Abstract

As more organizations and companies move to the cloud, the need to protect against data breaches becomes ever more important. Although previous systems using encrypted databases have been developed, the intricacies of their design lead to further attacks that compromise the privacy of the stored data.

We propose PartDB, a system to protect against such attacks. PartDB partitions data in a reconstructable manner to prevent an attacker from recovering individual rows of a database, while still allowing an application to treat the system as a single whole. We investigate multiple threat models, involving various degrees of compromise of the database system, and provide defenses countering each threat.

1 Introduction

As the amount of sensitive data that is stored by companies and governments increases, the need to secure it from possible attackers increases. Unfortunately, data breaches are becoming the norm, and the repercussions are becoming ever more severe. Just in 2015, several prominent data breaches have appeared in the news, including the Anthem health record breach, the exposure of Ashley Madison user accounts, and even the US Office of Personnel Management breach in which 21.5 million persons’ government background checks were compromised.

Companies concerned about such threats may not have the resources to adequately protect their data on local systems. However, outsourcing data to third parties requires an extension of the company’s trust. A compromise of these third party servers is a compromise of the company’s sensitive data. This has stimulated research into ways of encrypting sensitive data, so that data can be outsourced to third parties without outsourcing trust as well.

Encrypting databases through property-preserving techniques such as those offered by CryptDB [3] offers protection against malicious database administrators and introduces relatively little overhead by using property-preserving encryption (PPE) schemes. Ideally, the compromise of such a system would reveal very little about the sensitive data on it. However recent work [2] has shown that given publicly-available auxiliary information about the dataset, this curious database administrator can recover the plaintext from deterministic (DET) and order-preserving encrypted (OPE) columns. The attack leveraged the fact that by their nature, DTE and OPE leak some amount of relational information.

In this work, we address inference attacks by partitioning a database into multiple places. Naveed et al. [2] rely on the data being coalesced together in order to infer information. Specifically, they examine two types of attacks: individual attacks and aggregate attacks. In an individual attack, the adversary aims to recover information about a particular row, such as a person in a medical record database. In an aggregate attack, the adversary aims to recover information about a particular column, such as the number of patients that are uninsured. Both of these attacks can be thwarted by teasing apart the items of interest.
Thus, a solution based on vertical partitioning would defend against individual attacks, while one based on horizontal partitioning (or sharding) would defend against aggregate attacks. We argue that individual attacks are much more concerning to users, and so we focus our work on this area. However the methods we introduce can easily be applied to horizontal partitioning in order to thwart aggregate attacks.

2 Background

CryptDB uses PPE to construct and manage an encrypted database (EDB) supporting a subset of SQL. Solutions based on PPE by nature leak some amount of relational information about the ciphertext: DET reveals which elements are equivalent to others, and OPE reveals the ordering of elements. CryptDB introduces an “onion” model of encryption where each column is encrypted multiple times under increasing “layers” of encryption. For example, a column containing ages of patients would be encrypted first under OPE, then DET, then randomized encryption. If a column never needs to be checked for ordering, then this column would never be “unpeeled” to an OPE layer. If, however, a column needs ordering information (e.g. ages of patients), then it would be unpeeled to an OPE layer. This way the ages still cannot be deduced from the ciphertext as is, but the order of ages within the ciphertext remains correct.

Naveed et al. [2] leveraged CryptDB’s PPE properties to perform an inference attack against electronic medical records by using publicly available information. They attacked the EDB in steady state, meaning a state where its onion layers are peeled down to the lowest layer needed to support an application executing queries on it. They did not need any information about the queries themselves; just a snapshot of the EDB on disk. While their attack was against EDBs using PPE schemes, it built on the inference attacks proposed by Islam et al. [1] against searchable symmetric encryption (SSE) by using auxiliary information and knowledge of client queries to recover information about future queries.

3 Threat Models

Naveed et al. refer to two kinds of inference attacks: individual attacks and aggregate attacks. In our model, we only address the former.

3.1 Basic Attack

The threat model used by Naveed et al. assume an attacker who can observe a “snapshot” of the EDB. Realistically, this can be achieved by an attacker gaining access to a VM snapshot of a cloud server. We assume the database is in a steady state, and all columns have been “peeled” to their lowest needed encryption layer.

3.2 Query Sniffing

We also consider a stronger case in which an attacker is able to observe all packets going in or out of the server. Notably, the attacker would have access to all query/response pairs at the server. An example of this attack model is the standard “honest-but-curious” scenario of a server administrator with full access.

4 Defense Models

In this section, we describe how to defend against individual inference attacks.

4.1 Basic Attack

For the basic adversary scenario, there is only one single data server and attacker cannot see the queries generated by the proxy. Without encryption, if an adversary gains complete control of the server, all data is leaked. Encryption through PPE techniques can defend against such database theft. However, this still will not protect against inference attacks.

The straightforward defense is to split the table vertically into several partial subtables. For each subtable, all rows are re-ordered, and a lookup table at the proxy is used to recover the original ordering. In this way, even if an adversary can read all the subtables, the adversary cannot relate one row of a subtable to another subtable’s row. This restricts the information an attacker can infer.
4.2 Query Sniffing

If an attacker can observe all queries and responses at the server, the attacker can recover the correspondence between rows of subtables with high probability given enough time. For any database query on the original table, the proxy will generate several new queries to each of the individual subtables. The attacker would be able to detect all the rows these queries have touched on each subtable, and will then know the relation of these rows and deduce the whole row in the original table. Every query will leak some information of the original table, continuously monitoring all the queries, the attacker would eventually rebuild the entire original table.

One seemingly obvious mitigation for this situation is to insert dummy queries in each query operation, obfuscating the original query. However, with some additional statistical processing, a determined attacker would still be able to recover the original rows with enough patience.

To tackle this issue, we propose another solution: distribute the subtables among different servers. If we assume that attacker can only compromise one server, then the data is safe. We argue that this security model is realistic provided the client allocates different servers in different cloud providers. In this case, it would be highly unlikely for a single attacker to compromise machines from multiple data center operators. The proxy now translates queries into multiple queries to different servers. By separating the data, there is no way for an attacker at one server to be able to recover any information stored in any other server.

Note that in this solution, we have no longer have a need to reorder rows. In our assumption, an attacker would not have access to data other servers, so the order of rows in each server no longer matters. We can keep the ordering the same as the original table, making the proxy logic much simpler.

4.3 Multi-Server Compromise

In the previous solution, we assume an attacker can only compromise at most one of the data servers. In the case an attacker compromises all the servers, we do not have a good solution. Because the attacker can observe all servers, our model becomes equivalent to the single server model with query sniffing. As explained above, even with re-ordering rows in each subtable, given enough time and with enough statistical analysis, an attacker would be able to piece together all the rows of the original table with high probability.

5 Design

We present two implementations of our system with variations in complexity. The simpler system, termed PartDB$A$, protects against an attacker that has fully compromised one data server of many. The second system, termed PartDB$B$, can protect against an attacker who has compromised the contents of the database but cannot intercept queries. Both systems are implemented as augmentations to the CryptDB proxy, and thus sit between the application and the database(s).

5.1 Row Lookup

5.1.1 Ordered Rows

In the model where the database user employs multiple servers and the attacker can only gain access to one, we do not need to scramble the rows within each server. Under this model, the attacker does not have access to any more than one column of the original table, and thus cannot perform any correlation attacks with respect to other columns. Thus the added complexity of re-ordering and reassembling rows provides no additional security guarantees in this model.

As such, our system becomes much simpler. We have no need for a correlation table or cryptographic correlation. We can simply use the original primary key as a shared key among each column shard. If there is concern about correlation attacks against the primary key, we can instead key each column with a hash of the primary key, an ascending integer, or any other unique identifier, and treat the original primary key column as a regular data column.

In this system, we first assign a new row ID ($R$) to each row, which functions as a redundant primary key. Given the original table $T$ containing $n$ columns $C_1, \ldots, C_n$, and assuming we have at least $n$ databases $D_1, \ldots, D_n$, we perform the following for each $1 \leq i \leq n$: In database $D_i$ we create table
<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>HasCancer</th>
<th>LastVisit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>25</td>
<td>N</td>
<td>December 10, 2015</td>
</tr>
<tr>
<td>Bob</td>
<td>27</td>
<td>N</td>
<td>September 1, 2015</td>
</tr>
<tr>
<td>Carol</td>
<td>55</td>
<td>Y</td>
<td>March 19, 2016</td>
</tr>
</tbody>
</table>

Table 1: Original table

<table>
<thead>
<tr>
<th>R</th>
<th>Name</th>
<th>R</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
<td>2</td>
<td>27</td>
</tr>
<tr>
<td>3</td>
<td>Carol</td>
<td>3</td>
<td>55</td>
</tr>
</tbody>
</table>

(a) T1 (b) T2

<table>
<thead>
<tr>
<th>R</th>
<th>HasCancer</th>
<th>R</th>
<th>LastVisit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>N</td>
<td>1</td>
<td>December 10, 2015</td>
</tr>
<tr>
<td>2</td>
<td>N</td>
<td>2</td>
<td>September 1, 2015</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>3</td>
<td>March 19, 2016</td>
</tr>
</tbody>
</table>

(c) T3 (d) T4

Table 2: Tables after splitting

5.1.2 Re-ordered Rows

Using a correlation table

If the adversary manages to gain access to a copy of the stored database of all servers (or just a single database in a single database server setup), PartDB_A is insufficient as the relation between columns is clear from the row IDs, and the adversary can easily construct the original database. In addressing this attack scenario we assume that the adversary can’t see both the queries and returned answers, but has access to the entire stored database of all servers, whether it is on a single or multiple servers. This situation can arise if the adversary manages to compromise either the backups of a live system, or if the adversary gains access to the underlying storage system.

Instead of keeping the same order of the rows in the databases by using the same row ID for related rows, we scramble the relations by generating new IDs for every value in a row. For a row l composed of columns C_i and C_j in the original database, R_i l ≠ R_j l in the new database. To make this scheme work, the proxy will now have to keep track of the relations between the different row IDs.

Using Encryption

Instead of requiring a correlation table available for the proxy to use for retrieving the relations, a better alternative is to use a block cipher to compute the relations on the proxy. The proxy only has to store the symmetric key used for encryption and decryption, and thus avoids storing a large correlation table. Let \( E: \{0, 1\}^k \times \{0, 1\}^n \rightarrow \{0, 1\}^n \) be a block cipher.¹

To split a table having the columns \( C_1, \ldots, C_n \), into n different subtables as in PartDB_A, we generate a new row ID (R) for each row to be used as the primary key. R will be repeatedly encrypted, with each round of encryption serving as the primary key of the next subtable. Specifically, the scheme of the subtable storing column \( C_n \) will be \( E_{n-1}^k(R), C_n \) where \( E_n^k(R) \) denotes n encryptions of R.

With this system, the proxy no longer needs a correlation table to convert row IDs. For example, to compute the corresponding primary key in table \( C_1 \) when given the primary key from table containing \( C_3 \), the proxy need only apply the decryption algorithm twice.

More specifically, to relate the primary key from table \( i (R_i) \) with the primary key in table \( j (R_j) \), if \( i > j \) the proxy would compute \( D_{i-1}^k(R_i) \) to get \( R_j \), and if \( i < j \) the proxy would compute \( E_{j-1}^k(R_i) \) to get \( R_j \).

5.2 Column Splitting

It is not necessary to split every column of a table into its own subtable. However, there is a tradeoff between efficiency and security in how to split each table. Increasing partitioning leads to better security, in that an attacker must expend more resources to recover the contents of the table. However, having

¹We recommend using Blowfish [4] because of its short block size of 64 bits.
more partitions also increases the number of resulting queries and increases processing overhead.

We suggest that an application split its tables based on levels of sensitivity of the data. For example, columns that contain information that is public or otherwise easily accessible can be kept in the same subtable, as they are of little value to an attacker. On the other hand, we recommend columns with sensitive individual identifiers, such as social security numbers, be kept on their own, as otherwise a compromise would allow an attacker to relate this identity information with other information in the same subtable.

5.3 Query Conversions

We are given an original table \( T \) containing \( n \) columns \( C_1^A, \ldots, C_n^A \). As above, we assume at least \( n \) databases \( D_1, \ldots, D_n \) with each \( D_i \) containing a table \( T_i^A \) consisting of columns \( R_i^A \) and \( C_i^A \).

5.3.1 Insert

INSERT queries are simply converted to a set of inserts for each subtable, ensuring that the new entries share corresponding identifiers \( R \).

The number of additional queries generated scales linearly with the number of subtables. For any given system, the number of subtables does not fluctuate, and so the query complexity can be considered constant time.

5.3.2 Select/Update

For a simple SELECT query of the form “SELECT * FROM \( T \) WHERE \( C_i^A = X \)”, we perform the following:

1. SELECT \( R_i^A, C_i^A \) FROM \( T_i^A \) WHERE \( C_i^A = X \)
2. For each \( j \neq i \), SELECT \( R_j^A, C_j^A \) FROM \( T_j^A \) WHERE \( R_j^A = R_i^A \)
3. JOIN each \( C_i^A \) on \( R \)

For a SELECT query with multiple WHERE clauses, we repeat step 1 for each clause and take the resulting set intersection (or union) of \( R \) values before continuing to the later steps.

For a SELECT query comparing two columns (e.g. WHERE \( C_i^A = C_j^A \)) in which the two columns are stored in different subtables, the PartDB proxy must first download the full set of each column and perform the comparison locally. We argue that this is no worse than CryptDB’s performance on “secure” columns that can only be handled by the CryptDB proxy.

UPDATE queries proceed just as SELECT queries, with an additional update procedure performed on each resulting entry as necessary.

The number of additional queries generated scales linearly with the number of subtables. For any given system, the number of subtables does not fluctuate, and so the query complexity can be considered constant time.

5.3.3 Join

The query implementation for JOIN proceeds similarly to SELECT. When (left-)joining tables \( T_A \) and \( T_B \) on column \( C_k \), where \( C_k \) appears in \( T_A \) and is the primary key for \( T_B \), we perform the following:

1. SELECT \( R_k^A, C_k^A \) FROM \( T_k^A \)
2. For each \( i \neq k, i \leq n \), SELECT \( R_i^A, C_i^A \) FROM \( T_i^A \) WHERE \( R_i^A = R_k^A \)
3. SELECT \( R_k^B, C_k^B \) FROM \( T_k^B \) WHERE \( C_k^B = C_k^A \)
4. For each \( j \neq k, j \leq m \), SELECT \( R_j^B, C_j^B \) FROM \( T_j^B \) WHERE \( R_j^B = R_k^B \)
5. JOIN each \( C_i^A \) on \( R \)
6. JOIN each \( C_j^B \) on \( R \)
7. JOIN \( C^A, C^B \) on \( C_k \)

Note that the final three steps happen locally on the PartDB machine.

Other joins can be constructed simply by modifying steps 1 and 3. Additionally, WHERE clauses are easily supported in a similar fashion to the construction of SELECT queries above.

The number of additional queries generated scales linearly both with the number of subtables per table and with the number of tables being joined. For any given system, the number of subtables per table does not fluctuate, and so the query complexity can be considered linear in the number of tables joined.
5.3.4 Overhead Optimization

A common pattern in the SELECT and JOIN query conversions is the proxy retrieving a set of row IDs from one subtable, then immediately retrieving the corresponding rows from another subtable. In the PartDB_A system, where row IDs between subtables correspond exactly and each subtable is in a different server, we can reduce some of the communication overhead at the proxy by instructing the first server to forward the list of row IDs to the other servers. The other servers would then send the relevant rows directly to the proxy.

We argue that this optimization leaks no additional information, as the row IDs are simply sent between other servers directly instead of being routed first through the proxy. Since query results are still sent directly to the proxy, and not sent to other servers, an attacker on any single server still learns no additional information about data stored in other servers.

6 Prior Work

There exists a host of previous work in horizontal partitioning, specifically in the context of distributed databases. Horizontal partitioning allows a large dataset to be spread across multiple servers by distributing groups of rows to different servers. There exists little work in vertical partitioning because it is seen as a pre-optimization in terms of efficiency. To our knowledge, there is no prior work examining partitioning for security or privacy reasons.

7 Conclusion

We proposed PartDB, a system to thwart inference attacks on encrypted databases. By splitting tables vertically and re-ordering or physically separating the columns, we prevent attackers seeking to recover information contained in any given row. We consider multiple threat models, involving various degrees of compromise of the database system, and construct variations of our PartDB system to protect against each threat model.

References