Integrity-based Attacks for Encrypted Databases and Implications

Arvind Arasu
arvinda@microsoft.com

Raghav Kaushik
skaushi@microsoft.com

Donald Kosmann
donaldk@microsoft.com

Ravi Ramamurthy
ravirama@microsoft.com

ABSTRACT

Inference attacks on property-preserving encrypted databases (e.g., CryptDB) have been previously studied. These demonstrate how in certain scenarios one can recover plain text from databases that provide columnar encryption by using auxiliary information such as column statistics. Newer generation of encrypted databases are now being built using secure enclave technology. In this paper, we first show how the current generation of encrypted databases are robust against these previously published attacks. However, we identify two broad patterns that we identify as query integrity attacks and data integrity attacks that can be used to construct attacks that are similar in scope for a variety of encrypted databases built using enclaves. We believe this paper initiates an important discussion about the need for integrity protection for future encrypted databases.

1. INTRODUCTION

Encrypted Database Systems (EDBs) are becoming a necessity for cloud database offerings. There is an obvious need to protect PII (personally identifiable information) using encryption in the cloud where cloud administrators can freely access the database. In order to support querying on PII information, EDBs need to support the ability to run queries (such as equality, LIKE predicates) directly on encrypted data.

EDBs were first built using property preserving encryption (PPE) schemes [17, 18, 15] which are cryptographic schemes that support computation directly on encrypted data. They offered a simple security model – no encryption keys are required to be kept in the cloud. Research prototypes like CryptDB [15] supported operations such as equality and range predicates. However, these offerings were incomplete since customers needed to run more complex queries (such as LIKE predicates) on encrypted PII data and complex operations such as pattern matching are not supported by property preserving encryption schemes.

In order to expand the class of supported queries, new EDB architectures were explored using trusted hardware(e.g., Cipherbase [1], TrustedDB [4]). Trusted hardware [8] is used to store the keys securely in the cloud and perform computations on them. Enclaves provide the important guarantee that the encryption keys and any plaintext data are not accessible outside the trusted hardware. Using trusted hardware, EDBs can now fully address the workload requirements of PII.

EDBs have now started gaining commercial traction. Recently, SAP has prototyped SEEED [12] using HANA and SQL Server has shipped the Always Encrypted feature [10] using secure enclaves that supports advanced features including indexing (i.e., encrypted B-Trees) and LIKE predicates. This underscores the point that EDBs significantly improve the security bar for handling PII data and this technology has truly arrived.

However, there is an inherent tradeoff between performance and security and all forms of EDBs have some information leakage. As queries are executed on encrypted data, there is some information leakage about the computation performed. Given enough background information, an adversary can exploit this leakage to deduce plaintext. Such *inference attacks* have been studied in the context of PPE EDBs [14] such as CryptDB. These studies show that plaintext can be recovered if an adversary has sufficient background information such as single column statistics. However, prior work has not analyzed the applicability of these inference attacks to encrypted EDBs built using enclaves. In this paper we examine the following questions:
(1) Are the PPE attacks still applicable in their original form to enclave-enabled EDBs?
(2) If not, are there other attacks that are equivalent in scope?

We first present basic preliminaries on different EDBs and their leakage functions and discuss the adversary model (Section 2). We then recap PPE attacks and discuss why these attacks which only leverage the static database contents do not directly translate (Section 3) to enclave-enabled EDBs. In Section 4, we present two new form of attack vectors which enable inferences that are equivalent in scope to the original PPE attacks. Interestingly, these attacks mainly exploit the fact that the current generation of commercial EDBs *do not provide any integrity guarantees on the database* — as a result, an adversary can hence modify any queries (e.g., by rewriting any installed stored procedures) as well as tamper with the plaintext database by adding some fake data in order to infer correlations on the encrypted database. We believe this paper makes the key contribu-

11th Annual Conference on Innovative Data Systems Research (CIDR '21) January 10-13, 2021, Chaminade, USA.

tion of highlighting the importance of integrity protection in EDBs. While current EDBs (in both commercial offerings and in the research literature) do not consistently adopt integrity protection as a design goal, this paper illustrates conceptual attacks that can result due to this omission — we wish to underscore the point that that *integrity-based attacks form a broad attack vector* for a variety of EDBs built using enclaves.

Given EDBs are here to stay, the design of future generations of these systems will be largely influenced by understanding any new attack vectors and building safeguards for them — this paper highlights the importance of integrity protection for future commercial offerings of EDBs.

2. PRELIMINARIES: EDBS

At a high level, all EDBs support the following API. EDBs support columnar encryption — i.e., each individual tuple in a column is encrypted with a column encryption key (CEK) created by the user. Any plaintext SQL query that needs to evaluate a predicate on the encrypted column will be *re-written* to run on the encrypted column. For instance, if the original query has a plaintext constant, this would be encrypted by using the corresponding CEK of the column. This is done by the SQL client driver(e.g., ADO.net or JDBC) [10] or a custom middleware layer [15].

We note that current EDBs do not support the full generality of SQL. The class of queries supported include:
(1) Equality queries with constants encrypted.
(2) Range queries with constants encrypted.
(3) LIKE predicates with the LIKE predicate encrypted.

Different versions of EDBs support different subset of this workloads. For instance, EDBs based on property preserving encryption do not support LIKE predicates. Inference attacks previously studied in the context of PPE EDBs [14] focus on equality and range queries. We focus on those workloads but also include LIKE predicates since this has been recently shipped in a commercial offering [10].

We provide a brief overview of property-preserving EDBs and enclave-based EDBs and focus on the information leakage profile of both the static database contents and query execution.

2.1 Property-Preserving EDBs

Property preserving encryption (PPE) schemes [17, 18, 15] allow computation directly on encrypted data. They support columnar encryption where a user can choose to encrypt a particular column with an appropriate property preserving encryption schemes. Once data is encrypted, there is a middleware layer that can suitably rewrite SQL queries to run directly on the encrypted data (see CryptDB [15] for more details). The encryption schemes used include:
(1) DET - deterministic encryption that supports equality operations.
(2) OPE - Order preserving encryption that supports range predicates.

Information Leakage Profile: An important point to note is that PPE schemes do not store the individual tuples using ”strong” encryption that provides indistinguishability for the plaintext value (such as AES-CBC). For instance, both DET and OPE schemes discussed in [15] will use the same ciphertext value for the same plaintext value. Thus, the information leakage for PPE based schemes are as follows:
(1) DET encryption: The static database leaks distinct values in the tuples.
(2) OPE encryption: The static database leaks distinct values and ordering among the tuples.

Example 1: Static DB Leakage: Figure 1 shows a Table with eight tuples in plaintext as well as the corresponding table when the attribute is encrypted using DET. The ciphertext leaks distinct values and the frequency distribution of these values. However, it does not leak ordering, i.e., it is not possible to infer if the ciphertext xffb2 is greater or less than x22a5. However, if the table is stored using OPE, the database would maintain an additional index (e.g.,a balanced search tree in [15]) that encodes the ordering between the ciphertexts. Thus, in this example if the table were stored in OPE, the database would additionally leak the fact that the ciphertexts are ordered as: x22a5 <= xffb2 <= xbb33.

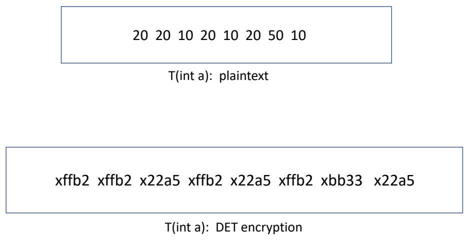

Figure 1: Static Leakage in PPE EDBs (ciphertexts are not full-length).

2.2 Enclave-based EDBs

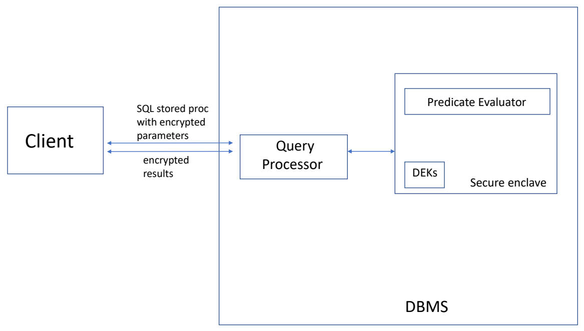

Figure 2: Fine-grained Enclave-based EDBs.

Enclaves [7] are hardware based solutions that support secure computation even when all the privileged software (kernel, hypervisor, etc) is potentially malicious. Cloud providers have started offering servers with enclaves in order to support secure computation in the cloud. There have been a variety of EDBs developed using enclaves (e.g., [1, 4, 10, 16]) — they typically support a larger class of queries on encrypted data (when compared to PPE based approaches). In certain cases, EDBs with enclaves can also provide the same functionality as PPE (e.g., OPE) with better performance.

There have been many designs explored in this space based on what components of the query engine actually run in the enclave. In this paper we focus on fine-grained architectures that use the enclave to store data encryption keys (DEKs) and run only select computations like predicate evaluation as in the Cipherbase project [1]. We focus on this architecture primarily because these have now gained commercial traction and is now shipping as part of the commercial SQL Always Encrypted [10] product.

At a high level, the workflow is as follows: SQL stored procedures are invoked with encrypted constants which are transparently encrypted by the client driver. Query processing works as usual using the iterator model. Any operator (e.g., a filter with a LIKE predicate) which needs to do any computation such as predicate evaluation on encrypted data would route the tuple to the enclave where the data is decrypted and the predicate is evaluated and the boolean result is returned in plaintext. The workflow is shown in Figure 2. A key feature of fine-grained architectures is the fact that the rest of query evaluation (including buffer management, query iterators, query parsing etc.) is mostly unmodified since they do not need access to plaintext values.

We note there are other approaches that use a"coarse-grained" architecture where larger chunks of the database including query operators, parts of the query engine (e.g., [4, 9, 16]) are run in the enclave. While we primarily focus on fine-grained architectures, we also comment in Section 5 where the discussed attacks are applicable to other EDBs built using enclaves. A thorough survey of all attacks possible for EDBs built using enclaves is beyond the scope of this paper — we primarily wish to underscore the broad pattern of integrity-based attacks that are applicable for a variety of EDBs built using enclaves.

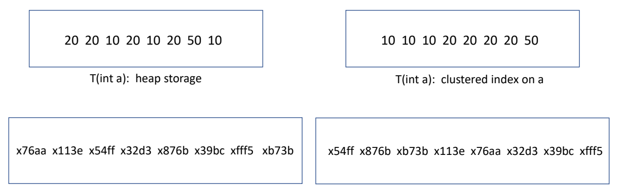
20 20 10 20 10 20 50 10
T(int a): heap storage
10 10 10 20 20 20 20 50
T(int a): clustered index on a
x76aa x113e x54ff x32d3 x876b x39bc xfff5 xb73b
x54ff x876b xb73b x113e x76aa x32d3 x39bc xfff5

Figure 3: Static Leakage in Enclave-enabled EDBs (ciphertexts are not full-length).

Information Leakage Profile: A key difference from PPE based EDBs is that data is stored encrypted at a cell level using encryption schemes that are IND-CPA secure (e.g., AES-CBC mode)[1]. IND-CPA secure schemes guarantee that the same plaintext value will map to different ciphertext values. Thus, the system ensures that distinct values are not leaked by the static database. However, the static database can leak ordering information between ciphertexts in case there is an index on that column as the following example illustrates.

[1]Enclave-based EDBs also support PPE schemes like DET. We ignore this feature while constrasting them from PPE based EDBs.

Example 2: Static DB Leakage: Figure 3 shows the same Table illustrated in Figure 1 for enclave-enabled EDBs. We distinguish two cases — one where the table is stored as a heap and another where there is a clustered index on the column a. In the first case, each tuple being encrypted with a IND-CPA secure scheme ensures that even the same plaintext values map to different ciphertext values — distinct value information is not leaked as in Example 1. Since the data is stored in a heap, in addition, no ordering information can be inferred. Thus, in this case, the system provides indistinguishability for the data at rest. However, in the case when there are indexes (e.g., a clustered index in Figure 3), the ordering of tuples leaks the ordering between ciphertexts i.e., an adversary can infer that x54ff <= x876b and x876b <= xb73b and so on.

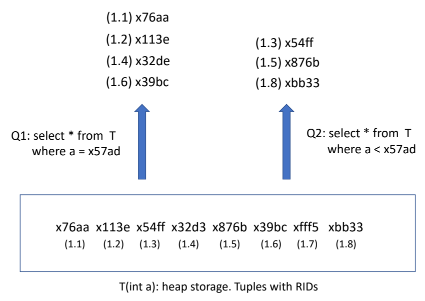
(1.1) x76aa
(1.2) x113e
(1.4) x32de
(1.6) x39bc
(1.3) x54ff
(1.5) x876b
(1.8) xbb33
Q1: select * from T where a = x57ad
Q2: select * from T where a < x57ad
x76aa x113e x54ff x32d3 x876b x39bc xfff5 xbb33
(1.1) (1.2) (1.3) (1.4) (1.5) (1.6) (1.7) (1.8)
T(int a): heap storage. Tuples with RIDs

Figure 4: Query Execution based Leakage in EDBs (ciphertexts are not full-length).

In addition to the static database contents, an adversary can use the leakage profile of query execution which is as follows:
(1) Equality queries: leaks the fact that the output tuples are equal to each other.
(2) Range queries: leaks the fact that the output tuples are in some predicate range (which is encrypted).
(3) LIKE queries: leaks the fact that the output tuples all satisfy some encrypted regex predicate.

Example 3: Query execution based Leakage: Consider the case where the table is stored as a heap in Figure 3. As we mentioned earlier, in this case, the system guarantees indistinguishability for the data at rest. Figure 4 illustrates how an adversary can learn additional relationships between ciphertexts (that is not revealed by the static database) by monitoring query execution. For instance, Query 1 selects all tuples that are equal to x57ad and Query 2 selects all tuples that are less than x57ad. By running both queries, an admin can now infer all ciphertexts in Table T that are equal to the value x76aa in the database and those that are less than x76aa. We note that this information can be tracked even if the output tuples are re-encrypted in the enclave. This is because most query execution proceeds using the iterator model [11] and as long as an admin can track which input tuples passed the filter predicate (e.g., by tracking RIDs of rows), he can derive the same relationships. Figure 4 illustrates this by making explicit the RIDs of the tuples in the database as well as the results.

2.3 Adversary Model

We now describe the adversary model for a database administrator used in this paper. First, we note that an admin *does not have access to encryption keys.* Thus, only database owners who own the encryption keys have the ability to run arbitrary plain-text queries by encrypting the query constants. Of course, the lack of ability to run arbitrary queries does not interfere with several regular tasks of an administrator (e.g., data backup, capacity planning).

We assume an adversary is a database administrator who has complete access to the database modulo the encryption keys. Unlike a *passive adversary* who only monitors query execution, we assume an active adversary can obtain information by trying to expand the information leakage profile of the set of queries that are actually run by legitimate users. He can do this in a variety of ways:
1) He can execute modified versions of a client's query (e.g., change the predicate in a query from equality to a comparison predicate or drop a predicate in the query).
2) While he cannot create any new encrypted constants (for a plaintext of his choice), he can modify any encrypted constant in a query by re-using encrypted constants from previous queries or by using constants from the encrypted database. For instance in Figure 4, if a legitimate user ran Query 1, an admin can run Query 2 and figure out ordering information which was not leaked by Query 1.
3) In the case when he has access to the plaintext database before encryption, he can potentially modify the plaintext database *before encryption* (e.g., by adding some fake rows).

In addition, we assume than an adversary has some background information on the database — we use the same assumptions in [14] which outlined inference attacks for PPE based EDBs. We assume *single-column statistics* on columns from the plaintext database (i.e., the adversary has no information on any joint distributions of the columns). Finally, we assume an adversary has mounted a successful attack if he can decode the plaintext completely for any fraction of rows for any encrypted column in the static database (as in [14]).

We first consider the special attack vector of *static database attacks* (Section 3) where an admin has access to no information except the static database (i.e., no ability to run any queries). We discuss this case because: (1) This is an interesting attack vector where an adversary has potentially stolen the database file on disk and tries to mount an offline attack. (2) Previously published attacks on PPE database [14] focus on this attack vector.

3. STATIC DB ATTACKS

3.1 Overview

We briefly review the attacks proposed in [14] for property-preserving encryption based EDBs. Previously proposed attacks for PPE based EDBs [5, 14] are static database attacks that only need access to the database[2]. This essentially models a "disk-stealing" adversary who steals the disk contents of the database and mounts an offline attack. The attacks assume the following single column statistics:
(1) Frequency histogram of plaintext values
(2) Rank of value in domain of column values.

[2]if the disk is encrypted then the admin needs to run a select * query that selects all rows and run the attack on the results

The key insight in the paper is that both OPE, DET implementations leak distinct values in the static database from which it is easy to compute frequency histograms and ranks on ciphertext. The key technical challenge remaining is how to match the ciphertexts with the plaintext values. We refer the reader to [14] for more details.
The following simple example illustrates an attack for a DET column. Assume a hospital scenario with a table storing Patient information with the following schema:
$Patients(id, name, disease, riskfactor, age)$. Assume all the attributes are encrypted and the encryption scheme is: (1) DET for id, $name$ and $diseasename$ (2) OPE for $riskfactor$ and age.

Example 4: DET Encryption: Assume the adversary has access to statistics in the disease column which has the following distribution: 50% have Flu, 30% have Pneumonia, 20% have COVID. The disease name is stored using DET encryption which preserves equality. By simply aggregating the rows in the database that have the same ciphertext and matching the ranks in the distributions, the adversary can figure out the corresponding disease for each ciphertext.

3.2 Applicability to Enclave-based EDBs

Claim: PPE attacks that rely on the static DB [14] are not applicable to enclave-enabled EDBs which store data strongly-encrypted.

An enclave-based EDB which stores data strongly encrypted only leaks ordering (in case there is an index on the column) but does not leak any duplicate information (as explained in Figure 3). Any frequency based attacks needs information about duplicate values as input— since enclave-enabled EDBs use non-deterministic encryption, duplicates cannot be detected from the static database. Thus, the static database based attacks (where an admin has access to no information except the static database) in [14] do not work for the case of enclave-based EDBs.

4. ATTACK VECTORS BEYOND STATIC DB

While the static database attacks in [14] are not directly applicable to enclave-enabled EDBs, the key question that arises is are there other attacks that are equivalent in scope?

In this section, we examine other potential attack vectors. First, we look at exploiting query results in Section 4.1. The act of query execution leaks information — if an adversary can monitor the results of queries, can he construct attacks similar in scope to the PPE attacks on the static database?

In addition, we examine some new attack vectors that are specific to enclave enabled databases — current commercial EDBs that use enclaves do not provide any integrity guarantees on queries and the plaintext database. Thus, DBAs can modify any installed stored procedures or even tamper with the plaintext database before encryption. These lead to potentially new attack vectors that we examine in Sections 4.2 and 4.3. We also address the question if these attack vectors are strictly equivalent or more powerful than attacks that only exploit query results. For the rest of the paper, we assume the same example schema as in Example 4 but all the columns are now stored using strong encryption (e.g., AES-CBC mode).

4.1 Exploiting Query Results

In the last section, we showed that the attacks illustrated in [14] are not applicable to the newer generation of EDBs based on enclaves. This is primarily because distinct values cannot be distinguished by examining the static database tuples. In this section, we illustrate how this limitation can be addressed by exploiting query results.

Example 5: (DET Encryption): Assume the adversary has access to a plaintext histogram (as in Example 1) in the disease column which has the following distribution: 50% have Flu, 30% have Pneumonia, 20% have Covid. Assume the application uses the following stored procedure to retrieve patient information:
*select * from patients where disease = @param*
Assume the adversary can log the results of all queries to get a $[queryid, outputtuples]$ mapping. He now knows all output tuples with the same qid have the same disease. By comparing the fraction of the tuples returned with the plaintext statistics, he can deduce the disease for these ciphertexts. Since the results simply filter the original tuples (as in Figure 4), the adversary can figure out the original tuples in the database with this particular disease.

Example 6: (OPE Encryption): Assume the application uses the following stored proc:
*select * from patients*
where riskfactor > @param1
and riskfactor < = @param2
By logging the output of the queries, the admin can do the following:
(1) By examining the subset of results of different queries, he can figure out containment relationships among the query results. (2) By using the cardinality, the admin can compute the ranks and the CDF (which are used in [14]). Thus, the OPE attacks in [14] can be replicated in this scenario by analyzing query results.
However, query result based attacks are not equivalent in scope to the PPE attacks [14] as the following example illustrates.

Example 7: (Problem Case): Consider the case where the application uses queries that do not match the admin's statistics profiles. For instance, consider a query workload that queries diseases in a particular age interval with a particular risk factor. By logging the output of the queries, the admin only gets to sample the joint distribution of $(disease, age, riskfactor)$.
While the results of a single equality query can be sufficient to obtain statistics on the domain value selected, in this case, since we only get to sample the joint distribution, the workload may not provide sufficient information to permit any inferences. For instance, assume the application only issues the following queries which are completely disjoint in all dimensions:
Q1(disease="Pnemonia", age <50, riskfactor=3)
Q2(disease="Covid", age > 50, riskfactor=5)
Given only the single column statistics of the disease column (as in Example 6), an adversary cannot draw any conclusions by examining the output of these queries.

In Section 4.2, we examine how an adversary can modify query execution to generate appropriate statistics in order to avoid this problem.

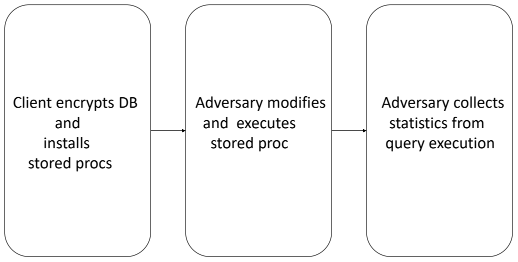
Client encrypts DB and installs stored procs
Adversary modifies and executes stored proc
Adversary collects statistics from query execution

Figure 5: Workflow for query integrity attack

4.2 Query Integrity based Attacks

While query results provide crucial information for an adversary, it does not provide a general purpose solution for inference attacks. As Example 5 illustrates, the key reason is that an adversary can only sample the joint distribution of columns using certain query results which is insufficient to reconstruct the PPE attacks. However, an active adversary can *obtain any missing information* by actively modifying the stored procedures that are installed. The current generation of EDBs focus on protecting keys and the data from disclosure and do not provide any integrity guarantees on the queries/stored procedures that are installed in the system. An admin can actively modify these queries to extract more information to enable inference attacks. The high level workflow is illustrated in Figure 5. The following example illustrates a specific case:

Example 8:(Addressing Example 7) Consider the case of Example 7 where the application uses a stored procedure that does not match the admin's statistics profiles. In order to generate statistics on the disease column alone (and not statistics corresponding to the joint distribution of $disease, age, riskfactor$), an admin can modify query execution to run an additional query as follows:

// run new query
*select * from patients where disease = @param1*
Log [qid, output tuples]
// run original query

The modified query execution first executes a query that generates statistics on only the disease column (which is useful in running attacks on that column). It then runs the original query so a client that executes the stored procedure will see the same result.
Assume the application only issues the following queries:
Q1(disease="Pneumonia", age <50, riskfactor=3)
Q2(disease="Covid", age > 50, riskfactor=5)
The admin modifies query execution to run additional queries Q3(disease="Pneumonia") and Q4(disease="Covid").
By comparing the cardinality output with the statistics on the disease column, an admin can now easily identify the patients with each disease (as in Example 6) which was not feasible by only monitoring the results of the original query in Example 7.

This attack is feasible in practice since the query string with any encrypted constants are stored in untrusted memory and can in principle be overwritten with the new set of queries by an admin. We note that the additional query need not be limited to use the original encrypted constant in the query, an admin can also use encrypted data values from the table in order to fully sample the domain of the column. We label such attacks as *query integrity based attacks* — with such attacks, an admin can always generate statistics that correspond to any single-column statistics that are available (as long as that column is queried in the original query) and as a result, he can replicate the attacks for PPE based databases [14].

Claim: (informal) Using query integrity based attacks, inference attacks equivalent to known attacks for DET and OPE [14] can be constructed for enclave-enabled EDBs.

4.2.1 LIKE predicates

We now examine LIKE predicates. Note that LIKE predicates while not supported in property-preserving EDBs, are fully supported in enclave-enabled EDBs. A key problem for constructing a successful attack for LIKE predicates is the fact that the LIKE predicate is encrypted and an admin cannot generate an encrypted constant corresponding to a new regex pattern. This is unlike the case of equality and range queries where the admin can modify the query and change query constants by randomly sampling values in the encrypted table. With only single column statistics, it is in general not possible to decipher the plaintext values as the following example illustrates.

Example 9: (LIKE predicates) Consider the following scenario: Assume there are 100 patients. Assume the statistics on the disease column is as follows:
Cardinality[disease="Pneumonia"] = 30
Cardinality[disease="Covid"] = 20
Cardinality[disease="Diabetes"] = 15
Cardinality[disease="Bronchitis"] = 15
Cardinality[disease="Flu"] = 10
Cardinality[disease="Fever"] = 5
Cardinality[disease="Cold"] = 5

If an attacker sees an equality query on the disease column with output cardinality of 30, he can immediately conclude that these tuples all have the disease "Pneumonia". However, if a LIKE predicate query on the disease column returned 30 tuples, it is not possible to make any inferences. It could be the case that the LIKE predicate queried only pneumonia patients or covered a regex pattern that included multiple diseases whose combination lead to 30 results such as: (("Covid", "Flu"), ("Diabetes","Flu","Cold"), ("Bronchitis","Flu","Cold"),("Covid", "Fever", "Cold")) Thus, without knowledge of the LIKE pattern it is not possible to construct a general purpose attack for LIKE predicates using query integrity based attacks.

In Section 4.3, we adopt an attack proposed in [6] against searchable encryption in order to infer information about the encrypted LIKE predicate. The key idea is for the adversary to tamper the plaintext database before encryption by introducing some fake tuples and monitor their usage by different queries in order to obtain information about the encrypted LIKE predicate.

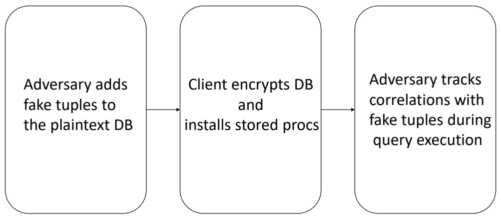
Adversary adds fake tuples to the plaintext DB
Client encrypts DB and installs stored procs
Adversary tracks correlations with fake tuples during query execution

Figure 6: Workflow for data integrity attack

4.3 Database Integrity based Attacks

Another important attack vector for an admin is to tamper with the plaintext database *before* the data is encrypted. For large databases, it is not efficient to insert the database records one at a time while encrypting them — it is more efficient to bulk-load the plaintext database to the cloud and then encrypt it using the enclave. In fact, this is one of the key features enabled in [10]. Moreover, many databases already reside in the cloud in plaintext before encryption is enabled. An admin can easily tamper with the database at this point in order to gain more information while executing queries on the encrypted database.

Similar attacks (where the plaintext data is tampered *before* encryption) have been studied in the context of searchable encryption [6]. Here, an adversary augments a document corpus with fake documents. By correlating the hashes of the words in the fake documents with other documents in the corpus, he can construct attacks [6].

The adversary first augments the database with fake tuples, the client then encrypts the database. While queries are executed, the adversary tracks the fake tuples to discover relationships between encrypted tuples and to infer the semantics of the encrypted LIKE predicate. Note that this gives the adversary the important ability to track correlations with any new encrypted tuples that have been inserted since the database was encrypted (and not just the tuples that were present in the original plaintext database). The high level workflow is illustrated in Figure 6 and the following example illustrates a specific case. In this paper, we assume an adversary explicitly adds fake tuples to the plaintext database — it is also possible that an adversary uses the RIDs of tuples that were already in the plaintext database to track correlations.

Example 10: LIKE predicates. Consider the example in Example 9 where we illustrated that query results provide incomplete information to construct a successful attack for LIKE predicates. We now illustrate how an adversary can use the workflow in Figure 6 to construct a successful attack.

That scenario had 100 patients and 7 distinct diseases. Assume the adversary has already inserted 7 fake tuples, one for each distinct disease in the database. For simplicity, assume the original database was empty modulo the fake tuples and a new whole new set of encrypted patient data (now totaling 100 tuples) has now been inserted. The question is can the adversary retrieve the plain-text corresponding to any of the new encrypted data?
Assume the adversary gets two LIKE predicate queries: Q1

with cardinality 30 and Q2 with cardinality 10. The adversary can now track if his fake tuples are part of each result set (e.g., by tracking the ROWIDs() of those tuples as part of query execution).

For instance, if Q1 includes the fake tuples for “Covid” and “Flu” and Q2 includes only the fake tuple for “Flu”, the adversary can infer that the first encrypted regex predicate is essentially equivalent to a (“Covid” OR “Flu”) query and the second regex predicate is equivalent to a (“Flu”) query. From this information, he can now deduce that all the second query output tuples have the disease “Flu” and all the first query tuples excluding the output of the second query (i.e., the output of the query difference Q1 - Q2) have the disease “Covid”. Thus, he can construct a successful attack since he can now guess the equivalent semantics of the encrypted LIKE predicate. This is something an adversary could not have accomplished (e.g., in Example 9) without tampering the data to introduce fake tuples. We note that an adversary does not need access to any background statistics to carry out a successful attack using this approach.

In general, the set of fake tuples included in a query result set provides a *unique signature* for that query (e.g., “Covid” OR “Flu” for Q1 in Example 11). By analyzing the signatures of a workload of queries, an adversary can try to obtain subsets of rows that have a signature that corresponds to an unique domain value (e.g., just “Flu” or “Covid” in Example 10) — then, he has uniquely decoded all ciphertexts in the table with that value. At a high level, an algorithm to carry out the attack is as follows: Compute differences in signatures between each pair of queries in the workload and then compute the transitive closure of such differences — any signature that is present in the transitive closure and identifies a unique domain value results in a successful attack for that domain value. We defer algorithmic details and an experimental evaluation that demonstrates the effectiveness of this attack for real workloads to future work. As far as we are aware, this is the first published attack vector for encrypted LIKE predicates.

However, there are important caveats— First, this attack requires the admin to insert fake tuples for *every* value in the domain in order to understand the semantics of the regex predicate. So this attack is ideally suited for low entropy ”string” columns (e.g., performance rating in a company) or even numeric columns (e.g., salary column with a small number of distinct values). In general, the signatures obtained using the fake tuples can also be used for constructing attacks for OPE and equality queries. Second, we note that this is not necessarily a general purpose technique since it is possible that in spite of “guessing” equivalent regex expressions, the set of queries run could provide incomplete information for deciphering ciphertext.

Example 11: LIKE predicates and incomplete information. Assume the query workload in Example 8 executed two LIKE predicate queries, Q1 which includes the fake tuples for (”Diabetes”, ”Flu”) and Q2 which includes the fake tuples for (“Fever”, “Cold”). An adversary can now guess that a particular output tuple is one among the output two diseases but lacks sufficient information to uniquely identify the disease. In general, applications could issue a query workload where unique inferences are not possible.

In Examples 10 and 11, we assume the application used a single LIKE predicate. In case, the application used a more complex query that ran a conjunction of predicates including the LIKE predicate, the admin would modify the query (as in Section 4.2) to run the LIKE predicate in isolation in order to carry out the attacks. Thus, we label this technique as a *database integrity based attack* since it can tamper with both the database and query contents.

5. DISCUSSION

We first summarize the attacks discussed in this paper and discuss the key implications. First, we wish to underscore the point that current EDBs already provide a significant security bar for PII data particularly when compared to the prior alternative of storing PII in plaintext. The attacks suggested in this paper are conceptual and need high sophistication on the part of an admin[3] to pull off in practice.

Given that EDBs are here to stay, the design of future generations of these systems will be largely influenced by understanding any new attack vectors and building safeguards for them. This paper emphasizes a broad attack vector of integrity-based attacks that can result when an adversary can tamper with either the query or the database contents. In particular, we also discuss how these attacks translate to other EDBs built using coarse grained architectures [4, 16] in Section 5.1.1 as well EDBs that use oblivious query operators [3, 9] in Section 5.2.

5.1 Summary of Attacks

Table 1 presents a high-level summary of the attacks discussed in the paper. The key takeaways are as follows. (1) Previously published static database attacks for PPE based EDBs [14] are not directly applicable. Thus, enclave-enabled databases do significantly improve security where an adversary steals the database file or where the adversary runs a select * query to select all the rows of a table to get a copy of the database.
(2) Query result based attacks are not a general purpose technique but can work in cases where the application issues queries that generate results that are consistent with any background statistics available. Hence, these entries are qualified with an asterix. In addition, such attacks do not provide enough information to construct successful attacks for LIKE predicate queries.
(3) All the previously published attacks proposed [14] for equality and range queries in PPE based EDBs directly translate to enclave-enabled database when we use query integrity based attacks where an adversary can modify a query to collect any appropriate statistics. However, this approach cannot work for LIKE predicates because the LIKE predicate is encrypted and cannot be modified.
(4) Finally, database integrity based attacks (where an adversary can tamper with both the query and the database contents) can be used by an adversary to successfully work with equality, range and LIKE queries. LIKE predicates are qualified with an asterix since in some cases there might be insufficient information in the query workload to decipher plaintext.

5.1.1 Applicability of Attacks to other EDBs

While we have centered the discussion on a particular architecture of EDBs (see Figure 2), we now comment on the

[3]For instance, there are orthogonal safeguards such as auditing an admin’s actions.

Attack Type	Equality	Range	Like
Static DB	No	No	No
Query Result	Yes*	Yes*	No
Query Integrity	Yes	Yes	No
Data Integrity	Yes	Yes	Yes*

Table 1: Attacks on fine-grained enclave-enabled EDBs

applicability of these attacks to other EDBs developed using coarse grained architectures (e.g., TrustedDB [4], EnclaveDB [16], ObliDB [9]). We postpone the discussion of ObliDB [9] to section 5.2.3 after we discuss oblivious operators in Section 5.2.2 and focus on TrustedDB [4] and EnclaveDB [16] in this section.

Recall that EDBs using a fine-grained architecture [1, 10] only run expression/predicate evaluation in the enclave (see Figure 2) in which any predicates on encrypted data are evaluated securely in the enclave and the results are returned in plaintext and used by a traditional query engine. EDBs using a coarse grained architecture run entire query execution engines in the enclave (e.g., TrustedDB runs a version of SQLite in the enclave). The "non-enclave" components are primarily used for data storage and data shipping to the query engine running in the enclave. In such EDBs, queries are encrypted and signed by the client and submitted directly to the query engine in the enclave. This provides the key property that any query string *cannot be modified* and executed by the administrator who cannot access the memory contents of the enclave.

Query encryption also provides the additional benefit that the SQL string skeleton is encrypted and hidden from the admin — however, this is *not a guarantee* for the following reason: An admin can guess the SQL query skeleton using a variety of background information: If only the data pages corresponding to a single column from a table is accessed by the query engine in the enclave, he can guess that it is a single table query that has some predicate on that column. In addition, he can guess the SQL from the application — for instance, if a hospital search interface with support for regular expression matching is typically used for searching patients by name or by disease – this can only translate to LIKE predicate queries on the name column or disease column in the database.

We note that both EDBs use the iterator model of execution and *do not aim to decorrelate the database tuples from the output result* (e.g., by using oblivious operators or buffering the results). The enclave typically uses a paging model where pages of encrypted data are read into the enclave one page at-a-time and the results are stored in host memory (e.g., [16]) as soon as they are generated. In this scenario, an adversary can figure out the correlation between input and output tuples as follows. Consider a simple example of a Table T having a single page that has ten tuples that is encrypted and read into the enclave using a scan operator. Assume that the enclave processes a filter tuple and immediately writes out the output to the host memory. An adversary can track correlations using one of the following methods:
(1) He can simply use a timing attack to figure out which tuple (among the ten) passed the filter predicate based on how long it takes to produce an output tuple.
(2) Both EDBs do not support data integrity on the plaintext database — thus an adversary can create a version of the plaintext database with ten pages and store only one original tuple in each page. Now by monitoring which page was read into the enclave when the result was generated, he can obtain the correlation.

Given these properties, we can derive the following conclusions:
(1) Since both TrustedDB and EnclaveDB use signed queries, query integrity based attacks (Section 4.2) which rely on modifying the query to get additional statistics are *not feasible* in such systems — this is a clear difference from EDBs that use a fine-grained architecture.
(2) Both EDBs do not aim to decorrelate the database tuples from the output result. Thus, query result based inferences are feasible in both systems.
(3) TrustedDB [4] does not provide any integrity guarantees and EnclaveDB [16] provides integrity guarantees for only the encrypted database and not the plaintext DB before encryption. The attacks in Section 4.3 work by correlating any fake tuples inserted by the admin before encryption with the actual query results — they do not rely on the need to tamper with the input queries. For instance, consider a database application that issues only LIKE predicates on a single column for the EDB— for this specific scenario, the attack discussed in Section 4.3 applies to both TrustedDB and EnclaveDB.

We defer a more thorough examination of how the attacks discussed in this paper can be ported to [4, 16] to future work.

5.2 Alternative Approaches

All EDBs that use the iterator model of execution [11] permit some form of query result based information leakage. We first examine query result based attacks (e.g., the leakage profile of query execution as shown in Figure 4) and ask the question: Can we build EDBs that can alter this information leakage profile in order to reduce query result based information leakage?

5.2.1 *Query Result Indistinguishability*

First, can we stop query result based attacks completely? Any EDB that uses cell level encryption and runs queries will leak this information. Only EDBs that leverage different architectures that provide some form of *query-result indistinguishability* [13] can offer protection against query result based leakage. For a query workload of filter queries, query result indistinguishability implies independent of the selectivity of the query, the query result sizes should be virtually indistinguishable. Thus, a query with 0.1% selectivity or 99.9% selectivity should return the "same" result size — which is virtually the entire table. If every filter query returns the entire table, then this workload effectively leaks no information. However, this has significant performance implications — for instance, any form of indexing is incompatible with this property and the filtering of the actual results has to be done at the client. Thus, the EDB is virtually reduced to a storage system — as a result, conceding query result leakage seems inherent given that EDBs need to support efficient query processing.

5.2.2 *Leveraging Oblivious Operators*

While some form of query result leakage seems unavoidable for EDBs, a key property that enabled the attacks in

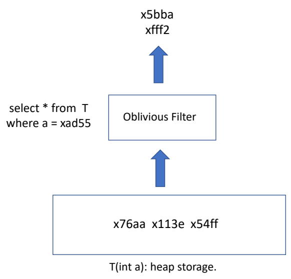

Figure 7: Oblivious Filter Operator

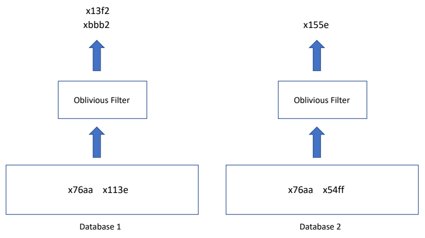

Figure 8: Data Integrity Attack for Oblivious Filter Operator

Section 4 is the ability to track correlations between tuples in the database and tuples in the query result. We first note a simple re-encryption of the tuple in the enclave is usually insufficient as traditional query engines process tuples using the iterator model and correlations between input and output tuples can easily be tracked by an active adversary (as discussed in Example 3). However, a complete redesign of the query engine centered around *oblivious operators* can remove such correlations. Can we stop the attacks in Table 1 by simply integrating oblivious operators in fine-grained EDBs [1, 10]? Oblivious operators [3, 9] provide the important property that they leak *only the cardinality* of the query result — thus, an adversary cannot figure out the source database tuples that belong in the query result. The following example illustrates an oblivious filter operator.

Example 12: Oblivious Filter Operator: Assume a Table T with three tuples (as shown in Figure 7). Assume two of these have the same plaintext value. Consider the problem if an admin can decode which two tuples are identical after a user runs an equality query. As shown in Table 1, this is feasible using query integrity Attacks. However, an oblivious filter operation can decorrelate the source tuples from the output and potentially stop this attack. A naive implementation of an oblivious filter [3] operator would scan the entire table into the secure enclave, evaluate the filter operator, re-encrypt the tuples and output them after buffering all the results in the enclave memory (in practice, implementations can handle limited enclave memory with the same guarantees). Note that this pattern of query execution breaks the classic iterator model [11] where an output tuple is generated immediately if an input tuple satisfies the filter and as a result this breaks the ability to correlate input and output tuples. Figure 7 shows why an admin who only has access to the query results cannot directly correlate the output tuples with the corresponding tuples in the database and figure out which of the two tuples are equal. Thus, an oblivious operator only leaks the cardinality of the result and *not the source database tuples* that belong to the output.

Example 13: Attack for Oblivious Filter: While an oblivious operator leaks only the cardinality, an admin with unrestricted ability to tamper with the data and queries can still construct an attack for this example. For instance, for the simple database in Figure 7, an admin can create two different databases with different subset of the tuples in the original EDB: Database 1 with (x76ee, x113e) and Database 2 with (x76ee, x54ff) and then run the query on both databases. While the admin still does not have the ability to correlate input and output tuples, the very fact that the query on Database 1 returned two results implies that the two source tuples in Database 1 are equal. In general, for a table with n rows, an admin can create n such databases with each database containing one tuple from the original database and by running the query on each of these databases can now compute all source database tuples are equal to the constant specified in the query. This is something that he could not have done in Example 12 by only examining query results.

Thus, even if we use oblivious operators in EDBs [3], an active adversary can still construct attacks that a passive adversary cannot by leveraging data integrity based attacks. This underscores the point that *integrity-based attacks form a broad attack vector* for a variety of EDBs built using enclaves.

5.3 Integrity Guarantees for EDBs

Given the fact that integrity based attacks are conceptually feasible for a variety of EDBs built using enclaves, can current systems that leverage traditional query processing architectures [10] be augmented to protect against the attacks discussed in this paper? We believe EDBs need to invest more in integrity protection of queries and data. We abstract two important properties that are necessary to protect against such attacks. We define an authorized user to be a user who has access to the data encryption keys — hence, a DBA is not considered an authorized user of the database.

Property 1: Any SQL query (read/update) that is executed is digitally signed by an authorized user and is run unmodified and executed exactly once by the DBMS.

Property 2: The state of the database system reflects a serialized order of all signed SQL updates that were issued by authorized users.

Any system that guarantees Property 1 for the encrypted database ensures that an admin cannot alter queries or run any other queries and hence protects against query integrity attacks. As we have shown in this paper, an admin should not be allowed to run arbitrary queries or modify existing queries — we note that this does not interfere with the regular tasks of an administrator (e.g., backups, capacity planning). In order to prevent against data integrity based attacks, we need to guarantee Property 1 and Property 2 for *even the plaintext database before encryption* and the encrypted database. Property 2 in particular prevents an admin from tampering the database in any form. As far as we are aware, this are no commercial query execution engines that provide the above guarantees. It is interesting future work to extend current commercial systems [10] to provide these guarantees.

One possibility is to re-examine coarse-grained architectures(e.g., TrustedDB [4], EnclaveDB [16]). Such systems guarantee Property 1 because they use signed queries and run parts of the query engine in the enclave — they can thus prevent the attacks discussed in Section 4.2. However, they do not provide Property 2 for the plaintext database before encryption. A large number of databases already reside in plaintext form in the cloud before encryption is enabled — if these systems can be extended to support Property 2 for the plaintext database before encryption then this attack vector can be prevented. In addition, these systems largely rely on the iterator model of query execution [11] (i.e., no oblivious operators) and thus cannot prevent an admin from correlating output tuples in a query with the corresponding tuples in the source database which can lead to the attacks discussed in Section 4.1.

ObliDB [9] proposes an EDB architecture that combines *oblivious operators and data integrity protection.* Opaque [19] is another system that combines oblivious operators and data integrity protection in a map reduce setting. However, since Opaque is not designed for traditional OLTP databases, we restrict our discussion to ObliDB. Despite the fact that ObliDB does not provide integrity guarantees for the plaintext database, it does not permit any of the attacks discussed in Section 4. This is because oblivious operators removes any correlations between the output and the database tuples. In addition, the attack for the oblivious filter operator discussed in Example 13 is not permitted because of integrity guarantees in the EDB. However, the redesign of the engine poses non-trivial performance constraints. For instance, a table scan query needs multiple scans of the table in order to buffer tuples in the enclave and database staples such as parallelism and concurrency control are not yet supported. As a result, these techniques are yet to be incorporated in commercial query engines which use highly-optimized traditional query processing architectures. Examining if we can extend ObliDB to efficiently support the full SQL API (including concurrency control) is an interesting area of future work.

In addition, there is recent work on building verified plaintext key-value stores (e.g., Concerto [2]) using trusted hardware that provides Property1 and Property2 for a (non-encrypted) key-value store. Concerto provides high throughput by using cryptographic primitives (such as Blum hashes) that can be optimized for concurrency. While these techniques provide high performance (supporting millions of transactions per second), they currently only cover a key-value store API. It is interesting to examine if these techniques can be generalized to build EDBs that cover the full SQL API. Finally, we note that any EDB that provides Property1 and Property 2 will provide an important guarantee that *an active adversary can only learn as much from the system as a passive adversary.* We believe that this is an interesting north star goal for next generation commercial EDBs to pursue.

6. REFERENCES

[1] A. Arasu, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, and R. Ramamurthy. Transaction processing on confidential data using cipherbase. In *31st IEEE International Conference on Data Engineering, ICDE 2015*, 2015.

[2] A. Arasu, K. Eguro, R. Kaushik, D. Kossmann, P. Meng, V. Pandey, and R. Ramamurthy. Concerto: A high concurrency key-value store with integrity. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017*, 2017.

[3] A. Arasu and R. Kaushik. Oblivious query processing. In *Proc. 17th International Conference on Database Theory (ICDT), 2014*, pages 26–37. OpenProceedings.org, 2014.

[4] S. Bajaj and R. Sion. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In *Proceedings of the 29th ACM SIGMOD International Conference on Management of Data.* ACM, 2011.

[5] V. Bindschaedler, P. Grubbs, D. Cash, T. Ristenpart, and V. Shmatikov. The tao of inference in privacy-protected databases. *Proc. VLDB Endow.*, 11(11):1715–1728, 2018.

[6] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In I. Ray, N. Li, and C. Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security 2015*, 2015.

[7] V. Costan and S. Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[8] V. Costan, I. A. Lebedev, and S. Devadas. Secure processors part I: background, taxonomy for secure enclaves and intel SGX architecture. *Foundations and Trends in Electronic Design Automation*, 11(1-2), 2017.

[9] S. Eskandarian and M. Zaharia. Oblidb: Oblivious query processing for secure databases. *Proc. VLDB Endow.*, 13(2):169–183, 2019.

[10] P. A. et al. Azure SQL database always encrypted. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, June 14-19, 2020*, pages 1511–1525, 2020.

[11] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[12] P. Grofig, I. Hang, M. Härterich, F. Kerschbaum, M. Kohler, A. Schaad, A. Schröpfer, and W. Tighzert. Privacy by encrypted databases. In B. Preneel and D. Ikonomou, editors, *Privacy Technologies and Policy*, 2014.

[13] M. Kantarcioglu and C. Clifton. Security issues in querying encrypted data. In S. Jajodia and D. Wijesekera, editors, *Data and Applications Security XIX, 19th Annual IFIP WG August 7-10, 2005, Proceedings*, volume 3654 of *Lecture Notes in Computer Science*, pages 325–337. Springer, 2005.

[14] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In I. Ray, N. Li, and C. Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA,*

October 12-16, 2015, 2015.

[15] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: processing queries on an encrypted database. *Commun. ACM*, 55(9), 2012.

[16] C. Priebe, K. Vaswani, and M. Costa. Enclavedb: A secure database using SGX. In *IEEE Symposium on Security and Privacy 2018*, 2018.

[17] R. L. Rivest, L. Adleman, and M. L. Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation, Academia Press*, pages 169–179, 1978.

[18] P. Wayner. Translucent databases. 2002.

[19] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017.