On The Security Of Querying Encrypted Data

Arvind Arasu
Microsoft Research
arvinda@microsoft.com

Raghav Kaushik
Microsoft Research
skaushi@microsoft.com

Ravi Ramamurthy
Microsoft Research
ravirama@microsoft.com

ABSTRACT

Data security is a serious concern when we migrate data to a cloud DBMS. Database encryption, where sensitive columns are encrypted before they are stored in the cloud, has been proposed as a mechanism to address such data security concerns.The intuitive expectation is that an adversary cannot "learn" anything about the encrypted columns, since she does not have access to the encryption key. However, it turns out that the act of query processing over encrypted data can reveal information that in the worst case can undermine the very purpose of encryption. In this paper, we argue that such information disclosure should not be handled in an *ad hoc* manner; in particular, query processing over encrypted data requires: 1) a precise contract (in the form of a security model) that specifies what information is permitted to be disclosed during query processing and 2) a query engine that is carefully engineered to meet the contract efficiently. We believe these are important building blocks in designing a "secure" database-as-a-service paradigm. In this paper, we develop a security model for query processing over encrypted data and take the first steps in understanding the space of secure query processing.

1. INTRODUCTION

There has been an increasing commercial [3, 22] and research [10] interest in the *database-as-a-service* paradigm over the cloud (*Cloud DBMS*). One of the barriers to adoption of cloud DBMSes is data security and privacy. Data is a valuable asset to most organizations and there exist realistic threats to security of data stored in the cloud [9, 14, 19, 27]. For instance, an external adversary can gain access to sensitive data by exploiting software vulnerabilities in cloud servers; insider attacks could range from a curious system or database administrator "browsing" the database to more sophisticated attacks (e.g., snooping the memory contents for any plain-text data) by someone having physical access to servers [14].

One important mechanism to address data security concerns [16, 23, 24, 2, 5, 15, 25] is *data encryption*, where a client encrypts any sensitive columns in the data before storing it in the cloud. Ideally, an adversary cannot "learn" anything about the encrypted columns without the encryption key which is known only to the client.

Encryption, however, makes it difficult to perform computations (in particular, query processing) over data. Standard encryption schemes are designed to hide data being encrypted, while, informally, we need to"see" the data to perform computations on it. We note that the straightforward approach of temporarily decrypting the data (in the cloud) to perform computations is not secure: The cloud service now needs to store the encryption key, which could find its way to the adversary; further, the adversary might be able to read plain-text data, e.g., by performing a memory scan during the time the data is decrypted. Recently, there has been theoretical work on *homomorphic encryption schemes* [11] that are designed to allow arbitrary computation over encrypted data. However, current homomorphic encryption schemes are prohibitively expensive both in space and time [5].

Prior Work: A simple strategy to deal with computational challenges introduced by encryption is to use the cloud just for storage and perform all computations on the client [20]. A generalization of this approach is based on the observation that many columns in a database are not sensitive. Such columns (that could include additional "computed columns") are stored in plain-text, and part of the query processing involving the plain-text columns is performed in the cloud and the remainder, in the client [15]. Another strategy relies on encryption schemes such as *deterministic* [6] and *order-preserving* [2, 8] that reveal some information about data being encrypted. For example, a deterministic encryption scheme encrypts all occurrences of the same value identically and therefore allows equality operations such as equi-joins and point lookups over encrypted data [25]. However, deterministic encryption potentially allows an adversary to learn plain-text values using frequency information of encrypted values (e.g., in `Gender` column)[1]. Similarly, order-preserving encryption allows range predicates over encrypted data. We note that these "weaker" encryption schemes do not cover *all* database operations and thus require either settling for a subset of SQL [25] or require non-trivial computation in the client (including potentially large data transfers) which negates some of the advantages of the cloud setting.

A different approach [5] to query processing uses a *trusted hardware module* such as a secure co-processor in the cloud server. Such trusted module is designed to be tamper-proof and inaccessible even to an adversary with physical access to the machine. The trusted module stores the client's encryption key and any encrypted data is securely decrypted and processed within the module. The idea of using trusted hardware for security has a long history in systems [17, 26] that is now being applied to data processing [5, 7].

[1]In contrast, more secure (e.g., CPA-secure) encryption schemes would encrypt different instances of the same value differently and hide its frequency.

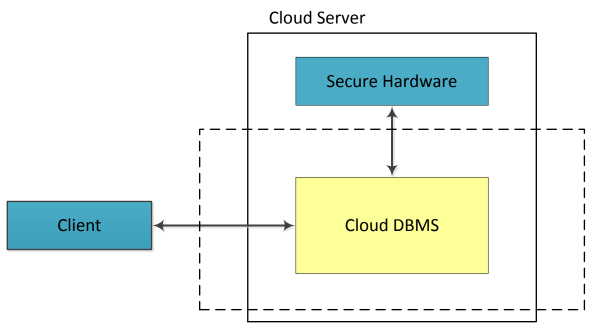


Figure 1: Secure-hardware based query processing architecture. Data and computation within dotted rectangle are visible to adversary.

1.1 Security and Query Processing

All of the approaches to database encryption reviewed above are concerned with security of *data-at-rest*. They ensure that an adversary getting access to the encrypted database (e.g., as stored in the disk) does not "learn" anything about the encrypted columns.

In this paper, we argue that query processing over encrypted data can itself leak information about the underlying plain-text data and can potentially undermine the overall security of the system. As a simple but somewhat obvious example, communicating the result of a query reveals information about the size of the query output; we shortly present examples where the degree and subtlety of information leakage is greater.

The fact that query processing leaks information is not very surprising: It is well-known that just the sequence of memory locations accessed by a program (e.g., merge-sort algorithm) can reveal information about its input such as the permutation of the input sequence [13]. Similarly, the related area of work in *private information retrieval* [28] is based on the observation that the access patterns for search queries (on non-encrypted data) can potentially reveal information about the query. *However, we believe that information leakage in general database query processing and mechanisms to formalize and address this leakage have been largely unstudied and form the central focus of this paper.*

We illustrate information leakage in the context of a trusted hardware based architecture for query processing, shown in Figure 1. We use a simple abstraction to discuss our examples of information leakage—we assume that the cloud server includes a DBMS that runs in the *untrusted module (UM)* which contains CPU, memory, and disk. The *trusted module (TM)* is a secure hardware that has CPU and a small amount of persistent storage to store the client key—any decryption of data encrypted by client happens within TM. Specific operators within the DBMS invoke TM for secure computations; e.g., a filter operator could send an encrypted record to TM which decrypts the record, evaluates the filter, and returns the result (true/false). The adversary, who we refer to as *Eve* (eavesdropping adversary) can observe data and computation in the UM and communication between UM, TM, and client. We assume that Eve has access to the query plan that runs in the UM. We note that this architecture is different from that of TrustedDB [5], which runs another DBMS within TM. We pick this architecture mostly for presentational simplicity and our general observations and results extend to alternate setups such as those of TrustedDB.

EXAMPLE 1.1. *Figure 2 shows a health care database instance with two tables. The client encrypts the* `Disease` *column of each* `Ailment` *record using a CPA-secure probabilistic encryption scheme; the same value* `AIDS` *is encrypted differently in different records. The* `Gender` *column in each* `Patient` *record is encrypted using a deterministic encryption scheme. The other columns are stored in clear-text. Consider the following query that finds the names and cities of patients with* `AIDS`*:*

Name	*Disease*
Alice	!@#$xyz
Bob	@%^abc
Chen	*&#pqr
Dana	(p#z~94

`Ailment` (Encrypted)

Name	*City*	*Gender*
Alice	Seattle	2xU%b
Bob	Madison	Ry!<4&
Chen	Palo Alto	Ry!<4&
Dana	New York	2xU%b

`Patient` (Encrypted)

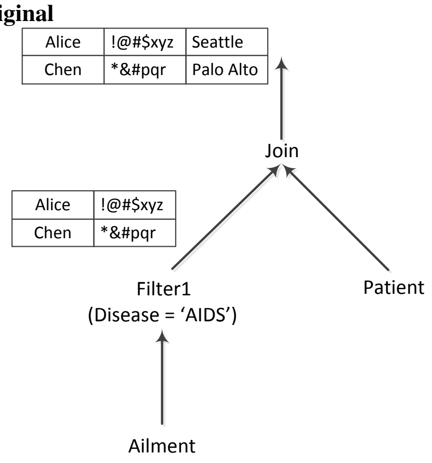
Name	*Disease*
Alice	AIDS
Bob	Flu
Chen	AIDS
Dana	Diabetes

`Ailment` (Original)

Name	*City*	*Gender*
Alice	Seattle	Female
Bob	Madison	Male
Chen	Palo Alto	Male
Dana	New York	Female

`Patient` (Original)

Figure 2: Sample database encrypted stored in a cloud DBMS and its original



Alice	!@#$xyz	Seattle
Chen	*&#pqr	Palo Alto

Alice	!@#$xyz
Chen	*&#pqr

Figure 3: Query plan 1 of Example 1.1

```
select Name, Disease, City
from Patient join Ailment on Name
where Disease = 'AIDS'
```

Consider Plan 1 for the query shown in Figure 3. Plan 1 applies the filter on `Disease` *column before joining the two tables. Consider an implementation (Filter1) of the filter operator that sends each* `Ailment` *record to TM, which decrypts the* `Disease` *column and returns* true *if its value is* `AIDS` *and* false *otherwise (we assume that the return value is in plain-text). Records that satisfy the filter are forwarded to the join operator. Plan 1 uses a traditional join operator in UM to perform join on the plain-text column* `Name`*.*

Recall that Eve has access to the plan and thus can infer that there is an equality predicate on the Disease column (we assume that the query constants are encrypted and thus Eve does not know the specific disease being queried for). By observing the output of the plan, Eve can infer that `Alice` *and* `Chen` *have a common disease, something that could not have been learned from the encrypted (data-at-rest) tables in Figure 2. This is a potential security breach: e.g., if Eve has external knowledge that Chen has AIDS, she learns that Alice too has AIDS. Further, (in absence of the above external knowledge), the only thing stopping Eve from learning that Alice and Chen have AIDS is the secrecy of the query*

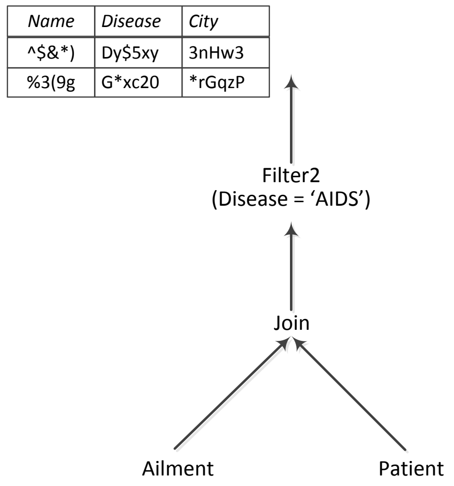


Figure 4: Query plan 2 of Example 1.2

text. Depending on the application, this may constitute a security weakness: Many applications invoke a small set of stored procedures whose templates might be public knowledge. Even otherwise, Eve can learn the query template if the plan is materialized in UM. Once Eve learns the query template, she might be able to guess the constant `AIDS` *from her knowledge of query result size and the general prevalence of various diseases.*

EXAMPLE 1.2. *Consider an alternate plan for the query of Example 1.1 shown in Figure 4. This plan first performs a join of the two tables in UM and then applies the filter on* `Disease`*. The implementation (Filter2) of the filter is slightly different: Filter2 sends each join record r_i to TM. TM decrypts the* `Disease` *column of r_i and if it is equal to* `AIDS`*, re-encrypts (using the client's key) each column of r_i using a probabilistic (CPA-secure) encryption scheme and returns the resulting encrypted record r_o; if it is not equal to* `AIDS` *nothing is returned. Note that the encryption of* `Disease` *column in r_o and r_i would very likely be different since the encryption scheme is probabilistic. Unlike Plan 1, note that Eve can no longer infer that Alice and Chen have the same disease from observing the output of the query.*

Surprisingly, Plan 2 as described leaks exactly the same information as Plan 1! Recall that Eve can monitor the traffic between the UM and the TM. Given the knowledge of the query plan (and which operators are pipelined), Eve observes that TM produces some (encrypted) output for certain inputs (in particular, join records corresponding to Alice and Chen) and infers that their records appear in the output. We can address the above problem by modifying Filter2 to buffer its output tuples until the whole input is read. However, this is not a general purpose solution since the TM is assumed to have limited memory.

We note that the above examples assume that Eve has access to the query plan. It might seem that we can hide the query plan by running operators within TM. In Section 4 we argue why hiding the query plan from Eve might be difficult. We use the examples above to make the following observations:

1. Query processing can leak information about data being processed, and encrypting data-at-rest and performing decryption unobserved by the adversary does not preclude this.

2. The security of some query processing strategies might rely on the secrecy of the query text or the query plan. This reliance is a potential security weakness for some applications.

3. Certain amount of information leakage may be unavoidable for practical reasons. For instance, Plan 2 reveals the final output cardinality—while this disclosure can also be avoided by padding the output with fake tuples to be constant independent of the join selectivity, it results in a significant performance penalty.

In this paper, we do not take a position that information leakage during query processing should be avoided at all costs; in fact, a client might settle for lesser security for better performance. But we argue that information disclosure should not be handled in an *ad hoc* manner. Therefore, query processing over encrypted data requires: 1) a precise contract in the form of a security model that specifies what information is permitted to be disclosed during query processing and 2) a query engine that is carefully engineered to meet the contract efficiently. We believe these are important building blocks in designing a "secure" database-as-a-service paradigm.

1.2 Our Contributions

Security Model: Our main contribution is a security model (Section 3) that formalizes information leakage during query processing over encrypted data. A client can suitably instantiate the security model to precisely specify what information can and cannot be leaked during query processing.

An instantiation of our security model is parameterized using a *permit function*, which bounds what information can leak during query processing; revealing any information not covered by the permit function violates security. To formalize information leakage, we adapt the classic cryptographic notion of *semantic security*. Informally, a query processing strategy is semantically secure (and does not leak information) if no polynomial time adversary can compute any function over the original plain-text database, discounting for what is allowed by the permit function. We show the robustness of this definition by showing its equivalence to *database indistinguishability*, an adaptation of another classic cryptographic notion of message indistinguishability.

Our security model assumes an adversary who has full access to the clear-text of the queries being executed, ensuring that overall security does not rely on secrecy of query text; as discussed in Example 1.1, relying on secrecy of query text could constitute a security weakness.

We highlight the following aspects of our security model: (1) It is orthogonal to base data encryption. It can be used to provide query processing security for any configuration of base data encryption that might combine plain-text, deterministic, order-preserving and strong CPA-secure encryption schemes. (2) The security model makes minimal assumptions about the query processing architecture and is applicable to architectures that perform secure computations in the client, in trusted hardware, or a hybrid of the two.

Query Processing Architecture: The second contribution of the paper is to discuss the challenges involved in building a secure query processing system based on our security model. We focus on the trusted hardware architecture. Our goal is to support the full complexity of SQL independent of how the data is encrypted. We argue that the strongest permit function for which we can meet the above goal requires every relational operator to reveal nothing but its output size (Section 4). We review prior work and find that they are either incomplete in the subset of SQL supported or insecure (based on the above permit function).

We then discuss a blue-print of a secure query processing architecture (Section 5). We address the following challenges: (1) We develop secure operators that can be used to address arbitrary SQL. As the examples in this section indicate, designing secure operators is not merely a matter of ensuring that decryption happens only in

TM; we have to suitably re-encrypt the operator output and also hide access patterns. We can incorporate the above techniques in a DBMS by adding new physical operators that are secure. (2) We design a secure B-Tree index over encrypted columns. We show that a straightforward adaptation of standard B-Tree indexes is insecure and propose a secure B-Tree that uses *Oblivious RAM (ORAM)* technology to make data access look (statistically) random to an adversary. One feature of the index design is that it minimally impacts the index code. Rather the changes are to the storage subsystem to add ORAM. In this way, we are able to support point and range lookups over columns that can be encrypted using the strongest CPA-secure encryption. (3) We show that a query optimizer can reason about the security of a plan on a per-operator basis by checking if each operator is secure.

We also discuss the performance of our blue-print. The good news is that our modifications do not impose a significant overhead for scan-based query plans, plans in which leaf nodes are scan operators (such plans are commonly used for decision support queries). In fact, we show that our secure query processing has the same data and time complexity as traditional query processing. However, the bad news is the fact that for queries that require the use of indexing either for point predicates or range predicates, the system can impose significant overheads. This is mainly because our reliance on ORAM technology leads to a loss of spatial and temporal locality of reference. Even though our architecture tries to minimize the reliance on ORAM technology, we show these overheads are unavoidable for supporting a "secure" version of indexing.

We believe these are important building blocks in designing a "secure" general-purpose database-as-a-service paradigm, which is an interesting research goal. Finally, we note that our description of our query processing architecture is at a conceptual level. Needless to say, designing and implementing a system based on this conceptual description would introduce many interesting engineering challenges beyond the scope of this paper. We defer such an implementation and its evaluation to future work.

2. PRELIMINARIES

We now review encryption of data-at-rest and associated security guarantees. We also present an abstraction for query processing over encrypted data that we use in our security model.

2.1 Encrypting "Data At Rest"

Classic cryptography focuses on encryption of messages modeled as strings. An *(message) encryption scheme* consists of an encryption and decryption function. The encryption function takes as input a secret key and a *clear-text* string and returns an encrypted *cipher-text* string. The decryption function given the secret key and cipher-text, returns the clear-text. In this paper, we use *symmetric* encryption schemes where the same secret key is shared between the encryption and decryption functions (in contrast, public-key encryption uses separate keys for encryption and decryption.)

A *database encryption scheme* builds on message encryption to encrypt databases. A natural approach would be to treat the entire database as a single string for the purpose of encryption. Alternately, we could break the database into blocks and encrypt each block separately. However, if we wish to perform query processing directly over encrypted data, we need to be able to "look inside" encryption. Accordingly, prior work has considered *column-level* encryption where we associate one message encryption scheme and key with each column. The database is encrypted cell by cell using the corresponding column-level encryption function and key. The formalism we introduce below for data encryption does not stipulate which of the above ways the database is encrypted. However, unless mentioned otherwise, we assume column-level encryptions in the rest of the paper.

DEFINITION 2.1. *A* (symmetric-key) database encryption scheme *is a pair of polynomial time algorithms (Enc, Dec) where Enc is probabilistic and Dec is deterministic such that for every key-length k, for every $K \in \{0,1\}^k$ and database instance D, $Pr[Dec(k, K, Enc(k, K, D)) = D] = 1$. Here, the probability is taken over random coins of Enc. Given a database instance D, we refer to its encryption under key K as D_K^E.*

Definition 2.1 uses a single encryption key. To get column-level keys we can, for example, encrypt the concatenation of table and column name using the master encryption key and use the encrypted value as the key for that column [25].

In the cloud DBMS setting, the client stores an encrypted database at the cloud DBMS. The encryption uses a secret key known only to the client and not the cloud DBMS. When using trusted hardware the client key is securely shared with the trusted hardware; this sharing requires the use of Public Key Infrastructure, e.g., as discussed in [5]. Further, for presentation in our examples, we assume that only data is stored encrypted and schema is in clear-text. In practice, schema could be anonymized using opaque identifiers for table and column names.

In practice, not all columns in a database require the same level of security. The column-level encryption allows the client to pick and choose encryption schemes with different security levels for different columns, including leaving some columns in clear-text, as illustrated in Figure 2. The ability to support an arbitrary configuration of column-level encryption is an important design consideration for cloud DBMSes supporting encryption since this provides the client a mechanism to tradeoff security for performance. Both our security model and secure query processing techniques are designed to work for an arbitrary configuration of data-at-rest encryption.

2.2 Security Guarantee

There exists a rich body of work in cryptography formalizing security of encryption schemes. We provide a brief and informal overview here; more details can be found in [18]. Two classic notions of security are *indistinguishability* and *semantic security*. A message encryption scheme is indistinguishable (for an eavesdropping adversary) if an adversary given a ciphertext c, cannot determine if $c = \mathsf{Enc}_k(m_1)$ or $c = \mathsf{Enc}_k(m_2)$ (c is guaranteed to be one of the two). A stronger security notion is *Chosen Plaintext Attack (CPA)*-indistinguishability where an adversary has more power in the form of oracle access to $\mathsf{Enc}_k()$. There exist encryption schemes with a deterministic encryption function that is indistinguishable for an eavesdropping adversary. For CPA-indistinguishability, the encryption function has to be non-deterministic (probabilistic), and there exist well-known constructions based on block ciphers such as AES (the Advanced Encryption Standard) [1]. An encryption scheme is *semantically secure* if an adversary cannot compute any function $f(m)$ given just $\mathsf{Enc}_k(m)$. A classic result in cryptography proves the equivalence of indistinguishability and semantic security.

We adapt semantic security to database encryption schemes. As before, a database encryption scheme E is semantically secure if Eve cannot learn any information about clear-text database D given its encryptions D_K^E. The information Eve seeks is modeled as a *goal* function f that maps databases to binary strings. For example, in Figure 2, Eve learns from the encrypted database that there are equal number of male and female patients, so the encryption scheme is not semantically secure. The fact that Eve cannot compute f is formalized by introducing a *simulator* that learns f with the same success probability, but without access to D_K^E.

We can show that any database encryption that is semantically secure in the strictest sense needs to use a non-deterministic encryption scheme for each column[2]. However, as discussed in Section 2.1, it is important for us to consider weaker configurations of column-level encryptions, and any configuration with even one column using deterministic encryption fails to be semantically secure. We therefore introduce a generalization of semantic security that lets us characterize security of database encryption schemes with less secure column-level encryptions.

The informal description of semantic security so far implicitly assumed that clear-text database D can be drawn from an arbitrary distribution of databases. Our generalization of semantic security involves restricting the space of database distributions. In particular, we introduce a *permit function* Δ that maps database instances to binary strings. A distribution $\mathcal{D}$ over databases is said to be *permitted* by Δ, denoted $\mathcal{D} \models \Delta$, if every database D in the distribution agrees on the value of Δ, i.e., have the same $\Delta(D)$ value. Our generalized definition of semantic security is parameterized by Δ and considers only distributions permitted by Δ. In the following definition, a function $n(k)$ is *negligible* if for every polynomial $p(k)$, for all sufficiently large k, $n(k) < \frac{1}{p(k)}$; e.g., $1/2^k$ is negligible, while $1/k^3$ is not.

DEFINITION 2.2. *A database encryption scheme E is* semantically secure *up to function Δ if for all distributions $\mathcal{D} \models \Delta$, for all goal functions f (independent of complexity), for all polynomial-time adversaries Adv, there is a polynomial-time simulator Sim_Δ such that the following function (of k) is negligible:*

$$|Pr[Adv(D_K^E, 1^k) = f(D)] - Pr[Sim_\Delta(1^k) = f(D)]|$$

Here, the probability is taken over the choice of D, K and the internal coins of the database encryption scheme, Adv and Sim, and $k = |K|$ is the key length.

In the above definition, we can think of Δ as information we are willing to reveal. The simulator implicitly has access to this information since the input distribution is constrained using Δ. If Δ returns column lengths and table cardinalities, then we reduce to classic semantic security. However, by making Δ return more information, we uniformly model the weaker encryption schemes. As an example consider a database consisting of just the `Ailment` table of Figure 2. The encryption scheme used is not semantically secure in an absolute sense since the `Name` column is in plaintext. However, it is semantically secure for $\Delta = \pi_{Name}(Ailment)$.

2.3 Abstraction for QP over Encrypted Data

For the purposes of our security model, we consider an abstraction for query processing over encrypted database models both client-server and trusted hardware-based architectures discussed in Section 1. Our abstraction is shown in Figure 5. It consists of an untrusted (database) system that stores the encrypted database and a trusted component that stores the encryption key and interacts with the untrusted system to execute queries. In general, the trusted component can call the untrusted server multiple times to evaluate a query. The trusted component could be the client itself or a trusted hardware module.

Formally, a query processing abstraction consists of two interacting randomized polynomial-time algorithms—a *trusted* algorithm

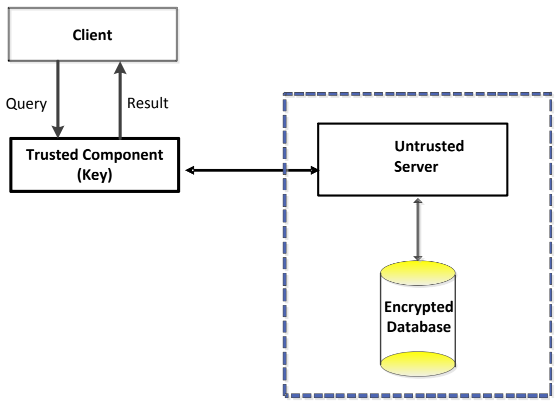


Figure 5: Abstraction for QP over Encrypted Database

$QP_t(K, Q, communication)$ running in the trusted component which is the initiator of the interaction, and an *untrusted* algorithm $QP_u(communication, D)$ running in the untrusted system. The trusted algorithm QP_t has inputs the secret key K, a query Q, and the communication from QP_u. The untrusted algorithm QP_u has inputs the communication from QP_t and encrypted database D_K^E. Given inputs a query Q, a key K and database D_K^E, the query processing system produces its output through the following r-round interaction between the above algorithms.

$$\begin{aligned} O_1 &= QP_t(K, Q) \\ O_2 &= QP_u(D_K^E, O_1) \\ &\dots \\ O_{2r} &= QP_u(D_K^E, O_1, \dots, O_{2r-1}) \\ O_{2r+1} &= QP_t(K, Q, O_1, \dots, O_{2r-1}, O_{2r}) \end{aligned}$$

The number of rounds is required to be polynomial in the input size. The output of query processing for query Q, key K and database D_K^E is denoted $\langle QP_t, QP_u \rangle(K, Q, D_K^E)$. A query processing system is *correct* if $\forall K \forall Q \forall D \Pr[\langle QP_t, QP_u \rangle(K, Q, D_K^E) = Q(D)] = 1$. The above definition of query evaluation is stated for a read-only query. It is straightforward to extend the above definition to handle updates. The only difference between queries and updates is that updates leave the original database modified. We require that updates honor the encryption scheme; therefore, if the database instance D is converted to instance D', then the output of the update is some valid D'^E_K. Henceforth, in the rest of this paper, we treat queries and updates uniformly and refer to them as queries.

EXAMPLE 2.3. *Consider the query $\sigma_{City=NewYork \wedge Gender=Male}(Patient)$ over the encrypted database of Figure 2. One query processing strategy in terms of our abstraction is the following: the trusted component on receiving the query sends the plaintext query $\sigma_{City=NewYork}(Patient)$ to the untrusted system. The untrusted system evaluates the query over encrypted database, which it can since `City` column is stored in clear-text. The result of this query is communicated to the trusted component, which decrypts the Gender field, applies the second filter, and communicates the final result to the client. This query processing strategy incurs a single round of communication. In contrast, the plans described in Examples 1.1 and 1.2 involve multiple rounds of communication.*

3. SECURITY MODEL

In this section, we introduce the security model for formalizing information leakage during query processing that forms the basis

[2]A encryption scheme with multi-message indistinguishability (a slightly weaker notion of security than CPA-indistinguishability).

for specifying a security contract between the cloud DBMS and the client. We state our model in terms of the query processing abstraction of Figure 5. Overall, our model has the following parameters: (1) the database available to adversary Eve in encrypted form, (2) the notion of a query *trace* that formalizes what an adversary can observe as part of the query execution and (3) the notion of a *permit function* that specifies what information the query processing system is allowed to reveal. The goal of the security model is to stipulate that by accessing the query trace, the adversary Eve gains no more information than what can be learned from the encrypted data, except for the permit function.

3.1 Query Trace

A query trace formalizes information made available to adversary Eve through query processing. As noted above, our formalism is developed with reference to the query processing abstraction of Figure 5. We assume that Eve has administrative privileges to the untrusted component and can monitor the communication to and from the trusted component. Formally, the *query trace* includes the following events happening inside the untrusted component: the sequence of instructions executed, the sequence of memory accesses (at every level of the memory hierarchy including disk), and all communication to and from the trusted component.

EXAMPLE 3.1. *Consider the execution of the plan in Figure 3. Its trace includes information about the skeleton of the plan, in what sequence the records were fetched from the disk and for a particular* `Ailment` *record, the fact that the record was sent to TM and the response of the TM for the record. It includes similar information about the other operators.*

We note that a variety of timing related side-channel attacks are possible. For example, string functions could reveal information about input string lengths from the time they take to process their inputs. By not explicitly modeling time as part of the query trace, we are not covering such attacks in our security model. A comprehensive treatment of such attacks is future work.

3.2 Permit Function

Ideally, a query processing system reveals no information about the clear-text database. However, as we will argue in Section 4.1, it is difficult to design an efficient query processing system over that handles arbitrary SQL over large databases and yet does not reveal any information about clear-text. Therefore, for our security model, we introduce the notion of a *permit function* $\chi(Q, D)$ that takes as input a clear-text query Q and a clear-text database instance D, and returns a binary string encoding all information about D that we are willing to reveal to Eve when evaluating Q. Intuitively, a permit function that reveals less information guarantees more security. We illustrate several examples of the permit function.

- The strongest permit function is one that returns the empty string for all Q and D (and reveals nothing).
- In order to allow the query processing system to communicate the query result to the client, we could modify the permit function to return the output size.
- In section 4.1, we discuss what is the strongest permit function that feasibly allows execution of arbitrary SQL queries over large data. We argue that a permit function that allows each *operator* in an execution plan for a query to reveal nothing but its output size is a good candidate.

3.3 Semantic Security of Query Processing

We now discuss the formal security model for query processing over encrypted data. For ease of exposition, we introduce the model for read-only queries and discuss extensions to handle updates later. As before, the goal of the adversary Eve is to learn some information about the clear-text database. Eve has access to the encrypted database. Since a mix of strong and weak encryption schemes can be used to encrypt the data, some information is revealed by the encrypted database itself. When we add query processing to the mix, Eve gets additional access to the query trace. Our formalism stipulates that *independent* of how the base data is encrypted, no extra information is revealed other than what is allowed by the permit function.

The formalism is simulation-based as in Definition 2.2. We would like to stipulate that whatever an adversary learns about the data through the execution trace can be learned by a simulator with access only to the permit function and the information revealed by the encrypted data, but not the trace. As in Definition 2.2, we grant access to the simulator to the information revealed by the encrypted data and to the permit function by constraining the space of database distributions. Given a query Q, a distribution $\mathcal{D}$ over databases is *permitted* by the permit function χ, denoted $\mathcal{D} \models \chi$ if all $D \in \mathcal{D}$, $\chi(Q, D)$ is the same. The formal definition is below.

DEFINITION 3.2. *A query processing system is* semantically secure *upto χ if for all database encryption schemes (Enc, Dec) that are secure up to function Δ, for any query Q, for all distributions $\mathcal{D}$ over databases such that $\mathcal{D} \models \Delta$ and $\mathcal{D} \models \chi$, for all goal functions f (independent of complexity), for all polynomial-time adversaries Adv, there is a polynomial-time simulator Sim such that the following function of k is negligible:*

$$|Pr[Adv(D^E_K, Tr(Q, D), 1^k) = f(D)] - Pr[Sim(1^k) = f(D)]|$$

Here, the probability is taken over the choice of D, Q, K and the internal coins of the database encryption scheme, the query processing system, Adv, and Sim, and $k = |K|$ is the key length.

We illustrate different aspects of the security model using a few examples:

EXAMPLE 3.3. *Any query processing over a clear-text database is trivially secure. Both adversary and simulator can evaluate any function $f(D)$ over database D since they both have access to $D = D^E_K$.*

EXAMPLE 3.4. *Consider the query and query processing strategy described in Example 2.3. This query processing strategy does not reveal any information (i.e., secure for an empty permit function). The simulator can reproduce the query processing trace happening at the untrusted system (since it is a plaintext computation) using her access to encrypted database and can therefore compute any function the adversary can.*

EXAMPLE 3.5. *Consider the query from Examples 2.3 and 3.4 but a different query processing strategy. Here the untrusted system evaluates the query $\sigma_{City=NewYork \wedge Gender=Ry!<4\&}(Patient)$ and communicates the (empty) output to the trusted system, and note that* `Ry!<4&` *is the (deterministic) encryption of the query constant* `Male`*. The trusted system then forwards the result to the client. Surprisingly, this query processing is not secure even for the permit function that reveals the query output size. Eve, who has access to the plaintext query learns that there are no male patients from New York, which the simulator cannot. Recall from Section 1 that the rationale for assuming an adversary with plaintext access is not to rely on query secrecy for overall security.*

EXAMPLE 3.6. *Consider the security discussion in Examples 1.1 and 1.2. We note that the notion of security formalized in our security model (for intermediate result sizes permit function) aligns with the informal discussion of security in these examples.*

We note that the above formulation makes no assumptions regarding the encryption of the query text. Even if Eve has access to the query text, the query processing system is required to reveal no extra information other than the permit function. However, this is not to suggest that the query text need not be encrypted. Encrypting the query text is an essential part of the security of the overall system. Otherwise, for instance if database inserts statements are left in clear-text, the security of the database is seriously compromised. In fact, Definition 3.2 does not require the query processing system to encrypt its queries. We present the full model that compels the query processing system to also encrypt queries in the full version of the paper.

3.4 Database Indistinguishability

In this section, we first adapt the notion of message indistinguishability to database indistinguishability to provide another formalism for security of query processing over encrypted databases. We then establish that in the query processing setting also, semantic security and indistinguishability are equivalent. This shows the robustness of our security model. Our notion of a secure operator and the proofs of security of various operators rely on the notion of database indistinguishability.

The intuition behind our adaptation is similar to classic message indistinguishability. We stipulate that for a given query Q, the query execution trace produced by any two databases are indistinguishable by a polynomially bounded adversary. We accommodate the permit functions Δ and χ by using them to constrain the two pairs.

DEFINITION 3.7. *A query processing system is* database indistinguishable *upto χ if for all data encryption schemes (Enc, Dec) that are secure up to function Δ, for any two databases D_1, D_2 and any query Q such that $\forall i \in \{1,2\} D_i \models \Delta \wedge D_i \models \chi$, for all polynomial-time adversaries Adv, the following function is a negligible function of k:*

$$|Pr[Adv(D^E_{1_K}, Tr(Q, D_1)) = 1] - Pr[Adv(D^E_{2_K}, Tr(Q, D_2)) = 1]|$$

Here, the probability is taken over the choice of Q, K and the internal coins of the database encryption scheme, the query processing system and Adv, and $k = |K|$ is the key length.

We establish the equivalence of the above definition to semantic security.

THEOREM 3.8. *A query processing system is database indistinguishable upto χ if and only if it is semantically secure upto χ.*

3.5 Extension For Updates

Since updates change the state of the database, we have to modify Definitions 3.2 and 3.7 to refer to the state of the database before and after the update. Recall that an update is required to honor the data encryption policy. We therefore modify the above definitions by requiring the state of the database after the update to also be permitted by function Δ. The formal details are presented in the full version of the paper.

3.6 Compositionality and Multiple Queries

We note that in the above discussion, the security model is presented for a single query. In general, multiple queries are run on the system. The question arises what the overall security guarantee is. We show that the security guarantee "composes" under our security model if the underlying query processing system guarantees security under chosen-plaintext attacks. We state a more general version of compositionality where the permit function for different queries is not necessarily the same. The following result models query processing as an operation over a *vector* of records, which is how physical execution takes place even though a database is intuitively a set of records. While the result below is stated for queries, it extends to handle updates in a straightforward manner.

THEOREM 3.9. *(Informal) Suppose the evaluation of queries $Q_i, i \in \{1,2\}$ over database D is CPA-secure for permit functions $\chi_i(Q, D)$. Then, the execution of Q_1 followed by Q_2 is CPA-secure for permit function $\chi_1(Q_1, D) \circ \chi_2(Q_2, D)$ where $\circ$ denotes string concatenation.*

The above result has implications even for a single query. Since relational operators are special cases of queries, it lets us decompose the execution of a single query into operators. Thus, if we design the fixed set of operators that are CPA-secure—we refer to them as *secure* operators, then we can compose them to run arbitrary queries. We will use the above observation in Section 5.

4. SECURE QUERY PROCESSING

In this section, we explore the challenges involved in building a secure DBMS that supports an instantiation of the security model described in Section 3. While solutions that work for a subset of SQL might be acceptable for several scenarios, our goal is to handle the *full* complexity of SQL independent of how the data is encrypted. Given the above requirement, we study what is the strongest permit function we can plausibly support. There are two considerations that govern feasibility. One is performance—while we do expect strong security to come with some overhead, how significant is the overhead? The other consideration is software engineering. Since database systems are complex pieces of software, we have to be concerned about the development cost of incorporating strong security. We therefore restrict ourselves to traditional operator-centric query processing architectures, so query processing proceeds by first generating a plan composed from relational operators and then evaluating the plan. We don't know of other query processing architectures that handle full SQL.

4.1 Choosing A Permit Function

The strongest permit function is one that reveals no information. Consider a query evaluation strategy, exemplified in Example 3.4, where the server treats the encrypted columns as a blob on which no querying is performed [21]. Query evaluation involves evaluating part of the query over plaintext columns in the server, transferring intermediate results to the client, which completes the "remainder" of the query processing. As we argued in Example 3.4, the above strategy reveals no information. There are many applications where the data transfer can be limited, e.g., OLTP applications where data is queried and updated mostly through key lookups. However, in the worst case, the amount of data transferred is unlimited. To illustrate, consider the query used in Example 1.1. The above strategy has to execute a variant of the plan in Figure 4 where the result of the *full* join between `Ailment` and `Patient` is fetched to the client to evaluate the predicate on `Disease`. The communication cost of transferring large amounts of data to the client can be prohibitive. As we have discussed in Section 1, while weaker encryption schemes can reduce the need to ship data in important special cases, they do not completely eliminate the possibility of transferring large amounts of data (e.g., if we have predicates that are neither equality nor range predicates.)

In order to guarantee completeness, we explore the use of trusted hardware (Figure 1) to ensure that all the query processing happens in the cloud server, thereby avoiding the need to ship large amounts of data to the client (other than possibly the query output). In this setting, a permit function that reveals nothing, not even the query output size, intuitively requires the output size to all queries to be equalized by suitable padding. Clearly, such a system would incur a huge performance overhead.

It is not hard to see that any system that seeks to hide information correlated with the query execution time faces similar performance overhead. We could consider intuitively permitting the execution time of queries to be revealed. We can formalize the above intuition by setting the permit function to be a logical model of query execution time such as the total number of intermediate results over all operators in an optimized plan using a standard DBMS[3].

Can we hide individual intermediate result sizes (operator input/output sizes) in a plan? While we do not have a formal proof, we believe the answer is "no" in an engineering sense. Many standard operators are blocking and materialize intermediate state to disk (which is in UM, visible to Eve), and the size of this intermediate state typically reveals the sizes of the operator's inputs. As a concrete example, for a sort-merge join the sum of sizes of the sorted runs reveals the size of the input to the join operator, which is the intermediate result size of the downstream operator. In traditional DBMSes this intermediate state is written contiguously on disk to minimize random seeks. We can argue that hiding the size of the intermediate state would require writing it non-contiguously and therefore incur a significant performance penalty.

We similarly argue that it is difficult to hide the plan for a query, even if all the operators are run inside the TM. Since TM has limited resources (and in particular no disk) the input and output of many operators goes through UM which reveals information about the plan. For example, if TM scans two base tables, then the plan most likely involves a join of the two tables.

Based on the above discussion, we argue that the strongest permit function that we can possibly hope to support consists of the set of intermediate (and final) result sizes in a query plan. We note that the above permit function essentially requires every operator to be optimally secure in that it reveals only its output size. We next review prior work that uses trusted hardware to study if they meet the above permit function.

4.2 Prior Work

We first discuss Transparent Data Encryption (TDE) that is implemented commercially [24, 23]. The idea behind TDE is to encrypt all the data on disk including intermediate results. Query processing is performed on clear-text by decrypting data as it enters the system from disk. TDE as commercially supported does not rely on trusted hardware. However, the encryption key is stored as part of the system which is a potential vulnerability. We discuss a modification to TDE where the encryption key is stored securely in TM. Data from the disk is decrypted by going to TM. We can secure the communication between UM and TM using channel encryption that is separate from data encryption. The above system guards against adversaries who monitor the disk and the communication between UM and TM but not against adversaries who access the contents of main memory. In terms of our model, this corresponds to restricting the query trace to exclude main memory accesses. Even with the above restriction on the trace, some information is revealed such as the disk IOs caused by spilling intermediate results. Furthermore, TDE also reveals disk access patterns.

We next discuss TrustedDB [5] that is the state of the art system based on trusted hardware. In TrustedDB, columns are stored either in clear-text or using semantically strong encryption. The TM runs a light-weight DBMS that communicates with the DBMS running in the UM by pulling data directly from the buffer pool of the UM. Predicates on clear-text are run in the UM and on encrypted data are run in TM. The authors of TrustedDB acknowledge that their system reveals data access patterns. Example 1.2 discusses information disclosure that happens via access patterns. We can see that in order to meet the above permit function in the context of Example 1.2, the filter operator needs to reveal only its output size which requires it to not reveal access patterns.

In summary, prior work using trusted hardware fails to meet the permit function introduced above. We next discuss an architecture that meets the above permit function and discuss its performance implications.

5. SECURE QUERY PROCESSING ARCHITECTURE

We now present an end-to-end conceptual design of a secure query processing system that does not reveal any information outside of intermediate cardinalities. For ease of exposition, we do not consider weaker encryption schemes. Hence, like TrustedDB, every column is either in clear-text or encrypted using CPA-secure encryption. Since we use an operator-centric architecture, our goal is to describe operators that reveal no information other than their output size. As part of this section, we first discuss how we can support relational operators over encrypted columns. Owing to lack of space, we focus on the following operators — filter, sort, foreign-key joins and grouping-aggregation. It is straightforward to extend the operators below to cover other operators such as anti-join and updates. It is well-known that the above operators together can be used to address all of SQL using scan-based plans. We will see that intuitive implementations of the above operators are insecure. The secure operators we present are not free; they increase the number of passes over the data and sometimes rely on the data being randomly permuted. However, the surprising result is that the overhead of securing them is not significant; in fact, their data complexity is the same as that of the original operators. Furthermore, they are based on simple primitives such as random permutations and oblivious sorting that are not difficult to incorporate in a DBMS.

We then discuss indexing where again we show that a straightforward adaptation of B-Tree indexing is insecure. In order to hide access patterns, we leverage prior work on oblivious storage. Throughout, we design our operators such that they use the larger resources available in UM as much as possible.

Finally, we discuss the combination of clear-text and encrypted columns. While we can process standard operators over clear-text as usual securely when the query treats the encrypted columns as a blob, supporting queries that mix and match clear-text and encrypted columns has implications for the physical design of the database which we discuss.

5.1 Scan-Based Operators

We now present our scan-based operators. We use the following convention when presenting operator details: if r denotes a record, then we use $\overline{r}$ to denote the encryption of r. Also recall that UM

[3]A subtlety here is that permit functions are defined over a clear-text database D and query Q. We are therefore referring to the total intermediate result size in a clear-text database. We also note that the plan selected for Q is a "function" of Q and D defined by the optimizer.

Algorithm 1 Secure filter over a randomly permuted stream $T = \overline{r}_1, \ldots, \overline{r}_n$; P is the filter predicate with selectivity $\frac{1}{\alpha}$ and M_t is available memory in TM

```
1: procedure SecureFilter($T, P, \alpha, M_t$)
2:     $OutQueue \leftarrow \phi$
3:     for $i = 1$ to $n$ do
4:         $r_i = \mathsf{Dec}(\overline{r}_i)$
5:         if $r_i$ satisfies filter predicate $P$ then
6:             $\overline{r}'_i \leftarrow \mathsf{Enc}(r_i)$                ▷ Re-encrypt $r_i$
7:             $OutQueue.Enqueue(\overline{r}'_i)$
8:         end if
9:         if $i \geq \alpha\frac{M_t}{2}$ and $\alpha | i$ then
10:            Output $OutQueue.Dequeue()$
11:        end if
12:    endfor
13:    Output remaining records in $OutQueue$
14: end procedure
```

refers to untrusted module and TM to trusted module in our architecture.

5.1.1 Filter

A filter operator evaluates a filter $\sigma_P(T)$ over an input stream of records T and outputs those that satisfy the filter predicate P. Recall that Example 1.2 illustrates why a standard pipelined filter that invokes TM to evaluate a filter and re-encrypts results is insecure since it reveals access patterns. In order to hide patterns, we build upon two intuitive ideas. One is to randomly permute the input. The other is to ensure that the TM produces a filtered record at a *fixed* rate that is a function only of the output size, i.e., the selectivity. In order to guarantee a fixed rate of output, the TM buffers records (recall that we had alluded to the buffering idea in Example 1.2). The algorithm ensures that the buffer size which is limited by the memory of TM is not exceeded. A similar idea is used in [28] in filtering step for ORAM simulations.

For simplicity, we assume that the selectivity of the filter is known. A simple way to get the selectivity is to make an additional pass over T; in the full version we show how this additional pass can be avoided. Further, we assume for simplicity that selectivity is of the form $\frac{1}{\alpha}$ for some integer α.

The secure filter begins by randomly permuting the records in T. This random permutation can be performed in UM and Eve can get full knowledge of the permutation and this does not affect security. We discuss the random permutation step in the full version of the paper. The secure filter iteratively feeds the randomly permuted records to TM. The secure filter logic within TM is shown in Algorithm 1. For any record r_i that satisfies the filter, we re-encrypt the record (using the same client key) and buffer the resulting record $\overline{r}'_i$. After an initialization phase that lasts $\alpha\frac{M_t}{2}$ input records, the buffered records are output from TM at a fixed rate of one record for every α input records. After all the input records have been processed, all remaining buffered records are output from TM. (We can show that there will be exactly $\frac{M_t}{2} - 1$ such buffered records.) The records output from TM comprise the final output of the filter.

The security of the operator follows from the observation that the input and output pattern of records to and from TM depend only on n and α and α can be revealed to Eve since it is simply the ratio of n and the filter output size. Also, since we re-encrypt output records, Eve cannot determine the correspondence between output and input records.

The secure filter of Algorithm 1 *fails* if $OutQueue$ is empty when we try dequeueing in Step 10 and $OutQueue$ uses up all M_t memory when we try enqueueing in Step 7. For example, if all the records passing the filter occur towards the end, $OutQueue$ would be empty when $i = \alpha\frac{M_t}{2}$. The purpose of the random permutation is to ensure that records that satisfy the filter are evenly spread out and make such failure unlikely. We can show that with even moderately large M_t the probability of failure is vanishingly small:

Theorem 5.1. *If $M_t \geq \frac{4 \cdot c}{\alpha}\sqrt{2n \ln n}$, Algorithm 1 succeeds with probability at least* $(1 - 1/n^c)$.

When Algorithm 1 fails, we fall back to a less efficient sort-based implementation of the filter described in the full version.

As an engineering optimization, we could consider storing the base table tuples randomly permuted and avoid random permutation for filters over base tables.

5.1.2 Sort

The sort operator sorts an input stream of records based on some binary comparison function defined over records. A sort operator is used to implement the `ORDER BY` clause in SQL and also as a subprimitive in join and group by. A standard sort that is extended to work with TM is insecure — it is well-known that the sequence of memory locations accessed by merge-sort can reveal information about its input such as the permutation of the input sequence [13].

To get a secure sort operator, we run the external memory *oblivious sorting* algorithm of [13]; an important detail is that when records are written out of TM, they are re-encrypted to hide correspondence between records that enter and those that emerge out of TM. An oblivious sorting algorithm by definition has the property that its data access patterns are independent of data values; this property, combined with CPA-secure encryption, ensures that the resulting sort operator is secure. The oblivious sorting algorithm of [13] has the same complexity as regular disk-based sort-merge join.

5.1.3 Joins

We first observe that the standard sort-merge and hash join operators are insecure. A sort-merge join algorithm is insecure for the same reason that a standard sort is insecure. Similarly, a standard hash join is insecure since it reveals the join graph (for each record in the probe side, how many records it joins with in the build side.)

The intuition behind the secure join operator we present is as follows. We could consider merely replace the sort step in a sort-merge join with a secure sort. However, the merge step still leaks the join graph. In order to address this issue, we securely sort the union of the two inputs while remembering for each tuple what relation it came from. Since the result of the above sort places joining records together, the TM can with some buffering return the joined records (re-encrypted.)

The algorithm between tables R and S is shown in Algorithm 2, assuming R is the table with the key. Steps 2-8 computes a standard union. For each tuple, the table from which it came is remembered using column `Id`. Bit 0 corresponds to R tuples and a bit 1 to S. If R and S tuples have different lengths, we use padding to ensure tuples in $\mathcal{U}$ have the same length. Next, we secure-sort $\mathcal{U}$ on $\langle A, Id \rangle$ (Step 9); by using Id in the sort attribute list, we ensure that if an R tuple and an S tuple agree on their A value, the (unique) R tuple occurs before in the sort ordering. Next, we implement the "merge" step by iterating over the tuples in $\mathcal{U}$. We can show that any S tuple $\langle s, 1 \rangle$ in $\mathcal{U}$ joins with the most recent R tuple in $\mathcal{U}$ (stored in $lastR$) or does not join with any R tuple. We use this property to generate $\langle r, s \rangle$ tuples in the join output (Step 17). To hide access patterns, we produce dummy output tuples when we read an R tuple (Step 14) or an S tuple that does not produce a join output (Step 19). The dummy tuples are removed using a secure filter (Step 23).

Algorithm 2 Secure foreign key join of $R = \overline{r}_1, \dots, \overline{r}_n$ and $S = \overline{s}_1, \dots, \overline{s}_m$ on attribute A.

```
1:  procedure SecureJoin($R, S, A$)
2:      $\mathcal{U} \leftarrow \phi$                                   ▷ Intermediate union stream
3:      for all $\overline{r}_i$ in $R$ do
4:          Append $\overline{\langle r_i, 0\rangle}$ to $\mathcal{U}$.
5:      endfor
6:      for all $\overline{s}_i$ in $S$ do
7:          Append $\overline{\langle s_i, 1\rangle}$ to $\mathcal{U}$.
8:      endfor
9:      Secure sort $\mathcal{U}$ on $\langle A, Id\rangle$
10:     $\mathcal{J}_d \leftarrow \phi$                                  ▷ Intermediate join stream
11:     $lastR \leftarrow$ null
12:     for all $\overline{u}$ in $\mathcal{U}$ do
13:         if $\overline{u} = \overline{\langle r, 0\rangle}$ then
14:             $lastR \leftarrow r$; Append $\overline{\langle dummy\rangle}$ to $\mathcal{J}_d$
15:         else                                                         ▷ Assert: $\overline{u} = \overline{\langle s, 1\rangle}$
16:             if $lastR[A] = s[A]$ then
17:                 Append $\overline{\langle r, s\rangle}$ to $\mathcal{J}_d$
18:             else
19:                 Append $\overline{\langle dummy\rangle}$ to $\mathcal{J}_d$
20:             end if
21:         end if
22:     endfor
23:     Output $\mathcal{J}_d$ with $\overline{\langle dummy\rangle}$ removed (secure-filter).
24: end procedure
```

Algorithm 3 Grouping and `COUNT(*)` aggregation of $R = \overline{r}_1, \dots, \overline{r}_n$ with a single grouping attribute A.

```
1:  procedure SecureGroupAggr($R, A$)
2:      $R_{sort} \leftarrow$ secure sort of $R$ on $A$
3:      $curA \leftarrow$ null
4:      $curCount \leftarrow 0$
5:      $\mathcal{G}_i \leftarrow \phi$                                  ▷ Output with dummy records
6:      for all $\overline{r}$ in $R_{sort}$ do
7:          if $r[A] = curA$ then
8:              $curCount \leftarrow curCount + 1$
9:              Append $\overline{\langle dummy\rangle}$ to $\mathcal{G}_i$
10:         else
11:             Append $\overline{\langle curA, curCount\rangle}$ to $\mathcal{G}_i$
12:             $curA \leftarrow r[A]$
13:             $curCount \leftarrow 0$
14:         end if
15:     endfor
16:     Output $\mathcal{G}_i$ with $\overline{\langle dummy\rangle}$ removed (secure filter)
17: end procedure
```

For the security of the join operator note that the input and output patterns of the union, secure sort, and the merge step do not depend on data values in R and S. Further, encryption ensures that at the end of the sort step, Eve cannot find the correspondence between tuples in $\mathcal{U}$ and the input tuples in R and S. Combining both observations, we can prove security of the operator. Note that at the end of the union step, Eve can determine which tuples in $\mathcal{U}$ came from R and which, from S. Given this, we can improve the efficiency of the overall join operator by computing union in UM and slightly modifying the sort step to encrypt `Id` column and perform padding.

Overall, the secure join has the same time and data complexity as traditional sort-merge join. The algorithms for general (natural and theta) joins are quite involved and outside the scope of this paper. They are presented in [4] with full empirical evaluation.

5.1.4 Grouping and Aggregation

We present a secure group by and aggregation operator for the special case of a single grouping attribute and `COUNT(*)` aggregate. Our algorithm can be generalized to handle more general grouping and other standard aggregation functions including `MIN`, `MAX`, `COUNT`, `AVG`, `SUM`, and `MEDIAN`. Our algorithm can also be adapted to get a secure duplicate elimination operator.

Algorithm 4 B-Tree Point Lookup procedure with key a. The index T is over $R.A$

```
1:  procedure BTreeLookup($T, a$)
2:      $\overline{bid} \leftarrow \mathsf{Enc}(T.Root)$                 ▷ Encr. Block Id of Root
3:      $\overline{curBlock} \leftarrow Read_{oss}(\overline{bid})$
4:      for $level = 1$ to $T.Height - 1$ do
5:          $\overline{bid} \leftarrow$ Search($\overline{curBlock}, a$)     ▷ Encr. child block id
6:          $\overline{curBlock} \leftarrow Read_{oss}(\overline{bid})$
7:      endfor
8:      $\mathcal{R} \leftarrow \phi$                                    ▷ Buffered output
9:      $curBlock \leftarrow \mathsf{Dec}(\overline{curBlock})$
10:     while $curBlock.First \leq a$ do
11:         $\mathcal{R} \leftarrow \mathcal{R} \cup \{\mathsf{Enc}(r) : r \in curBlock \wedge r[A] = a\}$
12:         $\overline{bid} \leftarrow \mathsf{Enc}(curBlock.Next)$
13:         $\overline{curBlock} \leftarrow Read_{oss}(\overline{bid})$
14:     end while
15:     Output $\mathcal{R}$
16: end procedure
```

Traditional sort-based grouping and aggregation with the sort step replaced with a secure sort step is not secure since it reveals the size of each group. Our implementation is a slight modification and is shown in Algorithm 3. We begin by securely sorting input stream R on grouping attribute A (Step 2). As in traditional aggregation, we scan the sorted stream and compute the counts of each group. The traditional aggregation produces one output tuple per group after the last input tuple belonging to the group has been processed (Step 11). Our modification is to produce dummy output tuples for the other input tuples of a group as well (Step 9). A secure filter is used to remove dummy tuples and get the final output (Step 16). The security of the operator follows since the input output pattern of the operator is independent of the contents of R, and the overall time and data complexity is the same as traditional group by aggregation.

5.2 Indexing Encrypted Columns

We now discuss building a secure B-tree index on attribute A of table R. We focus on the lookup operation and defer a discussion of updates and index construction, which is quite intricate, to the full version of the paper. For simplicity, we assume the height of the B-tree is public knowledge; this is not a serious information leak since the height can be determined to within an additive error of 1 solely from the cardinality of R and the order of the B-tree.

5.2.1 Insecurity Of Standard Lookup

We first show that the standard B-Tree lookup adapted to using TM is *not* secure. This procedure is shown in Algorithm 4. This is a standard B-tree lookup algorithm except that the binary search within a non-leaf block to identify the next block to visit is done within TM. Also, the scan of the chained leaf blocks to identify output records happens within TM. (All the blocks of the B-tree are stored encrypted.) Although Eve observes only encrypted bits entering and leaving TM, the lookup operation as described is not secure as illustrated in the following example:

Example 5.2. *Consider a B-tree over* `Employee.Age`*. The first lookup involves* ($Age = 34$) *and this touches three leaf blocks. Let the block ids of these blocks be* bid_1, bid_2 *and* bid_3*. Since the disk is in UM, Eve knows these ids. The second lookup involves* ($Age = 36$) *and this touches two blocks with ids* bid_3 *and* bid_4.

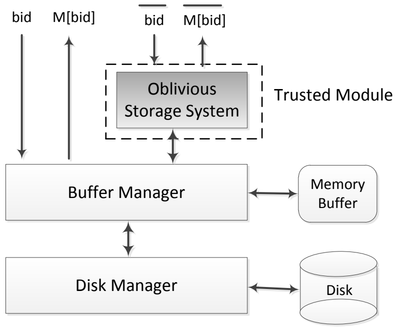


Figure 6: Storage subsystem

From this, Eve infers that any record with $Age = 35$ is stored in a single disk block with id bid_3; so she learns an upper bound (= order of the B-tree) on the number of records with $Age = 35$.

5.2.2 Secure Index

To get a secure B-tree lookup procedure, we use the same procedure but rely on a special storage system layer called *oblivious storage system (OSS)*. We next describe OSS functionality and discuss how we get secure B-tree index lookups with this functionality. The implementation of OSS is discussed in Section 5.2.3.

Oblivious Storage System (OSS) functionality: The traditional storage subsystem in a DBMS consists of a buffer manager which provides indexed read/write access to blocks on the disk, with some main memory buffer pool caching; given a block id bid, it reads/writes contents of the block identified by bid (see Figure 6). The OSS layer sits on top of buffer manager. It provides the same functionality except that the input blocks ids are encrypted using CPA-secure encryption (so Eve does not learn the block ids requested). Further, for each block access, OSS layer makes randomized block read/write requests to the buffer manager such that the distribution of read/write requests is the same, independent of the block accessed. The OSS itself is implemented within TM (see Section 5.2.3) and Eve does not get any information about the sequence of block id accesses to OSS by observing the physical block accesses.

In Algorithm 4, the subscript oss in $Read_{oss}$ emphasizes that index block reads go through OSS. With this modification any sequence of index block accesses results in physical block accesses that appears “random” to Eve, implying security of the lookup procedure. In particular, the attack described in Example 5.2 does not work since Eve does not know that the common block bid_3 was accessed for the two lookup operations. In terms of OSS accesses, the complexity of index lookup is identical to traditional B-tree lookup. But the OSS itself imposes a performance cost as discussed in Section 5.2.3.

5.2.3 Oblivious Storage (OSS)

The OSS functionality described above is isomorphic to well-studied *Oblivious RAM (ORAM)* functionality, so we can use any known solution of ORAM to implement OSS. Briefly, these solutions involve shuffling physical blocks around and making additional “spurious” disk accesses to hide the actual access pattern. Note that above strategy implies that the block ids that exist above OSS layer are now *virtual* and the block corresponding to a given id is stored in different physical locations at different times.

5.2.4 Overheads

OSS functionality comes at a cost. The current best solution results in an overhead of $O(\log M \log\log M)$ [29] physical block accesses for each block access using OSS; here M is the number of blocks being managed by OSS. Also, under some reasonable assumptions $\Omega(\log M)$ [12] is a lower-bound.

However, relying on OSS leads to a loss of spatial and temporal locality of reference, except for the important special case of full scans which can bypass OSS. For example, in a traditional DBMS, we get the benefits of sequential scans not only for full scans but also for large range and point scans. However, lookups of large ranges that go through OSS result in a large number of random seeks since every base record access becomes a random seek. Since large range scans are common in several database workloads, secure indexing does carry a significant cost in the worst case. Similarly, the effectiveness of the buffer pool is reduced. We note that there exists a fundamental connection between secure indexing and ORAM since we can implement ORAM using a secure index. This implies that a more efficient end-to-end secure indexing solution would imply a better algorithm for ORAM.

5.2.5 Engineering Optimizations

There exist a few database specific engineering optimizations that can be used to reduce the costs associated with OSS. The intuition is to have OSS managed smaller collections of blocks called *namespaces* such that access patterns need to be hidden only within a namespace and not across.

The above optimization yields two benefits. One is that the number of physical block accesses for each block access using OSS is a function of M, the number of blocks managed by OSS. In this context, a natural candidate for namespaces are blocks corresponding to the same level in a B-tree, since revealing that block corresponds to a particular level in the B-tree does not violate security. In particular, the root block of a B-tree can be in a namespace by itself, implying that we do not need to incur OSS overheads to access it. Second, the data within a namespace can be fully sequentially scanned thereby reducing the loss of spatial locality.

5.3 Clear-Text + Encrypted Columns

Above, we have described operators over encrypted columns. As noted in Section 3, if we treat the encrypted columns as a blob, then traditional operators over clear-text are secure. We now discuss the issues that arise when combining processing over clear-text and encrypted columns. We note that the secure operators described above encrypt their results *even if* parts of the input are in clear-text. For example, in the database shown in Figure 2, a secure filter over `Ailment` with a predicate on `Disease` encrypts the `Name` field in its output. Therefore, if we modify the plan shown in Figure 3 to replace a filter with a secure filter, the subsequent join has to operate over encrypted names.

In fact, the above issue has implications for the physical design of the database, which we illustrate through two examples. First, suppose we have an index on the `Name` column of the `Patient` table. Since `Name` is in clear-text, we can use a standard B-Tree index. However, the above index cannot be used for the above join between `Ailment` and `Patient`. In order to perform an index-nested loops join, we would need to keep a separate copy of the `Patient` table where all fields are encrypted and build a B-Tree going through OSS as described above.

In general, whenever we build an index on an encrypted column and the base table has clear-text columns, if we access the clear-text columns directly, we reveal information. Therefore, indexing encrypted columns requires us to store a separate copy of the data fully encrypted. However, one such encrypted copy of the data suffices for all indexes.

5.4 Query Optimization

We now briefly discuss the implications of the operators presented above on query optimization. The goal of query optimization is to find the optimal secure plan. Based on our security model and Theorem 3.9, a plan is secure if it is composed of secure operators. Checking whether a physical operator is secure is a function solely of the encryption of the operator and its input encryption. If the operator operates over encrypted columns, the physical operator has to be one of the operators introduced earlier in this section. Otherwise, we can use a standard relational operator. The details of costing secure plans and building statistics securely are beyond the scope of this paper and subject for future work.

6. CONCLUSIONS

Encryption is an important part of supporting database as a service. In this paper, we study the security of querying encrypted data. We show that securing the data-at-rest is insufficient — we need a principled way to handle information leakage during query processing. We address two important aspects to this problem: 1) a security model to precisely specify what information is permitted to be disclosed and 2) the design of a query engine catered towards the above model for a strict permit function that only allows the intermediate cardinalities of a query plan to be disclosed. The main takeaway is the fact that secure query processing is feasible (without extensive changes to an existing query engine) but comes at a price — in particular, the cost of secure indexing can be non-trivial.

Several important issues remain to be addressed. In addition to carefully engineering the secure operators, it is important to augment the current design of a secure query engine with mechanisms for navigating the security-performance tradeoff — exploring such mechanisms as well as other interesting permit functions for a query engine are interesting avenues of future work. Finally, we note that there are other important dimensions to security such as the security of transaction processing and application security that also remain to be addressed in future work.

7. REFERENCES

[1] Advanced Encryption Standard. `http://en.wikipedia.org/wiki/Advanced\_Encryption\_Standard`.

[2] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order-preserving encryption for numeric data. In *SIGMOD Conference*, pages 563–574, 2004.

[3] Amazon Corporation. Amazon Relational Database Service. http://aws.amazon.com/rds/.

[4] Arvind Arasu, Raghav Kaushik, and Ravi Ramamurthy. Secure joins using trusted hardware. Under preparation.

[5] Sumeet Bajaj and Radu Sion. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In *SIGMOD Conference*, pages 205–216, 2011.

[6] Mihir Bellare, Marc Fischlin, Adam O'Neill, and Thomas Ristenpart. Deterministic encryption: Definitional equivalences and constructions without random oracles. In *CRYPTO*, pages 360–378, 2008.

[7] Christophe Bobineau, Luc Bouganim, Philippe Pucheral, and Patrick Valduriez. PicoDMBS: Scaling Down Database Techniques for the Smartcard. In *VLDB*, 2000.

[8] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *CRYPTO*, pages 578–595, 2011.

[9] Privacy Rights Clearinghouse. Chronology of data breaches. `http://www.privacyrights.org/data-breach`.

[10] Carlo Curino, Evan P. C. Jones, Raluca A. Popa, Nirmesh Malviya, Eugene Wu, Samuel Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational cloud: a database service for the cloud. In *CIDR*, pages 235–240, 2011.

[11] Craig Gentry. Computing arbitrary functions of encrypted data. *Commun. ACM*, 53(3):97–105, 2010.

[12] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[13] Michael T. Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *SPAA*, pages 379–388, 2011.

[14] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, et al. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 52(5), 2009.

[15] Bijit Hore, Sharad Mehrotra, and Hakan Hacigümüs. Managing and querying encrypted data. In *Handbook of Database Security*, pages 163–190. 2008.

[16] IBM Corporation. IBM InfoSphere Guardium Data Encryption for DB2 and IMS Databases. `http://www-01.ibm.com/software/data/db2imstools/db2tools/ibmencrypt/`.

[17] IBM Corporation. IBM Systems cryptographic hardware products. `http://www-03.ibm.com/security/cryptocards/`.

[18] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC Press, 2007.

[19] Kevin Prince. Health care data security breaches in the U.S. *SC Magazie For IT Security Professionals*, 2008.

[20] Jinyuan Li, Maxwell N. Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (sundr). In *OSDI*, pages 121–136, 2004.

[21] Mehrotra, S. and Tsudik, G. and others . Database As Service. `http://www-db.ics.uci.edu/pages/research/das/index.shtml`.

[22] Microsoft Corporation. SQL Azure. http://www.windowsazure.com/en-us/home/features/sql-azure/.

[23] Microsoft Corporation. SQL Server Encryption. `http://technet.microsoft.com/en-us/library/bb510663.aspx`.

[24] Oracle Corporation. Transparent Data Encryption. `http://www.oracle.com/technetwork/database/options/advanced-security/index-099011.html`.

[25] R. A. Popa, C. M. S. Redfield, N. Zeldovich, et al. Cryptdb: protecting confidentiality with encrypted query processing. In *SOSP*, pages 85–100, 2011.

[26] Smart Cards. `http://en.wikipedia.org/wiki/Smart\_card`.

[27] National Vulnerability Database. `http://web.nvd.nist.gov/view/vuln/statistics`.

[28] Peter Williams and Radu Sion. Usable pir. In *NDSS*, 2008.

[29] Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *ACM Conference on Computer and Communications Security*, pages 139–148, 2008.