September 24: CryptDB

Scribe: Peihan Miao

1 Background

1.1 SQL Queries

Table: employee				
ID	Name	SSN	Salary	Timestamp
1	Alice	23. . .	200	120
2	Bob	88. . .	180	300
. . .	. . .	. . .	. . .	. . .

Table: people	
ID	Personal
2	. . .
5	. . .
. . .	. . .

- 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:**

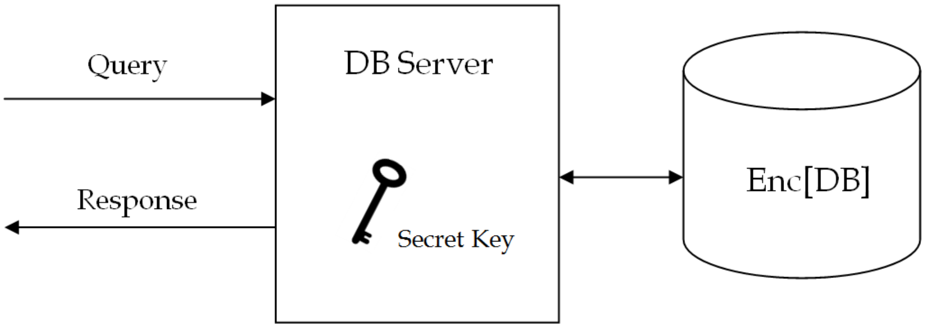

Figure 1: Encryption at rest

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.)

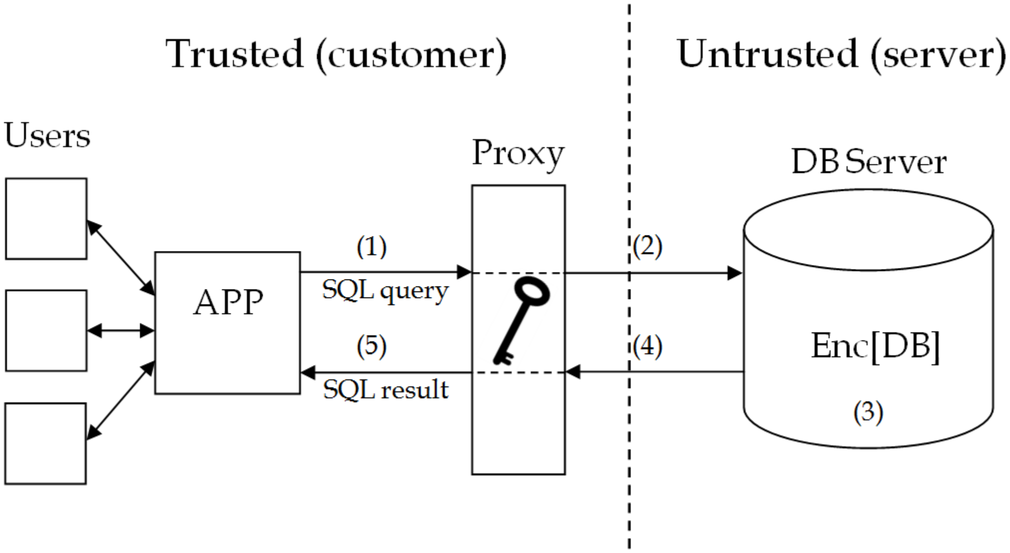

Figure 2: CryptDB steps overview

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 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 $c = \mathsf{Enc}(m_b)$, then

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

$\mathcal{A}$ cannot guess b correctly with non-negligible advantage than just guessing randomly. In other words, $\mathsf{Enc}(m_0)$ and $\mathsf{Enc}(m_1)$ 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

$$f\left(\mathsf{Enc}(x_1), \cdots, \mathsf{Enc}(x_n)\right) = \mathsf{Enc}\left(f(x_1, \cdots, x_n)\right).$$

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

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

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

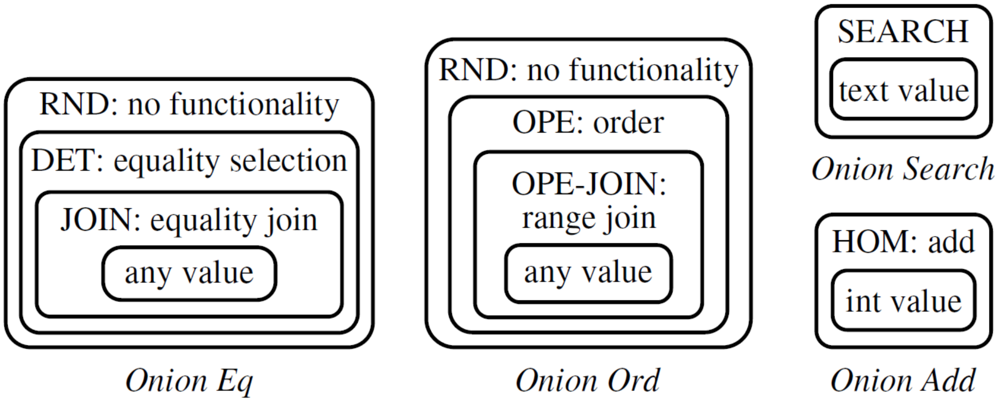

Figure 3: Onion encryption layers

the onion Ord, one to query over the order preserving encryption), and send to DB server:
$\mathsf{UPDATE}\ \mathit{OnionOrd} \leftarrow \mathsf{Dec}(\text{key}, \mathit{OnionOrd});$
SELECT field3 FROM employee WHERE filed5 $\geq \mathsf{Enc}(200);$
The DB server will first decrypt *OnionOrd*, and then runs on the encrypted data same as on unencrypted data.

References

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, volume 9, pages 169–178, 2009.

[PRZB11] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100. ACM, 2011.

[RAD78] Ronald L Rivest, Len Adleman, and Michael L Dertouzos. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.