September 24: CryptDB

Scribe: Peihan Miao

1 Background

1.1 SQL Queries

<table>
<thead>
<tr>
<th>Table: employee</th>
<th>Table: people</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Name</td>
</tr>
<tr>
<td>1</td>
<td>Alice</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
</tr>
</tbody>
</table>

- Sample queries:
  - SELECT * FROM employee WHERE Salary = 200 ORDER BY Name;
  - SELECT * FROM employee, people WHERE employee.ID = people.ID;

- Supported operators:
  - =, !=, +, DISTINCT, MAX, MIN, COUNT, GREATEST, ...

1.2 Traditional DB Security

(a) Permissions: Users have accounts on the DB server. Admin can grant privileges (SELECT, INSERT, ...) of tables to users by GRANT. For example,
GRANT SELECT ON employee TO User1;

(b) Encryption at rest:

![Figure 1: Encryption at rest](image)
2 CryptDB Overview

2.1 Threat Model
Attacker sees all data at server. Note that we only consider “passive/honest-but-curious” attacker who sees all the data, but all data and software are untouched. (In contrast, an “active/malicious” attacker can do anything, including modifying data, issuing queries, etc.)

2.2 Steps Overview
Here are the steps illustrated in Figure 2:

1. App issues a SQL query;
2. Proxy rewrites the query into an encrypted query;
3. DB Server computes the encrypted query on encrypted DB;
4. DB Server returns an encrypted result;
5. Proxy decrypts the result.

Note that the proxy does not store the whole database. Instead, it stores only the master key and the schema.

2.3 Computing on Encrypted Data
The notion of fully homomorphic encryption (FHE), originally called privacy homomorphism, was introduced by Rivest, Adleman, and Dertouzos [RAD78]. In 2009 Gentry proposed the first con-
construction of FHE using ideal lattices[Gen09]. FHE is a concept which combines confidentiality and functionality.

(a) **Semantic Security:** For any PPT adversary \(\mathcal{A}\), and for \(\forall m_0, m_1\), if \(b\) is chosen uniformly at random from \(\{0, 1\}\), and \(c = \text{Enc}(m_b)\), then

\[
\left| \Pr[\mathcal{A}(c) = b] - \frac{1}{2} \right| = \text{negligible}.
\]

\(\mathcal{A}\) cannot guess \(b\) correctly with non-negligible advantage than just guessing randomly. In other words, \(\text{Enc}(m_0)\) and \(\text{Enc}(m_1)\) are *indistinguishable* (i.e., look the same) to \(\mathcal{A}\).

(b) **Functionality:** Computing on encrypted data remains the functionality of computing on the plaintext. More precisely, for any polynomially computable function \(f\), FHE guarantees

\[
f(\text{Enc}(x_1), \cdots, \text{Enc}(x_n)) = \text{Enc}(f(x_1, \cdots, x_n)).
\]

However, a downside of FHE is that the construction so far is very inefficient, more than \(10^6\) slower than computing on the plaintext.

3 **CryptDB Construction**

Although FHE is inefficient, when we focus on the SQL system a key observation is that a small set of operations would suffice: =, +, >, FETCH, UPDATE, JOIN, SEARCH, etc. The idea of CryptDB [PRZB11] is to cover each operation with a fast (specialized) encryption scheme.
3.1 Encryption Schemes

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Function</th>
<th>SQL Operations</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>RND (based on AES)</td>
<td>get/put</td>
<td>SELECT/INSERT/UPDATE/DELETE</td>
<td>semantic security</td>
</tr>
<tr>
<td>HOM (Pallier: $\text{Enc}(x) \cdot \text{Enc}(y) = \text{Enc}(x + y)$)</td>
<td>+</td>
<td>SUM/+</td>
<td>semantic security</td>
</tr>
<tr>
<td>HOM (Elgamal)</td>
<td>$\times$</td>
<td>MULT/$\times$</td>
<td>semantic security</td>
</tr>
<tr>
<td>SEARCH (Song et al.)</td>
<td>match strings in text</td>
<td>LIKE restricted</td>
<td>$\approx$semantic security, with access path leakage</td>
</tr>
<tr>
<td>DET</td>
<td>$=$</td>
<td>DISTINCT/$\neq$/GROUP BY</td>
<td>$\approx$semantic security if values are unique permitting discovery of repeated values, but not the actual value</td>
</tr>
<tr>
<td>JOIN (columns encrypted with different keys, but when JOIN implemented, proxy will give server some capabilities to run)</td>
<td>join</td>
<td>JOIN</td>
<td>same as above</td>
</tr>
<tr>
<td>OPE (order preserving encryption: $x &lt; y \Rightarrow \text{Enc}(x) &lt; \text{Enc}(y)$. 3 properties: unique, high entropy, and sparse)</td>
<td>$&gt;$</td>
<td>ORDER BY/$\leq$/$\geq$/Range queries</td>
<td>order is leaked, but nothing else</td>
</tr>
</tbody>
</table>

3.2 Strawman

A natural idea is to encrypt each column with each scheme. But there are two main drawbacks: (a) it only has the worst security guarantee, and (b) it requires too much space.

3.3 Onions of Encryption

The idea of CryptDB is shown in Figure 3. The encryption schemes are implemented in a leveled fashion, like an onion. The security guarantee becomes stronger as one goes from the inside of an onion to the outside, and the functionality increases as one goes from outside to inside. Note that different keys are used per onion level per column.

3.3.1 Example

Query: SELECT SSN FROM employee WHERE Timestamp $\geq$ 200;

In order to implement this query, the proxy will first transform it into two queries (one to decrypt
the onion Ord, one to query over the order preserving encryption), and send to DB server:

\[
\text{UPDATE } \text{OnionOrd} \leftarrow \text{Dec(key, OnionOrd)};
\]

\[
\text{SELECT field3 FROM employee WHERE field5 } \geq \text{Enc(200)};
\]

The DB server will first decrypt \text{OnionOrd}, and then runs on the encrypted data same as on unencrypted data.

**References**

