CryptDB
Confidentiality for Database Apps

Raluca Ada Popa, Nickolai Zeldovich, Hari Balakrishnan

M.I.T.
http://css.csail.mit.edu/cryptdb/
Problem: Data Breaches and Theft

Sony Says PlayStation Hacker Got Personal Data

By NICK BILTON and BRIAN STELTER
Published: April 26, 2011

Last week, Sony’s online network for the PlayStation suffered a catastrophic failure through a hacking attack. And since then, the roughly 77 million gamers worldwide

Epsilon data breach results in a huge loss of customer data

Epsilon sends more than 40 billion emails a year on behalf of 2,500 brands. Security Week said the breach has affected a number of those brands, including grocery retailer Kroger, TiVo, Marriott Rewards, Ritz-Carlton Rewards, US Bank, JPMorgan Chase, Capital One, Citi, McKinsey & Company, New York & Company, Brookstone, and Walgreens.

TJX Says Theft of Credit Data Involved 45.7 Million Cards

The Framingham, Mass., retailer that owns national chains including TJ Maxx and Marshalls, estimated that a 2007 data breach would cost the company about $25 million. But in the end, the total cost was at least 10 times as high.

Citi Says Many More Customers Had Data Stolen by Hackers

The Citigroup credit card data breach exposed the private financial

2011 was a significant year for data security, with some of the biggest data breaches in our history reported. So far in 2011, we've tracked 535 breaches involving 30.4 million sensitive records. This brings the total reported records breached in the U.S. since 2005 to the alarming number of 543 million.

Privacy Rights Clearinghouse
Why Does Data Leak?

• Adversary exploits software vulnerabilities

• Curious/malicious administrators at hosting servers snoop on data

• Attackers with physical access steal data
Getting Worse, Not Better

• Increasing amounts of sensitive data online

• Trend toward database outsourcing (cloud computing)

• Tension between securing data and computing on data
Ideal Solution: Compute Over Encrypted Data

• Keyword search [Song et al. 2000]
• Tokenization [Navajo, Ciphercloud, etc.]
  – App-level proxy to capture client data, encrypt/decrypt (restricted operations)
• SQL databases
  – On-disk encryption: decrypt/re-encrypt to run queries
  – Encrypted SQL (prior work): Low security, DBMS rewrite, client-side query processing [Hacigumus et al. 2002]
• Fully homomorphic encryption [Gentry 2009, et al.]
  – Fully confidential, but tens of millions of times slower
• CryptDB – a practical system exploring interesting intermediate design point
CryptDB in a Nutshell

- Map SQL to equivalent operations over encrypted data
- Use fine-grained keys; chain these keys to user passwords based on access control
- Data never in plaintext form at the DB server

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

SQL over encrypted data

User 1
User 2
User 3

Application

DB Server
Threat 1: Passive DB Server Attacks

- Map SQL to equivalent operations over encrypted data
- Data never in plaintext form at the DB server

- Under attack
  - Process queries at DBMS, on encrypted data

- Trusted
  - Stores schema, master key
  - No data storage
  - No query execution

- Application
  - plain query
  - decrypted results

- Proxy
  - transformed query
  - encrypted results

- DB Server
  - Encrypted DB
Application

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

Proxy

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

Table 1

<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>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
Applicati
on

SELECT * FROM emp
WHERE salary ≥ 100

Proxy

SELECT * FROM table1
WHERE col3 ≥ x638e54

table1 (emp)
col1/rank | col2/name | col3/salary
--- | --- | ---
x638e54 | x638e54 | x1eab81
x638e54 | x638e54 | x638e54
x922eb4 | x922eb4 | x638e54
x638e54 | x638e54 | x638e54

Deterministic encryption
CryptDB: Two Main Techniques

1. Use SQL-aware encryption schemes

💡 Most SQL uses a limited set of operations

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>+ or *</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>new scheme</td>
<td>join</td>
</tr>
<tr>
<td>OPE</td>
<td>Boldyrev et al.’09</td>
<td>order</td>
</tr>
</tbody>
</table>

- Highest
- Confidentiality

- Restricted LIKE
  - e.g., =, !=, IN, COUNT, GROUP, BY, DISTINCT
- First implementation
How to Encrypt Data Items?

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

<table>
<thead>
<tr>
<th>rank</th>
<th>'CEO'</th>
<th>'VP'</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!
Idea: Onions of Encryption

- Start with the most secure encryption schemes
- Same key for all items in a column for same onion layer
Adjust Encryption

- Strip off layers of the onions
  - Proxy gives keys to server using a SQL UDF (“user-defined function”)  
  - Key corresponds to specific onion layer / column  
  - Proxy remembers onion layer for columns
- Do not put back onion layer (remains fast)
Example

```
SELECT * FROM emp
WHERE rank = 'CEO';
```

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>CEO</td>
<td>RND</td>
<td>SEARCH</td>
<td>RND</td>
</tr>
</tbody>
</table>

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

SELECT * FROM table1 WHERE col1-OnionEq = xda5c0407;

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

SELECT * FROM table1 WHERE col1-OnionEq = xda5c0407;
Features & Limitations

- Insert, update, delete, nested queries, indexes, transactions
- No changes to back-end DBMS server
- Some queries not supported
  - More complex operators
    - E.g., trigonometry, regular expression matching, …
    - Operations that combine incompatible encryption schemes
      - E.g., T1.a + T1.b > T2.c
- Possible extensions: split queries, pre-compute columns, add new encryption schemes
Confidentiality Level

 Queries $\rightarrow$ encryption scheme exposed $\rightarrow$ amount of leakage

 Encryption schemes exposed for each column are the most secure enabling desired queries

- 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*

$\n$Never reveals plaintext$\n$
Application Protection

- User password gives access to data allowed to user by access control policy
- Protects data of logged out users during attack

Arbitrary attacks on any servers 😈

Passive attacks 😈

User 1
User 2
User 3
## Challenge: Data Sharing

<table>
<thead>
<tr>
<th>msg_id</th>
<th>sender</th>
<th>recipient</th>
<th>msg_id</th>
<th>message</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Alice</td>
<td>Bob</td>
<td>5</td>
<td>“this is a secret”</td>
</tr>
</tbody>
</table>

1. **How to enforce access control cryptographically?**
   - Key chains from user passwords

2. **Capture read access policy of application at SQL level?**
   - Annotations

3. **Process queries on encrypted data**
Chaining Keys to User Passwords

- Principals (database rows) have keys
- Data encrypted for principals
- Delegation done with key chains

Alice’s password: hello
Ka = “hello”

Bob’s password: cloud
Kb = “cloud”

E_{Ka}[Ku1]

E_{Kb}[Ku2]

E_{Ku1}[Km5]

E_{Ku2}[Km5]

Ku1

Ku2

msgid 5

Km5

“this is a secret”
Implementing CryptDB

- No change to the DBMS
- Portable: from Postgres to MySQL with 86 lines
- One-key: no change to applications
- Multi-user keys: annotations and login/logout
Evaluation

1. Can CryptDB support real queries & apps?

2. What is the resulting confidentiality?

3. What is the performance overhead?
## Real Queries & Applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Total columns</th>
<th>Encrypted columns</th>
<th># cols not supported</th>
<th>Annotations + lines of code changed</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multi-user keys</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>phpBB</td>
<td>563</td>
<td>23</td>
<td>0</td>
<td>38</td>
</tr>
<tr>
<td>HotCRP</td>
<td>204</td>
<td>22</td>
<td>0</td>
<td>31</td>
</tr>
<tr>
<td>grad-apply</td>
<td>706</td>
<td>103</td>
<td>0</td>
<td>113</td>
</tr>
<tr>
<td><strong>One-key</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TPC-C</td>
<td>92</td>
<td>92</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sql.mit.edu</td>
<td>128,840</td>
<td>128,840</td>
<td>1,094</td>
<td>0</td>
</tr>
</tbody>
</table>

Examples:

- \( \text{SELECT } 1/\text{log(series_no+1.2)} \ldots \)
- \( \ldots \text{WHERE } \sin(\text{latitude} + \text{PI()} \ldots \)
### 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>Multi-user keys</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<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>One-key</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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**

**Columns at OPE were less sensitive**
Performance

MySQL

CryptDB

Application 1

Application 2

Application 1

Application 2

DB server throughput

Plain database

Encrypted database

Latency

Hardware: 2.4 GHz Intel Xeon E5620 – 8 cores, 12 GB RAM

Multiple applications/proxies to ensure database is saturated
TPC-C Benchmark
All Columns Encrypted

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

Throughput loss ~26%
Microbenchmarking TPC-C

No cryptography at the DB server in the steady state!

Encrypted DBMS is practical
Applications of CryptDB

- Untrusted DB administrators in enterprises
- Database services in public clouds
- Multi-user DB-backed applications
- Aggregate analytics over sensitive data
- Outsourcing testing and development
Conclusion

• Confidentiality: enabling SQL over encrypted data
• Two threats: (1) passive DB server attack (2) arbitrary attacks
• Two guarantees:
  – Threat 1: full confidentiality depending on queries
  – Threat 2: confidentiality for logged-out users
• No changes to DBMS server. Modest slowdown.

Papers and source code: http://css.csail.mit.edu/cryptdb/