Information Flows in Encrypted Databases

Kapil Vaswani
kapilv@microsoft.com

Ravi Ramamurthy
ravirama@microsoft.com

Ramarathnam Venkatesan
venkie@microsoft.com

Abstract

In encrypted databases, sensitive data is protected from an untrusted server by encrypting columns using partially homomorphic encryption schemes, and storing encryption keys in a trusted client. *However, encrypting columns and protecting encryption keys does not ensure confidentiality - sensitive data can leak during query processing due to information flows through the trusted client. In this paper, we propose SecureSQL, an encrypted database that partitions query processing between an untrusted server and a trusted client while ensuring the absence of information flows. Our evaluation based on OLTP benchmarks suggests that SecureSQL can protect against explicit flows with low overheads (*$<$ 30%*). However, protecting against implicit flows can be expensive because it precludes the use of key databases optimizations and introduces additional round trips between client and server.*

1. Introduction

The old adage that a chain is only as strong as its weakest link describes the state of data security in public cloud platforms. While encryption can protect data in cloud storage and during transit to/from the cloud, data appears in cleartext in main memory of untrusted servers during processing. In the absence of defenses like the firewall, this window of vulnerability is an alluring target for malicious cloud administrators and malware. It is therefore not surprising that applications which handle sensitive information are usually not deployed on public cloud platforms.

Trusted clients. In the absence of practical schemes for fully homomorphic encryption [20], we consider a recently proposed computational model for ensuring data confidentiality based on encrypted databases and *trusted clients* [17, 22, 25]. In encrypted databases, sensitive data is protected from an untrusted server by encrypting columns using partially homomorphic encryption (PHE) schemes, and storing encryption keys in a *trusted client*. For example, deterministic encryption schemes (e.g. AES in ECB or CBC mode with fixed initialization vector [1]) permit equality checks on encrypted data. Therefore, operations such as equi-joins, groupings, and set union/intersection can be performed without requiring encryption keys. Similarly, the Paillier cryptosystem supports addition of encrypted values. All other operations for which efficient homomorphic schemes are not known are delegated to a trusted client, a special node hosted in a trusted environment, potentially outside the public cloud. The trusted client has access to encryption keys and performs computations that cannot be performed on encrypted data.

Information flows. Prior work (including CryptDB [22]) raises the vision of building an encrypted database using trusted clients and PHE. However, a key problem with this model is the lack of a simple, strong security property that can be enforced without a significant loss in performance. In prior work, security is based on the premise that the server does not have access to encryption keys. *However, protecting encryption keys does not guarantee confidentiality*. Prior research [15] shows that confidentiality can be achieved using a combination of semantically secure, randomized encryption [16] and information flow security. Unfortunately, in an encrypted database, randomized encryption negates critical optimizations such as indexes, and forces the database to offload almost all computation to the trusted client. Therefore, selective use of weaker encryption schemes (such as deterministic or order-preserving encryption) is unavoidable.

However, using weaker encryption schemes not only weakens security of columns encrypted using those schemes, but it can also reveal information about other columns due to information flows. Consider a simple query `INSERT INTO T (B) SELECT A FROM T`, where *A* and *B* are encrypted using randomized and deterministic encryption schemes respectively. In the trusted client model, this query can be executed by retrieving values from column *A* from the server, decrypting them on the client, encrypting the values using a deterministic encryption algorithm, and writing the values to column *B*. Notice that this query leaks the relative frequency of values in column A even though the keys are always protected! (See section 2 for more subtle examples). The threat models defined in prior work [22, 25] do not consider such information flows.

In this paper, we explore this security-performance tradeoff. We define a security property that prevents such leaks while permitting the use of weaker encryption schemes. Informally, our security property prohibits *information flows* from an encrypted column to other columns encrypted using weaker encryption schemes. This property forms a strong contract between the database and the developer. For example, if this property is enforced, the query described above will be not be permitted unless both columns are encrypted using sufficiently strong encryption schemes.

Next, we describe the design of SecureSQL, an encrypted database which enforces this property. Figure 1 shows the architecture of SecureSQL. In SecureSQL the trusted client is a "empty" database (referred to as the *shell database*) which

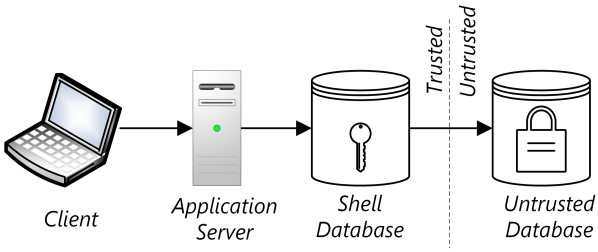

Figure 1: Architecture for databases that use partially homomorphic encryptions and trusted client for data confidentiality

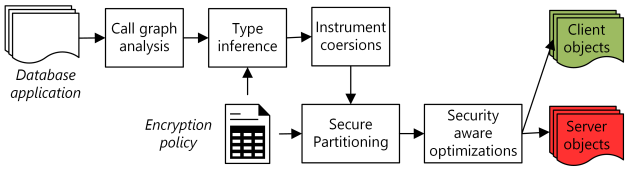

Figure 2: The tool chain for partitioning database applications. The input to the tool chain are a database application and an encryption policy.

stores encryption keys and performs residual query processing. Applications connect and run queries against the shell database, which orchestrates query processing with the server (using a commonly available database abstraction for distributed query processing known as *linked server* [6]) and returns cleartext results to the application.

The core component of SecureSQL is a query compiler for T-SQL (Figure 2). The inputs to the compiler are a database application consisting of a set of queries and stored procedures, and an *encryption policy*, which specifies the set of columns to be encrypted, and the encryption type, encryption algorithm and encryption key for each column. The compiler partitions queries and stored procedures into client and server components while preserving semantics of the application *and* ensuring the absence of insecure information flows. The compiler is based on a type system with knowledge of the computational capabilities of partially homomorphic encryption schemes and tracks information flows. The compiler also supports several optimizations for efficiently partitioning queries that are beyond the scope of conventional database optimizers.

We have evaluated SecureSQL (Section 5) using TPC-C, a standard OLTP benchmark, and a real world employee performance management application. Our evaluation suggests that SecureSQL can guarantee absence of explicit information flows with reasonable performance overheads ($< 30\%$). However, if absence of implicit flows is desired, or if computation on the critical path cannot be performed on encrypted data, performance can degrade (by as much as two orders of magnitude). Factors that contribute to the loss in performance include the inability to use indexes on columns encrypted using semantically secure encryption and additional round trips and data transfers between the client and the server. Users must therefore choose their encryption policy wisely.

The rest of this paper is organized as follows. We define the threat model for systems based on a trusted client in Section 2. We reduce the problem of securely partitioning an application into two sub-problems. The first problem is to rewrite the application so that its semantics are preserved when columns in the database are encrypted while ensuring the absence of information flows. We describe a type system that achieves this in Section 3. The second problem is to partition the rewritten application between the client and server while preserving information flow safety (Section 4). We present an evaluation in Section 5 and conclude in Section 6.

2. Threat model

We wish to protect sensitive data from an honest-but-curious adversary who has access to contents of the database and all queries. The adversary can observe the state of the server on disk, in memory and all communication over the network. The adversary, however, does not have access to encryption keys, which are stored on the trusted client. SecureSQL does not protect against *active* adversaries who can tamper with code and data. While certain kinds of integrity attacks can be detected using authenticated encryption [2], ensuring end-to-end integrity of query processing is an open problem beyond the scope of this paper. We also preclude adversaries who exploit side channels such as size of inputs and results, address traces, timing, and power consumption.

In SecureSQL, the adversary can gain information about sensitive columns due to (a) use of weaker encryption schemes which are not semantically secure, and (b) information flows. As discussed earlier, weaker encryption schemes are a core component of encryption databases - they allow computation to be offloaded to the server, and make use of database optimizations such as indexes. To permit these schemes, we must weaken the threat model, and consider any information revealed due to weak encryption as prior knowledge known to the adversary. Specifically, for deterministic encryption, we assume that the adversary already knows whether two cipher text values correspond to the same cleartext. In cryptographic proofs of security, this is achieved by restricting the adversary from encrypting the same value more than once; see [13, 9] for formal definitions of security of deterministic encryption. Similarly, we assume that the adversary is already knows the relative ordering of values in a column encrypted using order-preserving encryption. Our goal is to ensure that query processing does not reveal any *additional* information.

The second source of information leakage is insecure information flows through the trusted client. We illustrate such leakage using an example. Consider the TPC-C benchmark [7], an application that maintains customer, inventory, and order processing information. Figure 3 shows a stored procedure derived from TPCC. This procedure records payments received from a customer. For this stored procedure, we can define an encryption policy as follows. Since customer information is sensitive, we should ideally encrypt all personally identifiable information (PII) in the customer table (such as names, address details, account balance and credit rating) using randomized

```
1  CREATE PROCEDURE [dbo].[PAYMENT]
2  @c_w_id INT, @h_amount NUMERIC(6,2), @c_last CHAR(16)
3  AS BEGIN
4   BEGIN TRANSACTION;
5    UPDATE dbo.CUSTOMER
6    SET @c_id = C_ID,
7     @c_first =  C_FIRST, @c_credit = C_CREDIT,
8     @c_balance = C_BALANCE = C_BALANCE + @h_amount,
9    WHERE CUSTOMER.C_W_ID = @c_w_id
10   AND CUSTOMER.C_LAST = @c_last;
11
12   INSERT dbo.HISTORY (H_C_ID, H_C_BALANCE)
13   VALUES (@c_id, @c_balance)
14
15   IF @c_credit = 'BC'
16    UPDATE dbo.CUSTOMER
17    SET C_DATA = @TIMESTAMP + C_DATA
18    WHERE CUSTOMER.C_W_ID = @c_w_id
19    AND CUSTOMER.C_LAST = @c_last;
20
21   SELECT @c_id AS N'@c_id', @c_first AS N'@c_first',
22    @c_last AS N'@c_last', @c_credit AS N'@c_credit',
23    @c_balance AS N'@c_balance',
24  COMMIT TRANSACTION;
25 END
```

Figure 3: A T-SQL procedure derived from TPC-C

encryption. However, the column CUSTOMER.C_BALANCE is involved in an addition, and columns CUSTOMER.C_LAST and CUSTOMER.C_CREDIT are used in equality checks. Since randomized encryption prohibits these operations, we can use Paillier encryption (which is semantically secure) for C_BALANCE and deterministic encryption for C_LAST and C_CREDIT. Deterministic encryption also allows the query engine to maintain and use an index on these columns. We can leave all other non-PII columns (such as inventory and order information) unencrypted.

While this policy appears to protect all sensitive information, it does not guarantee confidentiality (even under the relaxed threat model discussed above).

- There is an explicit flow from CUSTOMER.C_BALANCE to HISTORY.C_BALANCE, which is not encrypted. The flow reveals account balances even if the adversary does not have access to encryption keys. Preventing this flow requires a simple change to the policy - encrypt HISTORY.C_BALANCE using Paillier encryption.
- There is an implicit flow from CUSTOMER.C_CREDIT to CUSTOMER.C_DATA in the update query at line 17 because C_DATA (which is not encrypted) is conditionally updated based on C_CREDIT. Due to this flow, an adversary can potentially infer whether a customer has a bad credit history. This flow can be prevented by encrypting C_DATA using deterministic or randomized encryption.

Clearly, these flows are not desirable. We now define a security property that disallows insecure information flows.

Definition 2.1 *A application preserves confidentiality if there are no information flows from a column/variable encrypted using a stronger encryption scheme to a column/variable encrypted using a weaker encryption scheme.*

Note that this definition assumes the presence of a partial order over encryption schemes based on their relative strength.

```
1  --Server
2  CREATE PROCEDURE [dbo].[__Closure]
3  @c_w_id INT, @h_amount VARBINARY(2048), @c_last VARBINARY(256)
4  AS BEGIN
5   SELECT @pubkey = ...
6   BEGIN TRANSACTION;
7    UPDATE dbo.CUSTOMER
8    SET @c_id = C_ID,
9     @c_first =  C_FIRST, @c_credit = C_CREDIT,
10    @c_balance = C_BALANCE =
11     dbo.PaillierAdd(C_BALANCE, @h_amount, @pubkey)
12   WHERE CUSTOMER.C_W_ID = @c_w_id
13    AND CUSTOMER.C_LAST = @c_last;
14
15   INSERT dbo.HISTORY (H_C_ID, H_C_BALANCE)
16   VALUES (@c_id, @c_balance)
17
18   IF @c_credit = 0x002057E9A8865AAA7D59DA69AD...
19    UPDATE dbo.CUSTOMER
20    SET C_DATA = @TIMESTAMP + C_DATA
21    WHERE CUSTOMER.C_W_ID = @c_w_id
22    AND CUSTOMER.C_LAST = @c_last;
23
24   SELECT @c_id AS N'@c_id', @c_first AS N'@c_first',
25    @c_last AS N'@c_last', @c_credit AS N'@c_credit',
26    @c_balance AS N'@c_balance'
27  COMMIT TRANSACTION;
28 END
29
30 -- Trusted Client
31 CREATE PROCEDURE [dbo].[PAYMENT]
32 @c_w_id INT, @h_amount NUMERIC(6,2), @c_last CHAR(16)
33 AS BEGIN
34  SELECT @key = ... // private key
35  SELECT @pubkey = ... // public key for paillier
36  SELECT
37   @enc_h_amount = dbo.AEncrypt(@h_amount, @key, @pubkey),
38   @enc_c_last = dbo.DEncrypt(@c_last, @key),
39
40  EXEC [SERVER].[tpcc].dbo.__Closure
41   @enc_c_w_id, @c_d_id,@enc_c_amount, @enc_c_last,
42   out @c_id, out @c_first, out @c_last,
43   out @c_balance, out @c_credit
44
45  SELECT @c_id AS @c_id,
46   dbo.RDecrypt(@c_first, @key) AS @c_first,
47   dbo.DDecrypt(@c_last, @key) AS @c_last,
48   dbo.DDecrypt(@c_credit, @key) AS @c_credit,
49   dbo.ADecrypt(@c_balance, @key, @pubkey) AS @c_balance,
50 END
```

Figure 4: A partitioned stored procedure which preserves semantics and absence of explicit flows.

Also note that this definition includes all columns and variables such as parameters passed to any procedures executed on the server, and results of intermediate query processing. We can extend this definition to an application partitioned into a trusted client and an untrusted server.

Definition 2.2 *A partitioned application preserves confidentiality if there are no information flows from a column/variable encrypted using a stronger encryption scheme to a column/variable stored on the server encrypted using a weaker encryption scheme.*

Unlike Definition 2.1, this property permits information flows from encrypted columns to cleartext variables on the trusted client. This allows the trusted client to decrypt values retrieved from encrypted column, perform computations on those values, encrypt the results, and store them on the server *as long as resulting values are encrypted using a sufficiently*

$$
\begin{array}{llllllll}
n & \in & \text{Names} & x & \in & \text{Values} & v & \in & \text{Variables} \\
f & \in & \text{Functions} & T & \in & \text{Tables} & c & \in & \text{Columns} \\
k & \in & \text{Keys} & pk & \in & \text{PublicKeys} & S & \in & \text{Server}
\end{array}
$$

$$
\begin{array}{lll}
e & ::= & x \mid v \mid t.c \mid [n_1 : e_1, ..., n_m : e_m] \mid e_1 = e_2 \mid e_1 < e_2 \\
& & \mid e_1 + e_2 \mid fe \mid \sigma_{e_1}(e_2) \mid \pi_{n_1,...,n_m}(e) \mid e_1 \cup e_2 \mid e_1 \backslash e_2 \mid e1 \times e2
\end{array}
$$

$$
\begin{array}{lll}
s & ::= & e_1 := e_2 \mid s_1; s_2 \mid \text{if } e_1\ s_1\ s_2 \mid \text{select}_{n_1,...,n_m}(e) \\
& & \mid \text{insert}_{n_1,...,n_m}(t)\ e \mid \text{update}_{n_1,...,n_m}(t)\ e \mid \text{delete}(t)\ e
\end{array}
$$

$$
p \quad ::= f(v).s \mid p\ p
$$

$$
\begin{array}{lll}
e_t & ::= & r\_encrypt(k,v) \mid r\_decrypt(k,v) \mid d\_encrypt(k,v) \\
& & \mid d\_decrypt(k,v) \mid op\_encrypt(k,v) \mid op\_decrypt(k,v) \\
& & \mid a\_encrypt(k,pk,v) \mid a\_decrypt(k,pk,v) \mid [S].s
\end{array}
$$

Figure 5: Syntax of λ_{SQL}

strong encryption scheme.

The example described above also illustrates the security-performance trade-offs that arise in the trusted client model. Ensuring the absence of explicit and implicit flows often requires more columns to be encrypted, and additional round trips to the client. SecureSQL's type system identifies such flows, and gives users the choice of enforcing absence of explicit and/or implicit flows. Figure 4 shows a partitioning generated by SecureSQL for this example when implicit flows are permitted. The partitioning is efficient since all computation is offloaded to the server. The trusted client simply encrypts parameters, uses linked server at line 39 to invoke the server component, and decrypts results. Refer to [5] for the partitioning that contains no explicit or implicit flows. This partitioning requires the column `C_DATA` to be encrypted and delegates the string concatenation operation in line 18 to the client (at the cost of an additional round trip).

3. Information flow types for partially homomorphic encryptions

The type system in SecureSQL serves dual purposes. First, it automatically rewrites a database application by inserting calls to encryption/decryption routines so that its semantics are preserved in the database with encrypted columns. Furthermore, the type system ensures the absence of explicit and/or implicit flows, assuming all local variables are stored in trusted memory. The partitioning algorithm (Section 4) further partitions stored procedures, moving local variables and statements from the trusted client to the server while preserving information flow safety. While type systems for information flow have been studied extensively [26, 24], the combination of partially homomorphic encryption and information flow, and its use for automatic partitioning are unique to our system.

$$
\begin{array}{lll}
\tau & ::= & \text{CT} \mid \text{RE} \mid \text{AE} \mid \text{DE} \mid \text{OPE} \mid \text{Void} \\
& & \mid [\tau_1, ..., \tau_n] \mid [n_1 : \tau_1, ..., n_n : \tau_n] \mid \tau_1 \rightarrow \tau_2
\end{array}
$$

Figure 6: Types in the type system for partially homomorphic encryptions.

Language. Instead of presenting the type system for the entire T-SQL language, we define a simpler, core language called λ_{SQL} (Figure 5) which models key features of declarative query languages. An λ_{SQL} program p is a collection of named stored procedures. The state of a λ_{SQL} program consists of a database with a set of tables t and a set of local variables v. The body of a procedure is a statement (s). An expression is either a constant (x), a variable (v), reference to a column in a table ($t.c$) or a named record $[n_1 : e_1, ..., n_m : e_m]$. A named record is a tuple with a unique name associated with each element. Named records permit relational operations such as projection and cartesian product. λ_{SQL} supports basic arithmetic, boolean operations and function application, assignment, imperative control flow in the form of branching statement, standard relational operators for querying data including projection ($\pi_{n_1,...,n_m}(e)$), selection ($\sigma_{e_1}(e_2)$), union ($e_1 \cup e_2$), difference ($e_1 \backslash e_2$), and cartesian product ($e1 \times e2$). $\pi_{n_1,...,n_m}(e)$ selects a subset of columns from a collection of named records. $\sigma_{e_1}(e_2)$ selects all tuples from the result of the expression e_2 that satisfy the predicate e_1. Tables (t) can be modified using insert, update, and delete statements. The select statement can be used to query tables and return results.

λ_{SQL} also supports a set of expressions used in generated code but not available to the programmer (e_t). This includes the implementations of encryption and decryption algorithms. The algorithms for randomized, deterministic and order-preserving encryption use symmetric private keys, whereas additive encryption use asymmetric, public-private key pairs. We assume that the algorithm for randomized and additive encryption are secure under standard cryptographic assumptions. The language does not support explicit creation or manipulation of encryption keys - we assume that the application (i.e. the trusted client) has access to a finite number of predefined keys/key pairs. Finally, $[S].s$ offloads execution of statement s from a client database to a server S.

Types. Given an λ_{SQL} program, our goal is to assign each expression in the program a type which represents whether the value returned by the expression is encrypted or not, and the type of encryption used i.e. randomized (RE), additive (AE), deterministic (DE) or order-preserving (OPE). SecureSQL's type system (Figure 6) consists of a set of primitive types, one for each type of encryption. The type CT represents cleartext values, and the type Void represents expressions that do not return values (e.g. assignment). The type system also supports function types and named record types. A named record type is a type representing a tuple of values, where each element

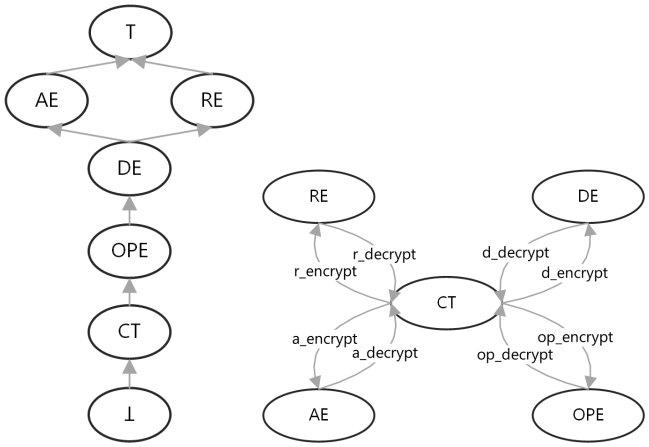

Figure 7: Types and encryption and decryption routines for coercing values between types.

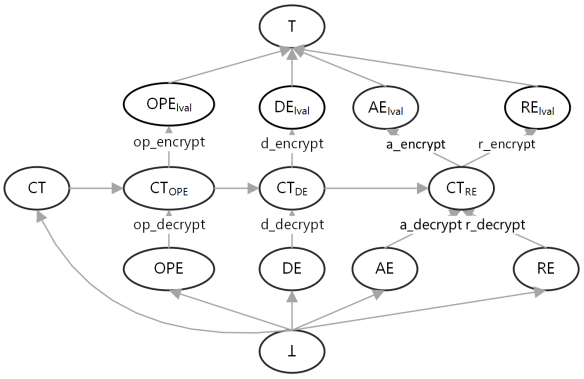

Figure 8: A coercive subtyping relation which tracks information flow from encrypted types. Unlabeled edges have identity functions as coercions.

in a record type is associated with a name. Record types are a natural way of representing types of tables and relational operators (such as projection and product).

Note that types described above do not refer to encryption keys. For the purpose of this paper, we make a simplifying assumption that all values encrypted using the same kind of encryption use the same encryption key. In our implementation, a type consists of the pair <encryption type, keyid>, where keyid is a unique identifier representing each encryption key.

Encryption/decryption as Coercions. In a conventional type system for information flows [26, 24], types correspond to secrecy levels in a lattice. For example, the types $\top$ and $\bot$ represent secret and public values respectively. In an encrypted database, secrecy levels correspond to partially homomorphic encryption schemes. The encryption schemes can be ordered by the relative strength of encryption; this ordering forms a lattice shown in Figure 7. A type system based on this lattice can be used to *check* if a partitioned application contains any explicit or implicit information flows, and if the application uses partially homomorphic encryptions correctly. However, just type checking does not suffice if our starting point is a monolithic application written for a database without encryption – we wish to instrument encryption/decryption routines to preserve semantics of the application, and partition the application while preventing information flows.

Towards this end, we define a type system where encryption/decryption routines are viewed as *coercions* between types, similar to how compilers coerce of values from one (usually less precise) type to another (more precise) type. Specifically, we propose a type system with coercive subtyping [18]. In coercive subtyping, τ_1 is said to be a subtype of τ_2 if a value of type τ_1 can be coerced into a value of type τ_2 using a coercion function $\sigma_{\tau_1 \to \tau_2}$. In such a system, the *subtyping relation* $<:$ defines the set of coercions permitted by the type system. In our type system, encryption and decryption routines are coercion functions. Figure 7 shows all coercion functions along with the source and target types. Essentially, we can coerce a value of a given encryption type to a value of any other encryption type using one or more encryption/decryption routines.

At first glance, the relation in Figure 7 appears to be an obvious choice for a subtyping relation. However, permitting coercions between all encryption types is not desirable. If we permit arbitrary coercions, values from encrypted columns can be coerced into values encrypted using weaker encryption schemes, resulting in insecure information flows. Another reason for not permitting all coercions is efficiency – type inference is tractable only when the subtyping relation is a partial order, and efficient when the partial order is a lattice [23].

Figure 8 shows the subtyping relation used in SecureSQL. This relation (which is a lattice) is derived from Figure 7 by unrolling cycles once and adding a few additional types. The types $\mathsf{CT}_{\mathsf{OPE}}$, $\mathsf{CT}_{\mathsf{DE}}$, and $\mathsf{CT}_{\mathsf{RE}}$are variants of the type CT – they distinguish cleartext values obtained via decryption from cleartext values that were never encrypted. Both randomized and additive encryption share the same clear text type because they are both equally secure (in the cryptographic sense). $\mathsf{OPE}_{\mathsf{lval}}, \mathsf{DE}_{\mathsf{lval}}, \mathsf{AE}_{\mathsf{lval}}$, and $\mathsf{RE}_{\mathsf{lval}}$ are variants of encrypted types which represent re-encrypted values. Unlabeled edges have identity functions as coercions. Observe that this subtyping relation permits encrypted values to be decrypted into cleartext values, and encrypted back using the same or stronger encryption scheme. As described in Section 4, these additional types allow the partitioning algorithm to identify computations that can be safely offloaded to the server.

Type inference. We now describe SecureSQL's algorithm (based on [21]) for automatically inferring encryption types and coercions. Informally, the algorithm work as follows. The algorithm associates two *type variables* with every expression, an *assumed type* α, which represents the type of the expression without coercions, and an *expected type* β, which represents the type after a coercion permitted by the subtyping relation $<:$ has been applied. The algorithm collects set constraints on these type variables during a traversal of the AST. A solution

to these constraints, if one exists, assigns values to assumed and expected type variables of every expression. Expressions which require requires coercions are expressions with different assumed and expected types.

For tracking implicit flows, the algorithm associates a type variable γ with every statement s. s has an encryption type γ if all columns/variables updated by s have encryption type at least as strong as γ. γ also represents the *context* in which a statement can execute without introducing implicit flows. A statement s with type γ can safely execute in a context of type γ or weaker.

The algorithm can be stated as a set of type rules. Figure 9 shows a subset of the type rules (refer to [5] for the complete set of type rules). Type judgments for expressions take the form $\Sigma, \Gamma \vdash e : \beta$. Σ is a set of type constraints of the form $\tau <: \beta$ or $\tau \in S$, where τ is a type expression, β is a type variable, and S is a set of types. Γ is a map from expressions to their assumed types. A type judgment is interpreted as follows: under constraints Σ and assumptions Γ, the expected type of e is β. Type judgments for statements take the form $\Sigma, \Gamma, \gamma \vdash s$, where γ represents the context in which the statement s can safely execute.

Consider the rule VAR for variables. The rule allocates two fresh type variables α and β, sets the assumed type of v to α, and adds a constraint that $\alpha <: \beta$, where β is the expected type of v. The rule for column references is similar, with the only difference that the assumed type of a column reference is obtained from the encryption policy E.

Now consider the rule EQUALS. Let β_1 and β_2 be the expected types of e_1 and e_2 respectively. Equality checking requires types of both sub-expressions to be the same. This is enforced by unifying both types (using the function UNIFY). Equality also requires that both types should be at least at weak as deterministic encryption. We enforce this using a subset constraint $\beta_1 \in [\mathsf{OPE}, \mathsf{OPE}_{\mathsf{lval}}, \mathsf{DE}, \mathsf{DE}_{\mathsf{lval}}, \mathsf{CT}, \mathsf{CT}_{\mathsf{OPE}}, \mathsf{CT}_{\mathsf{DE}}, \mathsf{CT}_{\mathsf{RE}}]$. The algorithm also unifies the assumed types of variables common to e_1 and e_2. The constraint on the type of equality check reflects the fact that unlike other partially homomorphic encryptions, deterministic encryption and OPE reveal the result of the check in cleartext. The rule COMP for comparison follow the same template and only differ in the set of permitted encryption types.

Inserting into a table (INSERT) requires that the type of values being inserted must be a subtype of the column being written to (as defined in the policy). Here, we use the function LVALTYPE to obtain the re-encrypted type corresponding to an encryption type. Observe that this rule allows re-encryted values to be written to a column to a column of a given encryption type. For example, a value of type $\mathsf{RE}_{\mathsf{lval}}$ obtained by decrypting a value of type $\mathsf{DE}_{\mathsf{lval}}$ and re-encrypting it using randomized encryption can be written to a column of type RE. The type γ captures the strongest encryption type that is written by the statement.

Consider the rule IF. The constraint on the contexts of

$$\text{[CONST]}\ \frac{\beta = \mathsf{FRESH}()}{\{\mathsf{CT} <: \beta\}, \{x : \mathsf{CT}\} \vdash x : \beta} \qquad \text{[VAR]}\ \frac{\alpha, \beta = \mathsf{FRESH}()}{\{\alpha <: \beta\}, \{v : \alpha\} \vdash v : \beta}$$

$$\text{[COLUMN]}\ \frac{\alpha = E(t,c) \qquad \beta = \mathsf{FRESH}()}{\{\alpha <: \beta\}, \{v : \alpha\} \vdash t.c\ : \beta}$$

$$\text{[EQUALS]}\ \frac{\begin{array}{c}\Sigma_1, \Gamma_1 \vdash e_1 : \beta_1 \qquad \Sigma_2, \Gamma_2 \vdash e_2 : \beta_2 \qquad \beta = \mathsf{FRESH}() \\ S = \mathsf{UNIFY}(\{\alpha, \beta \mid v : \alpha \in \Gamma_1 \wedge v : \beta \in \Gamma_2\} \cup \{\beta_1, \beta_2\})\end{array}}{\begin{array}{c}S(\Sigma_1) \cup S(\Sigma_2) \cup \{\beta \in [\mathsf{CT}, \mathsf{CT}_{\mathsf{OPE}}, \mathsf{CT}_{\mathsf{DE}}, \mathsf{CT}_{\mathsf{RE}}]\} \cup \\ \{\beta_1 \in [\mathsf{OPE}, \mathsf{OPE}_{\mathsf{lval}}, \mathsf{DE}, \mathsf{DE}_{\mathsf{lval}}, \mathsf{CT}, \mathsf{CT}_{\mathsf{OPE}}, \mathsf{CT}_{\mathsf{DE}}, \mathsf{CT}_{\mathsf{RE}}], \\ S(\beta_1) <: \beta\}, S(\Gamma_1) \cup S(\Gamma_2) \vdash e_1 = e_2 : \beta\end{array}}$$

$$\text{[COMP]}\ \frac{\begin{array}{c}\Sigma_1, \Gamma_1 \vdash e_1 : \beta_1 \qquad \Sigma_2, \Gamma_2 \vdash e_2 : \beta_2 \qquad \beta = \mathsf{FRESH}() \\ S = \mathsf{UNIFY}(\{\alpha, \beta \mid v : \alpha \in \Gamma_1 \wedge v : \beta \in \Gamma_2\} \cup \{\beta_1, \beta_2\})\end{array}}{\begin{array}{c}S(\Sigma_1) \cup S(\Sigma_2) \cup \{\beta \in [\mathsf{CT}, \mathsf{CT}_{\mathsf{OPE}}, \mathsf{CT}_{\mathsf{DE}}, \mathsf{CT}_{\mathsf{RE}}]\} \cup \\ \{\beta_1 \in [\mathsf{OPE}, \mathsf{OPE}_{\mathsf{lval}}, \mathsf{CT}, \mathsf{CT}_{\mathsf{OPE}}, \mathsf{CT}_{\mathsf{DE}}, \mathsf{CT}_{\mathsf{RE}}], \\ S(\beta_1) <: \beta\}, S(\Gamma_1) \cup S(\Gamma_2) \vdash e_1 < e_2 : \beta\end{array}}$$

$$\text{[ADD]}\ \frac{\begin{array}{c}\Sigma_1, \Gamma_1 \vdash e_1 : \beta_1 \qquad \Sigma_2, \Gamma_2 \vdash e_2 : \beta_2 \qquad \beta = \mathsf{FRESH}() \\ S = \mathsf{UNIFY}(\{\alpha, \beta \mid v : \alpha \in \Gamma_1 \wedge v : \beta \in \Gamma_2\} \cup \{\beta_1, \beta_2\})\end{array}}{\begin{array}{c}S(\Sigma_1) \cup S(\Sigma_2) \cup \{\beta_1 \in [\mathsf{AE}, \mathsf{AE}_{\mathsf{lval}}, \mathsf{CT}, \mathsf{CT}_{\mathsf{OPE}}, \mathsf{CT}_{\mathsf{DE}}, \mathsf{CT}_{\mathsf{RE}}], \\ S(\beta_1) <: \beta\}, S(\Gamma_1) \cup S(\Gamma_2) \vdash e_1 + e_2 : \beta\end{array}}$$

$$\text{[ASSIGN]}\ \frac{\begin{array}{c}\Sigma_1, \Gamma_1 \cup \{e_1 : \alpha_1\} \vdash e_1 : \beta_1 \qquad \Sigma_2, \Gamma_2 \vdash e_2 : \beta_2 \\ S = \mathsf{UNIFY}(\{\alpha, \beta \mid v : \alpha \in \Gamma_1 \wedge v : \beta \in \Gamma_2\} \cup \{\alpha_1, \beta_1\})\end{array}}{S(\Sigma_1) \cup S(\Sigma_2) \cup \{S(\beta_2) <: S(\beta_1)\}, S(\Gamma_1) \cup S(\Gamma_2), S(\beta_1) \vdash e_1 := e_2}$$

$$\text{[INSERT]}\ \frac{\forall i,\ \beta_i = \mathsf{LVALTYPE}(E(t,c_i)) \qquad \Sigma, \Gamma \vdash e : \beta \qquad \gamma = \mathsf{FRESH}()}{\Sigma \cup \{\beta <: [\beta_1, ..., \beta_n]\} \cup \{\forall i \in [1, ..., n], S(\beta_i) <: \gamma\}, \Gamma, \gamma \vdash : \mathsf{insert}_{n_1, ..., n_m}(t)\ e}$$

$$\text{[SELECT]}\ \frac{\Sigma, \Gamma \vdash e : \beta \qquad \forall i \in [1, ..., m],\ \beta_i = \mathsf{PROJECT}(\beta, n_i) \qquad \gamma = \mathsf{FRESH}()}{\{\Sigma \cup \{\forall i \in [1, ..., n], S(\beta_i) <: \gamma\}, \Gamma, \gamma \vdash : \mathsf{select}_{n_1, ..., n_m}(e)}$$

$$\text{[IF]}\ \frac{\begin{array}{c}\Sigma_1, \Gamma_1 \vdash e_1 : \beta_1 \qquad \Sigma_1, \Gamma_1, \gamma_1 \vdash s_1 \qquad \Sigma_2, \Gamma_2, \gamma_2 \vdash s_2 \\ S = \mathsf{UNIFY}(\{\alpha, \beta \mid v : \alpha \in \Gamma_i \wedge v : \beta \in \Gamma_j\} \cup \{\gamma_1, \gamma_2\})\end{array}}{\begin{array}{c}S(\Sigma_1) \cup S(\Sigma_2) \cup S(\Sigma_3) \cup \{S(\beta_1) <: S(\gamma_1)\}, \\ S(\Gamma_1) \cup S(\Gamma_2) \cup S(\Gamma_3), S(\gamma_1) \vdash \mathsf{if}\ e_1\ s_1\ s_2\end{array}}$$

Figure 9: A subset of rules from the type inference algorithm.

'then' and 'else' statements these statements can only update columns encrypted with a stronger scheme than the variables involved in the branching condition. For example, consider the IF statement at line 16 in Figure 3. If the column `C_CREDIT` is encrypted using deterministic encryption, the equality check can be performed on the server by encrypting the constant `'BC'`. In this case, the type of the branching condition is $\mathsf{CT}_{\mathsf{DE}}$ (rule EQUALS). The rule IF ensures that all columns updated in the 'then' branch should have a type at least as strong at $\mathsf{CT}_{\mathsf{DE}}$. If the column `C_DATA` is not encrypted, this condition cannot be satisfied due to an implicit flow, and the

algorithm reports an error.

Parameters and return values. The algorithm introduces additional constraints on parameters and return values of procedures that can be externally invoked. Specifically, we require that the type of all parameters and the return variable of such procedures must be one of the cleartext types. This constraint ensures that the interface of these stored procedures does not change from the perspective of external callers.

Procedure and function calls. The SecureSQL compiler handles procedure calls as follows. Given a set of stored procedures, the compiler constructs a call-graph, performs a bottom up traversal of the call-graph and runs the type inference algorithm described above for each procedure. After processing each node, computes a *summary* for each procedure, which consists of the encryption types of the formal parameters and return values. The summary also records whether the stored procedure requires any coercions. At a procedure call, the compiler uses the summary to introduce constraints on parameters and return values. Specifically, we require that the expected type of an actual parameter should be a subtype of the corresponding formal parameter, and the type of the return value should be a subtype of the expected type of the actual return value. In case of recursive calls, the compiler repeats the inference until a fixed point is reached.

Cost-aware constraint solving. The constraints generated by the type system are inequalities over the subtyping relation, which is a partial order. In general, the problem of poset solving is NP-complete [23]. However, our subtyping relation is a lattice with the join defined by the lowest common ancestor relation. Solving inequalities over a lattice is linear in the number of expressions and the height of the lattice [23]. The result of constraint solving is either a type assignment which assigns a type to each type variable, or a set of unsatisfiable constraints. A set of constraints may be unsatisfiable if the program contains insecure information flows. We report all unsatisfiable constraints to the user, and also recommend the changes to the encryption policy that will satisfy the constraints.

One shortcoming of the inference algorithm described above is that it unaware of the runtime cost of coercions. For example, consider the equality check at line 10 in Figure 3. If the column `CUSTOMER.C_LAST` is deterministically encrypted, there are two possible assignments of types to expressions `CUSTOMER.C_LAST` and `@c_last` that will preserve semantics of the filter, one in which both expressions are assigned the type $\mathsf{CT}_{\mathsf{DE}}$, and the other where they are assigned the type $\mathsf{DE}_{\mathsf{lval}}$. Both type assignments result in the same number of coercions (i.e. one). In the first case, the column `C_LAST` is decrypted, and in the second case, the variable `@c_last` is deterministically encrypted. The resulting queries have very different query plans. In the first case, the column `C_LAST` must transferred to the client and the check is performed on the client. The second case lends to a more efficient plan where the filter is performed on the server using an index.

In order to find efficient type assignments, we extend the inference algorithm in [23] with a simple cost model and a procedure that systematically explores the space of all valid type assignments to find type assignments with minimal cost. Our cost model computes the cost of a type assignment by assigning a cost to all expressions that must be coerced under the assignment. The cost of a coercion on a column reference is approximated by the cardinality of the column (estimated using statistics from the server). The cost of coercing as primitive value is 1. The cost of an expression is the sum of the cost of its sub-expressions.

We explore the space of all valid type assignments using the following iterative algorithm. We start with a valid type assignment obtained using the inference algorithm described above, and systematically derive other type assignments by substituting the type assigned to a type variable with its super-types. We use each derived type assignment to assign the *assumed* type of each type variable and rerun the inference algorithm. If the inference algorithm succeeds, we have found another valid but different type assignment and we can compute its cost. If the inference fails, we ignore the type assignment and continue the search. The complexity of the exploration is polynomial in the number of expressions and the height of the subtyping relation.

Inserting Coercions. Given a type assignment, the expressions that require coercions are expressions which generated constraints of the form $\alpha <: \beta$ where $\alpha \neq \beta$. Such expressions require a coercion from α to β because the assumed and expected type are different. For every such constraint, we derive the coercion function by traversing the path from α to β in the subtyping relation and composing coercions functions for each edge on the path, and instrument the expression with the coercion function. We also identify all addition expressions where both sub-expressions are assigned a type AE or $\mathsf{AE}_{\mathsf{lval}}$, and replace each such addition with a call to homomorphic addition routine.

4. Secure Partitioning

The next phase in the compilation process is to generate a secure and efficient partitioning of a database application that offloads as much computation as possible to the server without leaking sensitive information. In this section, we describe the analysis and optimizations that together generate such partitionings.

Baseline partitioning. The first step in the algorithm is to generate a *baseline partitioning* which preserves semantics. For the baseline partitioning,SecureSQL relies on named references, which is a commonly abstraction for distributed query processing in most databases. For example, SQL

Server supports a primitive called *linked server* [6] and Oracle supports a similar primitive called database link [4]. A linked server is a named reference that can be used by a client database to issue queries on remote servers. For example, a client database can issue a query `SELECT * FROM [LINK].[DB].CUSTOMER ORDER BY C_BALANCE` to select customers from a remote table[1]. The query optimizer on the client partitions queries that use linked server between remote servers and the client, utilizing any statistics that may be available on the remote server.

SecureSQL uses linked server to partition a stored procedure by rewriting all table references to remote table references, and installing the resulting stored procedure on the client. Linked server automatically promotes transactions initiated on the client to distributed transactions involving the remote server.

Unfortunately, while the baseline partitioning is simple and preserves semantics, it is neither secure not efficient. In this partitioning, the number of round-trips between the client and server is proportional to the number of queries because the client drives execution, and control returns back to the client after each query, even if the next query can execute entirely on the server. Furthermore, transactions are promoted to distributed transactions even if all queries within the transaction are safe. The partitioning is also not secure because information can leak through local variables that hold sensitive data in cleartext, and via intermediate results of query processing. For example, consider the join query `SELECT * FROM [LINK].R, [LINK].S WHERE dbo.DECRYPT(R.a) = S.a`. If $|R| \ll |S|$, a possible query plan is to decrypt the column `R.a` on the client, and transfer the semi-join to the server to compute the join. This plan leaks contents of the column `R.a` in plaintext. SecureSQL uses a series of analysis and optimizations for generating a secure and efficient partitioning from the baseline.

Safety analysis. Safety analysis is a static analysis that infers whether a type-annotated T-SQL expression can be safely offloaded to the server without introducing insecure information flows. The analysis is based on the following definition for safety. A T-SQL expression is *safe* if (a) does not invoke encryption or decryption routines, (b) it does not use any local variables that contain values obtained from prior calls to decryption routines.

Checking whether an expression invokes encryption/decryption routines is straightforward – we simply check if any sub-expressions require coercions (using the type assignment). We can also check the second condition recursively as follows. A local variable is it not safe if its expected type is one of the types $[\mathsf{CT}_{\mathsf{OPE}}, \mathsf{CT}_{\mathsf{DE}}, \mathsf{CT}_{\mathsf{RE}}]$. An expression is safe if all constants and variables in the expression are safe.

Secure partitioning. SecureSQL uses safety analysis to transform the baseline partitioning into a secure (but inefficient) partitioning as follows. The compiler identifies and instruments all unsafe scalar expressions with special "identity" routines installed on the client. The presence of these routines forces the query optimizer to consider plans where these functions are evaluated on the client. For example, consider the query `SELECT * FROM [LINK].R WHERE dbo.IDENTITY(dbo.DECRYPT(R.a) = @filter)`. Assume that the variable `@filter` is unsafe. The call to the identity routine forces the filter to be evaluated on the client.

Security-aware optimizations. We now describe three optimizations in SecureSQL, which increase the number of safe expressions, and hence offload more computation to the server. SecureSQL supports other optimizations such as splitting sub-queries and splitting materialized view definitions – we omit these due to space constraints. A key feature of these optimizations is that they can be expressed as source-to-source transformations, as opposed to transformations on a physical query plan.

Invariant code motion. An expression is *invariant* with respect to a query if its value does not change during the query's execution. Invariant code motion is an optimization that identifies invariant expressions (such as calls to encryption/decryption routines), and moves them before the query. For example, consider the `SELECT` query at line 6 in Figure 3. In this example, the type system infers that variables `@c_last` and `@h_amount` should be encrypted, and introduces encryption routines. However, the expressions `d_encrypt(@c_last,...)` and `a_encrypt(@h_amount,...)` are invariant with respect to the query. The transformation moves these expressions before the query, introduces temporary variables (`@enc_c_last` and `@enc_h_amount`) to capture the values of these expression, and replaces the expressions with temporary variables. The resulting query is free of calls to encryption and decryption routines, and therefore safe.

Offloading Safe Blocks. This optimization offloads safe statements to the server. The optimization consists of two phases. The first phase identifies *maximal safe blocks*. A *block* is a set of statements in the same static scope. A *maximal safe* block is the largest block such that all statements within the block are safe but at least one of the siblings is not safe.

The second phase of the transformation partitions the stored procedure further by *extracting* maximal safe blocks into separate stored procedures. This is akin to creating closures [3]. Each maximal safe block is transformed into a stored procedure whose input parameters are variables used within the block, and output parameters are variables defined within the block but used outside. This transformation then deploys closures on the server, and replaces the maximal safe block with a remote call to the closure. As a special case, if

[1] Here `LINK` is the name of the linked server.

all statements in a stored procedure are safe, the optimization deploys the entire stored procedure on the server. Figure 4 shows an instance of this optimization, where the entire procedure is replaced by a call to a closure.

Distributed transaction elimination. Recall that the client database automatically promotes local transactions to distributed transactions while processing queries with remote references. Distributed transactions are often implemented using expensive, blocking protocols such as 2PC. This optimization eliminates distributed transactions wherever possible. In particular, the optimization checks if all remote accesses within the scope of a transaction occur in one maximally safe block. If this condition is satisfied, this optimization eliminate the distribution transaction by reducing the scope of the transaction to the maximal safe block. This effectively pushes the transaction into the closure, which executes locally on the server. Figure 4 shows an example of this transformation.

5. Evaluation

The goal of our evaluation to is determine the cost of supporting information flow security in encrypted databases.

Implementation. SecureSQL is implemented as a C# library (˜ 11000 LOC) based on a publicly available T-SQL parser and code generator [8]. The type system, rewriting for adding coercions, query partitioning and optimizations are implemented as transformations over the AST generated by the parser. We use 256-bit AES block cipher in CBC mode for randomized encryption, AES in GCM mode with an initialization vector derived from the hash of the cleartext for deterministic encryption, 1024 bit Paillier encryption and an order-preserving encryption scheme based on [22].

Methodology and Benchmarks. The experiments were conducted on an in-house cluster connected via a 1 Gbps network. Each machine has a 2 Intel Xeon x64 processors with 8 cores each, 16GB RAM, a 1TB SATA drive and runs Windows Server 2008 Enterprise. Our evaluation is based on two applications TPC-C, and Perf, an in-house employee performance evaluation application. The TPC-C database maintains customer, order and inventory information. We instantiated this database with 100 warehouses (approximately 10 GB). The workload consists of five transactions implemented as stored procedures. The workload generator simulates a specified number of concurrent users issuing transactions with a recommended mix (45% new order transactions, 43% order status, and 4% each of other transactions). Each run lasts 5 minutes preceded by a warm-up period of 2 minutes. We use the number of transactions per minute (TPM), and the average latency of transactions (in milliseconds) as measures for throughput and latency.

The Perf database maintains employee performance data for one year (approximately 50 GB). The database application consists of 104 stored procedures of varying complexity (from 100 - 2500 LOC), and 69 views. The workload consists of a mix of 7 complex workflows, where each workflow invokes several stored procedures. We choose this application because it has strong security requirements but much lower throughput and latency requirements than TPC-C, and therefore serves as an interesting design point.

Policy	# Coercions					# DT
	DEL	*NEWORD*	*OSTAT*	*PAY*	*SLEV*	
NoFlows	3	3	8	44	0	4
NoExp	3	3	4	24	0	2
NoExpAdd	0	0	2	18	0	0

Table 1: Number of coercions and the number of procedures requiring distributed transactions in the TPC-C benchmark for different encryption policies.

5.1. TPC-C

For evaluating the security performance trade-off in TPC-C, we defined three progressively weaker encryption policies. The policy *NoFlows* uses randomized encryption for all columns in the customer table containing personally identifying information. SecureSQL's type system certifies that this policy does not contain any implicit or explicit insecure information flows. However, the policy does not utilize indexes or permit server side computation on these columns. The policy *NoExp* weakens *NoFlows* by using deterministic encryption for customer last name and credit status, and leaving the column `C_DATA` in cleartext. This policy has an implicit flow from the column `C_CREDIT` to `C_DATA`, but is able to use indexes on these columns. The third policy *NoExpAdd* is similar to *NoExp* except it uses Paillier encryption for customer's account balance. Our baseline is a configuration (*ClearText*) where all columns are cleartext, and the application runs entirely on the server. The compiled stored procedures for all policies are available at [5].

Static properties. We measured two properties of a partitioned application that serve as good indicators of performance i.e. the number of coercions, and the number of distributed transactions (Table 1). With policy *NoFlows* 4 out of 5 stored procedures require coercions, with the procedure *Payment* requiring as many as 44 coercions, illustrating the complexity of rewriting. Both the number of coercions and the number of procedures that require distributed transactions reduces as the policy is weakened. With policy *NoExpAdd*, all computation (with the exception of encryption & decryption routines, and simple scalar operations) is offloaded to the server, and distributed transactions are eliminated altogether. We also observe that encryption increases the space overheads of the database by roughly 45% (with negligible variance across policies).

Performance Overheads. Figure 10 shows the average

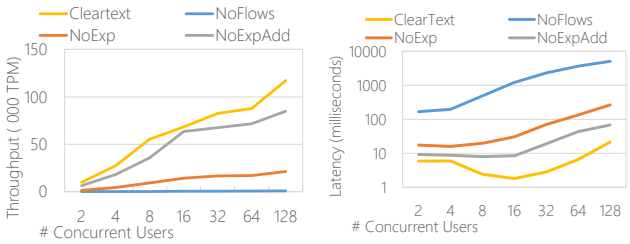

Figure 10: Throughput and latency of TPCC with varying load and encryption policies.

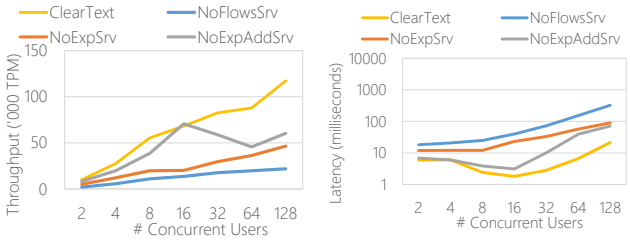

Figure 11: Throughput and latency of TPCC for configurations where the server has access to keys.

throughput and latency of the partitioned TPC-C with varying number of users. As one might expect, throughput scales almost linearly with load in the baseline configuration with all columns in cleartext. The throughput with the policy *NoExpAdd* closely matches the baseline, except at high load where the throughout is lower by 28%. The discontinuity at 16 users is explained by the number of cores available on the server. Further analysis shows that the lower throughput is due to encryption and decryption operations on the trusted client. Partitioning has a larger effect on latency for policy *NoExpAdd* – average latency increases from 1.8ms to 8.4ms with 16 concurrent users. This is due to at least one additional hop between application and the database server, and the latency of encryption and decryption operations.

Both throughput and latency drop significantly for policies *NoFlows* and *NoExp* (with a maximum of 1000 and 19800 TPM respectively). The drop is performance with *NoFlows* is not be entirely unexpected. In this configuration, most of the query processing and control flow remains on the client. An analysis of the most time consuming queries using SQL profiler reveals that cursors are the worst affected, followed by queries that filter on encrypted columns. Since these queries now execute on the client, performance suffers due to the inability to use indexes, and the additional data transfers to the client. However, the drop in performance with *NoExp* is more surprising.

We performed a set of experiments to isolate the cause of loss in performance. We considered three additional configurations *NoFlowsSrv*, *NoExpSrv* and *NoExpAddSrv* where the database is encrypted with policies *NoFlows* *NoExp* and *NoExpAdd* respectively, and the database server has access to encryption keys. Therefore, both the server and trusted client

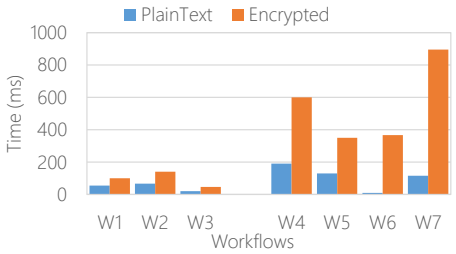

Figure 12: Average latency of workflows with and without encryption.

components of the application can be deployed on the server. These configurations are not secure – they simply eliminate overheads of the network.

Figure 11 shows the throughput and latency for these configurations. The throughout and latency of *NoExpSrv* are significantly better than *NoExp*. Both through and latency improve by a factor of 5, with latency within 10% of *NoExpAdd* on average. Throughput is however lower than *NoExpAdd* by a factor of 3. This suggests that most of the overheads (if not all) can be attributed to the cost of distributed query processing. Recall that in *NoExp*, the column `C_BALANCE` in the customer table is encrypted using randomized encryption. Therefore, operations on this column are executed in the trusted client component. This requires additional round trips and use of distributed transactions in two stored procedures (`DEL` and `PAY`), which hurts throughput.

Also observe the performance of *NoExpAddSrv* reduces significantly at high loads. Further analysis of CPU usage reveals that the loss in performance is due to encryption and decryption routines, which contend for CPU cycles with the rest of the application. Contrast this with *NoExpAdd* where the load is shared between the client and the server.

Optimizations. We also measured the performance of TPC-C for each of the encryption policies with the optimizations described in Section 4 disabled. In this mode, the performance of configurations *NoExp* and *NoExpAdd* drops significantly, with throughput even lower than *NoFlows*. We also observed that no single optimization when enabled in isolation improved performance significantly. Performance improved only when these optimizations are enabled together. Therefore, optimizations play a big role in realizing the full benefits of a weaker encryption policy.

In summary, the experiments suggest that SecureSQL can ensure the absence of explicit flows even in a performance sensitive application like TPC-C with reasonably low overheads. However, overheads can he high if the encryption policy is not carefully chosen to maximize the use of existing PHE schemes, or protection from implicit flows is desired.

5.2. Employee performance evaluation

For this application, we use a strong encryption policy where all personally identifiable data and data related to an em-

ployee's performance is encrypted using randomized encryption. SecureSQL's type system is able to verify that with this policy, the partitioned application has no implicit or explicit flows. Even with this policy, we find that over 80% of the stored procedures and views do not require any coercions. However, there are procedures where the compiler introduces up to 46 coercions. Figure 12 shows the average latency of 7 main workflows. We compare the latency with a configuration where the database is in cleartext. The three most critical workflows in this application, W1, W2 and W3 are invoked when employees lookup their appraisals. These workflows are are dominated by lookups and complex joins on cleartext columns and moderately impacted by partitioning. The latency of workflows W4-W7 increase significantly after encryption. This is due to the presence of calls to string manipulation routines on encrypted data within the scope of distributed transactions, and joins over strongly encrypted columns which cannot be offloaded to the server. We also tested this application through its web based client – on the whole usability of the application is not affected by partitioning.

6. Conclusions and Future work

Encrypted databases are a first step towards the goal of applications which can "compute" on encrypted data and offer strong end-to-end confidentiality guarantee. This paper takes a step towards defining strong security properties for encrypted databases. We show that encrypted databases can support real-world workloads with reasonable security and performance. However, ensuring strong information flow security with software-only trusted clients in performance sensitive applications is challenging. Encrypted databases with trusted hardware [10, 19] may be able to offer both information flow security and robust performance. From a security perspective, formally stating the threat model and proving that SecureSQL's type system ensures confidentiality is an open problem.

7. Related Work

Trusted clients. The trusted client model was first proposed in [17]. Their system does not exploit partially homomorphic encryption. Instead, the system maintains additional indexes on encrypted columns, which permits some filters to be pushed to the server. They present an algebraic framework for partitioning individual queries along with a set of heuristics for maximizing the amount of computation pushed to the server - which requires changes to the underlying query optimizer. In contrast, we handle a richer set of abstractions (views/stored procedures) and require no changes to the underlying database infrastructure.

CryptDB [22], Monomi [25] continue the line of work in [17] by using partial homomorphic encryption schemes. They differ in the amount of computation that is allowed in the trusted client - CryptDB uses a web proxy and is not a general purpose system. Monomi permits arbitrary residual computation on the client, but is geared towards analytical workloads. The main focus in both these systems is individual queries - in this paper we study abstractions such as stored procedures, views, and transactions in addition to ad-hoc queries. Interestingly, these abstractions fundamentally change the security model - in particular, stored procedures introduce information flow between columns. Any system that ignores these information flows can potentially leak the contents of encrypted columns to a passive adversary. Our compiler uses a type system that statically checks for insecure information flows and rewrites stored procedures. Finally, we evaluate our system using complete benchmarks (as opposed to traces used in [22]). Our evaluation offers a more realistic picture of the performance of a fully general trusted client based database architecture for transactional workloads.

Untrusted clients. Chong et al [14] propose a system for automatically partitioning web applications. In their security model, the client is considered untrusted. The objective of partitioning the web application is to ensure that sensitive data does not flow to the client. Our security property is a variant of the information flow security tailored for partially homomorphic encryption schemes, which their system does not use.

Hardware based security. Another approach for securely processing query on untrusted platform using dedicated trusted hardware such as FPGAs and Intel SGX [19]. TrustedDB [11] combines a secure co-processor and a commodity server. It runs a lightweight on the secure co-processor and a full-fledged database on the server. Query processing is distributed between two databases - encrypted data is processed on the co-processor and cleartext data is processed using a commodity database. Cipherbase [10] also relies on secure hardware (in the form of an FPGA device) to process encrypted data. However, unlike TrustedDB, the database is more tightly coupled with the hardware. The database performs as much computation as possible while delegating computation that requires sensitive data in cleartext to secure hardware. Haven [12] is a system for running unmodified applications on an untrusted platform using SGX. While Haven can isolate the entire database from the platform, security is predicated on a large trusted computing base, which includes the database server and the guest operating system. Haven also does not offer strong information flow properties. Compared to an on-premises trusted client, a hardware based approach can potentially yield better performance due to lower round trip latencies. A key deficiency of hardware based approaches is that they are intrusive and cannot be implemented on top of existing infrastructure. Another shortcoming is that these systems focus on data confidentiality but do not consider leaks due to information flow. It should be possible to extend our type system and partitioning algorithm to target trusted hardware instead of a trusted client and provide stronger security

guarantees.

References

[1] Advanced Encryption Standard. http://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

[2] Authenticated encryption. http://en.wikipedia.org/wiki/Authenticated_encryption.

[3] Closure (computer programming). https://en.wikipedia.org/wiki/Closure_(computer_programming).

[4] Database link. https://docs.oracle.com/cd/B28359_01/server.111/b28310/ds_concepts002.htm#ADMIN12083.

[5] Information flows in encrypted databases. http://github.com/securesql/securesql.

[6] Linked servers. http://msdn.microsoft.com/en-us/library/ms188279(v=sql.110).aspx.

[7] TPCC Benchmark. http://tpcc.org.

[8] Tsql scriptdom. http://msdn.microsoft.com/en-us/library/microsoft.data.schema.scriptdom.sql.tsqlparser(v=VS.100).aspx.

[9] Georgios Amanatidis, Alexandra Boldyreva, and Adam O'Neil. New security models and provably-secure schemes for basic query support in outsourced databases. In *Proceedings of the International Symposium on Cryptography*, 2007.

[10] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. Orthogonal security with cipherbase. In *CIDR*, 2013.

[11] Sumeet Bajaj and Radu Sion. Trusteddb: a trusted hardware based database with privacy and data confidentiality. In *Proceedings of the 2011 SIGMOD International Conference on Management of data*, pages 205–216, 2011.

[12] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[13] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neil. Deterministic encryption and efficiently searchable encryption. In *Proceedings of the International Symposium on Cryptography*, 2007.

[14] Stephen Chong, Jed Liu, Andrew C Myers, Xin Qi, Krishnaprasad Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *ACM SIGOPS Operating Systems Review*, pages 31–44, 2007.

[15] Cédric Fournet, Jérémy Planul, and Tamara Rezk. Information-flow types for homomorphic encryptions. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 351–360, 2011.

[16] Shafi Goldwasser and Silvio Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Proceedings of the Symposium on Theory of Computing (STOC)*, 1982.

[17] Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 216–227, 2002.

[18] Zhaohui Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, 1999.

[19] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. Innovative instructions and software model for isolated execution. In *Workshop on Hardware and Architectural Support for Security and Privacy*, 2011.

[20] Daniele Micciancio. A first glimpse of cryptography's Holy Grail. *Communications of ACM*, 53(3):96, 2010.

[21] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, 1991.

[22] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *SOSP*, pages 85–100, 2011.

[23] Vaughan Pratt and Jerzy Tiuryn. Satisfiability of inequalities in a poset. *Fundamenta Informaticae*, 28:1–2, 1982.

[24] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003.

[25] Stephen Tu, M Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. In *PVLDB*, pages 289–300, 2013.

[26] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. pages 607–621. Springer-Verlag, 1997.