JOIN</td>
<td>our new scheme</td>
<td>join</td>
</tr>
<tr>
<td>OPE</td>
<td>Boldyreva et al. '09</td>
<td>order</td>
</tr>
</tbody>
</table>

Security:
- **Highest**
  - Equality matches bw 2 columns
  - Fullword matching
  - e.g., >, <, ORDER BY, DISTINCT

Function:
- e.g., sum
- restricted ILIKE
- e.g., =, !=, IN, COUNT, GROUP BY, DISTINCT
- Equality matches bw 2 columns
- first implementation
JOIN

- Equality checks between two columns
- Do not know columns to be joined a priori!

KeyGen (sec. param): SK

Encrypt (SK, m, col i): C_m^i (with ) - deterministic

Token (SK, col i, col j): (t_i, t_j)

Adjust (t_i, C_m^i): C_m (with )

Correctness: adjustment yields correct join relations
JOIN (cont’d)

- Security: do not learn join relations without token

- Implementation:
  - 192 bits long, 0.52 ms encrypt, 0.56 ms adjust
OPE

- Preserve the order of ciphertext to remain as they were in plaintext.

- For example, for any secret key $K$, if $x < y$, then $\text{OPE}_K(x) < \text{OPE}_K(y)$.
  - if a column is encrypted with OPE, the server can perform range queries when given encrypted constants $\text{OPE}_K(c_1)$ and $\text{OPE}_K(c_2)$ corresponding to the range $[c_1, c_2]$.

- OPE is a weaker encryption scheme than DET because it reveals order.
## Encryption schemes

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Construction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>RND</td>
<td>AES in CBC</td>
<td>none</td>
</tr>
<tr>
<td>HOM</td>
<td>Paillier</td>
<td>+, *</td>
</tr>
<tr>
<td>SEARCH</td>
<td>Song et al., ‘00</td>
<td>word search</td>
</tr>
<tr>
<td>DET</td>
<td>AES in CMC</td>
<td>equality</td>
</tr>
<tr>
<td>JOIN</td>
<td>our new scheme</td>
<td>join</td>
</tr>
<tr>
<td>OPE</td>
<td>Boldyreva et al.’09 + our new scheme</td>
<td>order</td>
</tr>
</tbody>
</table>
How to encrypt each data item?

- Encryption schemes needed depend on queries
- May not know queries ahead of time

<table>
<thead>
<tr>
<th>rank</th>
<th>col1-RND</th>
<th>col1-HOM</th>
<th>col1-SEARCH</th>
<th>col1-DET</th>
<th>col1-JOIN</th>
<th>col1-OPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘CEO’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>‘worker’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ALL? Leaks order!
Onions of encryptions

Ideal: To stack encryption schemes into onion of encryption

- Same key for all items in a column for same onion layer
- Start out the database with the most secure encryption scheme
Onions of encryptions

- Novel way to compactly store multiple ciphertexts within each other in the database and avoid expensive re-encryptions
- Each value is dressed in layers of increasingly stronger encryption
- Each layer of each onion enables certain kinds of functionality
- For each layer of each onion, the proxy uses the same key for encrypting values in the same column and
- Different keys across tables, columns, onions, and onion layers
Adjust encryption

- Dynamically adjusts the layer of encryption on the DBMS server
- Strip off layers of the onions
  - Proxy gives keys to server using a SQL UDF ("user-defined function")
  - Proxy remembers onion layer for columns
- Do not put back onion layer
Example:

```
SELECT * FROM emp WHERE rank = 'CEO';
```
UPDATE table1 SET col1-OnionEq = Decrypt_RND(key, col1-OnionEq);

SELECT * FROM emp WHERE rank = 'CEO';

UPDATE table1 SET col1-OnionEq = xda5c0407;

SELECT * FROM table1 WHERE col1-OnionEq = xda5c0407;
Confidentiality level

Queries ➔ encryption scheme exposed ➔ amount of leakage

- Encryption schemes exposed for each column are the most secure enabling queries
  - equality predicate on a column ➔ DET ➔ repeats
  - aggregation on a column ➔ HOM ➔ nothing
  - no filter on a column ➔ RND ➔ nothing

*common in practice*

Never reveals plaintext
Implementation

- No change to the DBMS
- **Portable**: from Postgres to MySQL with 86 lines
- no change to applications
Evaluation

1. Does it support real queries/applications?
2. What is the resulting confidentiality?
3. What is the performance overhead?
Queries not supported

- More complex operators, e.g., trigonometry
- Operations that require combining incompatible encryption schemes
  - e.g., T1.a + T1.b > T2.c

Extensions: split queries, precompute columns, or add new encryption schemes
### Real queries/applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Total columns</th>
<th>Encrypted columns</th>
<th># cols not supported</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>TPC-C</td>
<td>92</td>
<td>92</td>
<td>0</td>
</tr>
<tr>
<td>sql.mit.edu</td>
<td>128,840</td>
<td>128,840</td>
<td>1,094</td>
</tr>
</tbody>
</table>

```
SELECT 1/log(series_no+1.2) ...
... WHERE sin(latitude + PI()) ...
```
## Resulting confidentiality

<table>
<thead>
<tr>
<th>Application</th>
<th>Total columns</th>
<th>Encrypted columns</th>
<th>Min level is RND</th>
<th>Min level is DET</th>
<th>Min level is OPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>phpBB</td>
<td>563</td>
<td>23</td>
<td>21</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>HotCRP</td>
<td>204</td>
<td>22</td>
<td>18</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>grad-apply</td>
<td>706</td>
<td>103</td>
<td>95</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>TPC-C</td>
<td>92</td>
<td>92</td>
<td>65</td>
<td>19</td>
<td>8</td>
</tr>
<tr>
<td>sql.mit.edu</td>
<td>128,840</td>
<td>128,840</td>
<td>80,053</td>
<td>34,212</td>
<td>13,131</td>
</tr>
</tbody>
</table>

Most columns at RND were less sensitive.
Performance

MySQL:

Application 1

Application 2

DB server throughput

Plain database

CryptDB:

Application 1

Application 2

CryptDB Proxy

CryptDB Proxy

Encrypted database

Latency

Hardware: 2.4 GHz Intel Xeon E5620 – 8 cores, 12 GB RAM
TPC-C performance

- **Latency (ms/query):** 0.10 MySQL vs. 0.72 ms CryptDB

Max. Throughput loss 26%
No cryptography at the DB server in the steady state!

Homomorphic addition

Encrypted DBMS is practical
Conclusions

CryptDB:

1. The first practical DBMS for running most standard queries on encrypted data
2. Protects data of users logged out during attack even when all servers are compromised
3. Modest overhead and no changes to DBMS

Website: http://css.csail.mit.edu/cryptdb/