CryptDB: Processing Queries on an Encrypted Database

Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan
MIT CSAIL
Problem

- Confidential data leaks from databases (DB)
  - 2012: hackers extracted 6.5 million hashed passwords from the DB of LinkedIn

- Process SQL queries on encrypted data
Contributions

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

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

3. No changes to DBMS (e.g., Postgres, MySQL) and no changes to applications
Unencrypted databases

salary
60
100
800
... 

query?

FHE

query input

salary

xa32601
x8199f3
x62d03b
xcef3f7
...

fast

insecure

fast

high degree of security

slow

strong security

Most SQL uses a limited set of operations
Security: Reveal only relations among data that are required by queries at column granularity

100 =

query

Gentry’09, [GHS’12]...

Circuit C

output

CryptDB

index

salary

x4be2
x95c6
x2ea8
x17ce
...

query

x98aa = ?

query

100 =
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])
- Require significant client-side processing
System Setup

- **Application**
  - plain query
  - decrypted results

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

- **DB Server**
  - Encrypted DB
  - Process queries completely at the DBMS, on encrypted database

- **Under passive attack** 😈

- **Trusted**
Application

SELECT * FROM `emp`
WHERE `salary` = 100

Proxy

SELECT * FROM `table1`
WHERE `col3` = x5a8c34

<table>
<thead>
<tr>
<th>col1/rank</th>
<th>col2/name</th>
<th>col3/salary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>x9341c1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x5a834</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
```
SELECT * FROM emp
WHERE salary ≥ 100
```

```
SELECT * FROM table1
WHERE col3 ≥ x638e54
```

```
Proxy
```

```
Application
```

```
Deterministic encryption
```

```
OPE (order)
```

```
Proxy
```

```
Application
```

```
Deterministic encryption
```

```
OPE (order)
```

```
Proxy
```

```
Application
```

```
Deterministic encryption
```

```
OPE (order)
```

```
Proxy
```

```
Application
```

```
Deterministic encryption
```

```
OPE (order)
```

```
Proxy
```

```
Application
```

```
Deterministic encryption
```

```
OPE (order)
```

```
Proxy
```

```
Application
```

```
Deterministic encryption
```

```
OPE (order)
```

```
Proxy
```

```
Application
```

```
Deterministic encryption
```

```
OPE (order)
```

```
Proxy
```

```
Application
```

```
Deterministic encryption
```

```
OPE (order)
```

```
Proxy
```
Two techniques

1. Use SQL-aware set of encryption schemes

2. Adjust encryption of database based on queries
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>BCLO '09 + our new scheme</td>
<td>order</td>
</tr>
</tbody>
</table>

- e.g., sum
- restricted ILIKE
- e.g., =, !=, IN, COUNT, GROUP, ORDER
- BY, DISTINCT
- e.g., >, <, ORDER
- BY, SORT, MAX, MIN, GREATEST
JOIN

- 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 )
JOIN (cont’d)

- **Security:** do not learn join relations without token

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

Functionality:
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>'CEO'</th>
<th>'worker'</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>ALL?</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>
</table>

Leaks order!
Onions of encryptions

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

- 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';
```

Table 1:

<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>

JOIN

Onion Equality

RND

DET

JOIN

'CEO'

```
col1-OnionEq  | col1-OnionOrder  | col1-OnionSearch  | col2-OnionEq  |
-------------|------------------|-------------------|---------------|
RND          | RND              | SEARCH            | RND           |
RND          | RND              | SEARCH            | RND           |
```
Example (cont’d)

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);
Security guarantees

Queries \(\rightarrow\) encryption schemes \(\rightarrow\) leakage

- Encryption schemes exposed for each column are the most secure enabling queries
- Overall: Reveal only data relations needed for query type, at column granularity
  - equality predicate on a column \(\rightarrow\) DET \(\rightarrow\) repeats
  - aggregation on a column \(\rightarrow\) HOM \(\rightarrow\) nothing
  - no filter on a column \(\rightarrow\) RND \(\rightarrow\) nothing

*common in practice*

Never reveals plaintext
Security threshold

SSN column $\geq$ repeats

Leakage:

- virtually nothing
- repeats
- order
- everything

 Plaintext

Most sensitive columns naturally stay above threshold.
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 encryption schemes
  - e.g., $T1.a + T1.b > T2.c$

Extensions: split queries, precompute columns, use FHE or other 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>

```sql
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
*Most columns at OPE analyzed were less sensitive*
Performance

MySQL:

- Application 1
- Application 2

CryptDB:

- Application 1
- Application 2
  - CryptDB Proxy
  - CryptDB Proxy

DB server throughput

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 CryptDB
TPC-C microbenchmarks

MySQL
CryptDB

Homomorphic addition

No cryptography at the DB server in the steady state!

CryptDB is practical
Demo
Conclusions

CryptDB:

1. The first practical DBMS for running most standard queries on encrypted data
2. Modest overhead and no changes to DBMS

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

Thanks!