ENKI: Access Control for Encrypted Query Processing

Isabelle Hang
University of Bremen
Isabelle.Hang@gmx.de

Florian Kerschbaum
SAP AG
Florian.Kerschbaum@sap.com

Ernesto Damiani
University of Milan
Ernesto.Damiani@unimi.it

ABSTRACT

A data owner outsourcing the database of a multi user application wants to prevent information leaks caused by outside attackers exploiting software vulnerabilities or by curious personnel. Query processing over encrypted data solves this problem for a single user, but provides only limited functionality in the face of access restrictions for multiple users and keys. ENKI is a system for securely executing queries over sensitive, access restricted data on an outsourced database. It introduces an encryption based access control model and techniques for query execution over encrypted, access restricted data on the database with only a few cases requiring computations on the client. A prototype of ENKI supports all queries seen in three real world use cases and executes queries from TPC-C benchmark with a modest overhead compared to the single user mode.

Categories and Subject Descriptors

H.2.7 [**Database Management**]: Database Administration—*Security, Integrity, and Protection*

General Terms

Security

Keywords

Encryption-based Access Control, Encrypted Query Processing, Multi User

1. INTRODUCTION

Outsourcing an application's database backend offers efficient resource management and low maintenance costs, but exposes outsourced data to a service provider. To ensure data confidentiality, data owners have to prevent unauthorized access while the data is stored or processed. Storing data on an untrusted database requires protection measures against curious personnel working for the service provider or outside attackers exploiting software vulnerabilities on the database server. In addition, data owners also have to control data access for their own personnel.

An emerging solution to the problem of untrusted databases is encrypted query processing [28, 16, 8, 1, 2, 6, 30, 5, 25, 24] where queries are executed on encrypted data.

To grant or restrict shared data access to personnel processing unencrypted query results, data owners have to implement additional fine-grained access control mechanisms. Implementing such a multi user mode using encrypted query processing for a single user operating with one key [28, 16, 8, 1, 2, 6, 5, 25] combined with an additional authorization step at the application server like [26] can be compromised: Assume that a user working for the data owner and a service provider's employee collude. If the user knows the decryption key of the data and the employee provides the encrypted data stored in the database, they are able to decrypt all data bypassing the access control mechanisms.

This paper presents the design, implementation, and evaluation of ENKI, a system that securely processes relational operations over encrypted, access restricted relations. Its approach is to encrypt data with different access rights using different encryption keys. Further, it introduces techniques to handle query processing over data encrypted with multiple encryption keys. ENKI builds on previous work in encrypted query processing for a single user as described in [16, 8, 1, 6, 25], but ENKI is the first system that efficiently supports queries over data encrypted with different keys. Existing approaches only support query processing with multiple keys for searchable encryption which allows to check if an encrypted value matches a token [30, 24, 3] or if there is no shared data [25].

The support of query processing over access controlled encrypted data presents two major challenges: The first challenge is the mapping of any complex access control structure required in a multi user scenario to an encryption enforced access control model which still allows query execution. Previous works only focus on the access control mechanism [9] or the key management [4, 7]. The second challenge is to efficiently execute a range of queries while minimizing the revealed information on the server and the amount of computations on the client. Current approaches for multiple users offer either limited functionality [30, 25, 24, 3] or expose confidential information to the database [21].

We tackle these challenges using two ideas: First, we introduce a new model for encryption based access control in Section 3 which defines access control restrictions on the level of attribute values and applies encryption as a relational operation to enforce the access restrictions on a relational algebra.

Second, we present three different techniques to support the execution of relational operations in multi user mode. The first technique is query rewriting to adapt relational operations over data encrypted with different keys described in Subsection 4.1. The next technique

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2749439.

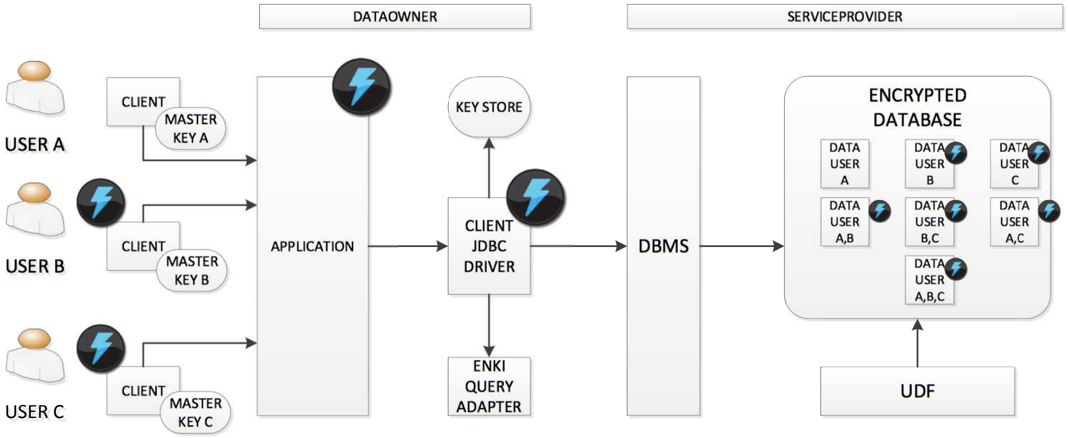

Figure 1: ENKI's architecture and threat model

is a new privacy-preserving method for join, set difference, and count distinct in multi user mode presented in Subsection 4.2. The last technique is a post-processing step on the client saving computational effort on the client while preserving the confidentiality of the data explained in Subsection 4.3. Our results are subsumed in a multi user algorithm introduced in Subsection 4.4.

We have implemented ENKI for a SAP HANA database extending HANA's JDBC driver and a client application. Our solution supports most relational operations and aggregation functions. The evaluation of different query types seen in three use cases and in the TPC-C benchmark shows that this range is suitable for real world applications. Our performance evaluation shows that ENKI consumes an average overhead of 36.98% (which is a time penalty of 0.6181 ms on average) for the query execution of queries seen in the TPC-C benchmark in a two user scenario compared to the single user mode and that the overhead increases modestly in a more complex scenario.

2. OVERVIEW

Problem Statement. Consider two users, Alice and Bob, who share access to a database with two tables R and S. Assume that Alice has private access to one tuple and shares access to other tuples with Bob in both tables R and S respectively.

We encrypt tuples of table R that are only accessible for Alice with key r_a and tuples of table S that are only accessible for Alice with key s_a. Tuples of table R that are accessible for both Alice and Bob are encrypted with key r_ab and of table S with key s_ab. Alice knows the keys r_a, r_ab, s_a, and s_ab and Bob knows the keys r_ab and s_ab.

Assume Alice issues an equal join on tables R and S. Therefore, the database executes a cartesian product on all tuples of R and S that Alice is allowed to access and proxy re-encrypts these tuples to check the equal condition. Current proxy re-encryption protocols for deterministic encryption schemes [25],[22] cannot adhere the access restrictions while applied to the tuples: they reveal private information. To illustrate the problem, consider the proxy re-encryption of keys r_a and r_ab to a new key r_c denoted as $r\_a \sim r\_c$ and $r\_ab \sim r\_c$. Existing protocols are symmetric and transitive such that a proxy re-encryption $r\_a \sim r\_c \sim r\_ab$ exists. Therefore, Bob can proxy re-encrypt all data encrypted with Alice's key r_a to their shared key r_ab. This circumvents the defined access restrictions as the proxy re-encryption reveals information exclusively accessible by Alice.

Architecture. Figure 1 shows ENKI's overall architecture and the involved entities. These involved entities are the data owner who also maintains the application and the service provider who operates the database. There are also different users (denoted in Figure 1 as User A, User B, User C) which are personnel working for the data owner.

A user accesses the database with a client which issues queries via JDBC driver to the database backend. We extended the JDBC driver with the ENKI Query Adapter to rewrite an incoming query with minimal effort to be processable in the multi user mode. We also modified the clients to post-process the returned query results. The execution of a rewritten query on a database containing encrypted tuples requires that the predicates of this query must be encrypted, too. Based on access policies, users are acquainted with the necessary encryption keys. These keys are stored encrypted in a key store. If a user logs in, she hands over her masterkey to decrypt her encryption keys stored in the key store. Using the stored and decrypted encryption keys, the ENKI Query Adapter encrypts the rewritten query. The database management system (DBMS) receives the rewritten, encrypted query and executes it on the encrypted database. The encrypted query result is returned to the JDBC driver where it is decrypted by the ENKI Query Adapter before it is post-processed on the client. Note that keys stored in the key store cannot be decrypted if their respective users are logged out.

DBMS and database stay unmodified. User-defined functions (UDFs) perform cryptographic operations like our new privacy-preserving proxy re-encryption introduced in Subsection 4.2.

Threat. Our threat model assumes that an attacker has compromised application and database server. The attacks are depicted with flashes in Figure 1. We assume that the attacker is passive:

she can read all information stored on the database, but does not manipulate the stored data or issued queries. The attacker learns the encryption keys of all users logged in at the time of the attack. Acquainted with their masterkeys and their encryption keys, the attacker is able to read all data of these compromised users stored on the database. In particular, the attacker can access data shared with other uncompromised users. ENKI offers confidentiality guarantees for non-compromised users during such an attack: the attacker cannot learn their private data i.e. data which are not shared with compromised users.

ENKI provides this security guarantees in the face of a passive attacker. An active attacker which alters or deletes information stored on the database is out of scope for ENKI. We argue that such manipulations might be easier to detect than a passive attack. ENKI does also not prevent attacks on client machines that lead to the compromise of keys.

3. ENCRYPTION-BASED ACCESS CONTROL ON A RELATIONAL ALGEBRA

This section presents a new model how to specify access rights on attribute values of relations and how to enforce them cryptographically.

3.1 Access Restrictions on Relations

We define access restrictions on attribute values of a relation using an access control matrix. Note that an access control matrix may serve as a base for more enhanced access models exploiting role-based access control [27].

Let $\mathcal{A}$ be an access control matrix where the rows correspond to subjects s and the columns correspond to objects o. Figure 2 illustrates an access control matrix with two users, Alice and Bob, as subjects and a relation R containing five tuples, $t_1, \ldots, t_5$, as objects. We denote S as the set of all subjects with $|S| = n$ and O as the set of all objects.

A data owner grants access for an object o to a subject s by setting the entry in the access matrix $\mathcal{A}[s, o]$ to 1. This enables the user to read, update, or delete the object. If no access is granted, the entry is set to 0. Our approach does not support the implementation of different types of access rights e.g. read only or read-write.

A column of an access control matrix is a representation of the set of subjects which has access to an object o. We denote this as the *qualified set* QS_o of object o. We assume that each object can be accessed by at least one subject such that there are no zero columns and no empty qualified sets. Consider $QS_{t_4} = \{1, 1\}$ in Figure 2. This is the qualified set of object t_4 denoting that user Alice and user Bob have access to tuple t_4.

We further name $\mathcal{P}^*(S)$ the power set of all subjects S without the empty set. We denote each of these subsets as $p_i \in \mathcal{P}^*(S)$ for all $i = 1, \ldots, 2^n - 1$ and call it a *user group*. From the access control matrix depicted in Figure 2, we derive three user groups:

$$\begin{aligned} p_1 =& \{Alice\} := A \\ p_2 =& \{Bob\} := B \\ p_3 =& \{Alice, Bob\} := AB. \end{aligned}$$

We define a mapping which assigns each user to the user groups she participates in. For each user s, there is a set of p_j with $j = \{1, \ldots, 2^n - 1\}$ of all user groups the user participates in. This mapping is called *user group mapping* and can be stored as a relation with the attributes user and user group. Figure 3 shows the user group mapping for our example. It has two attributes: User and User Group. It shows that user Alice is member of user groups A and AB and user Bob is member of user groups B and AB.

Access Control Matrix of Relation R

User	t_1	t_2	t_3	t_4	t_5
Alice	0	1	1	1	1
Bob	1	1	0	1	0

Figure 2: Access Control Matrix for relation R. The relation contains five tuples $t_1, \ldots, t_5$. Two users, Alice and Bob, access the the relation. The users' access is binary on tuple level such that a user is either granted the right to read and alter a tuple (denoted as 1) or not (denoted as 0).

User Group Mapping

User	User Group
Alice	A
Alice	AB
Bob	B
Bob	AB

Virtual Relation Mapping

User Group	Relation	Virtual Relation
A	R	R_A
B	R	R_B
AB	R	R_AB

Figure 3: Depicted on the left is the User Group Mapping which relates a user to the usergroups the user participates in. Depicted on the right is the Virtual Relation Mapping which relates a pair of relation and user group to a virtual relation.

A qualified set of an object maps to one and only one user group which contains the same set of subjects. We group all objects accessible by the same user group and call this an *object set*. It is defined as

$$O(p_i) = \{o | o \in O \wedge QS_o = p_i\} \quad (1)$$

for a user group $p_i \in \mathcal{P}^*(S)$. This is the set of all objects assigned to the same user group. In Table 2, it is $O(p_1) = \{t_3, t_5\}$, $O(p_2) = \{t_1\}$, and $O(p_3) = \{t_2, t_4\}$.

Note that all $O(p_i)$ form a partition over O as each two object sets are pairwise disjoint and the union of all object sets (which are non-empty by definition) is equal to the set of all objects O.

We use this resulting partition to divide the underlying relation. This is to store each object set in a separate relation which we call *virtual relation*. A virtual relation indicates that one user group can access all of its tuples. This saves the annotation of a tuple with access information as its insertion in a virtual relation implies that this tuple can be accessed by a certain user group.

For n users, each relation is partitioned in a maximum of $2^n - 1$ virtual relations. The total number of tuples does not change as each tuple of a relation is stored in one and only one virtual relation. We define a mapping which assigns each user group and relation to the virtual relation containing the tuples this user group is granted access to. This mapping is called *virtual relation mapping* and can be stored as a relation with the attributes User Group, Relation, and Virtual Relation.

Figure 3 shows the virtual relation mapping for the user groups A, B, and AB and the relation R. The pair user group A and relation R is mapped to the virtual relation R_A, the pair user group B and relation R is mapped to the virtual relation R_B, and the pair of user group AB and relation R is mapped to the virtual relation R_{AB}.

The data owner specifies and maintains the user group mapping and the virtual relation mapping.

3.2 Encryption as Relational Operation

We now define encryption as a relational operation and show how to enforce the previously defined access restrictions.

Definition 1. Consider a relation $R = R(A_1, \ldots, A_n)$ with $A_1, \ldots, A_n$ attributes. It contains tuples $t_k = (t_{1_k}, \ldots, t_{n_k})$ with $1, \ldots, n$ the number of attributes and $k = 1, \ldots, j$ the cardinality.

An *encryption* $\kappa_z(A_i)$ of attribute A_i with key z is defined as

$$\kappa_z(A_i) := \{\kappa_z(t_{i_k}) | t_{i_k} \in A_i \text{ for all } k = 1, \ldots, j\} \tag{2}$$

with t_{i_k} the attribute values of A_i for all $k = 1, \ldots, j$.
The encryption of relation $R = R(A_1, \ldots, A_n)$ is the encryption of its attributes $A_1, \ldots, A_n$ and their attribute values $t_{1_k}, \ldots, t_{n_k}$ for all $k = 1, \ldots, j$ as

$$\begin{aligned}\kappa_z(R) :=& R(\kappa_z(A_1), \ldots, \kappa_z(A_n)) \\ =& \{\kappa_z(t_{1_k}), \ldots, \kappa_z(t_{n_k}) | t_{i_k} \in A_i \\ & \text{for all } i = 1, \ldots, n \text{ and for all } k = 1, \ldots, j\}.\end{aligned} \tag{3}$$

We apply adjustable query-based encryption introduced in [25] to efficiently support the execution of relational operations over encrypted data. In the following sections, we only refer to an encryption scheme as relational operation κ_z with key z, but omit the details of the adjustable encryption and the different encryption keys of each attribute value.
We now use encryption to enforce access restrictions on the attribute values of a relation. Consider a relation $R = R(A_1, \ldots, A_n)$ and three user groups A, B, and AB. The data owner splits R into virtual relations R_A, R_B, and R_{AB}. These virtual relations adopt the same schema as relation R and contain the tuples accessible for the respective user group. The data owner generates encryption keys for each user group and encrypts the respective virtual relation with its key. She generates key *r_a* for user group A and encrypts R_A as

$$\kappa_{r\_a}(R_A) = \{\kappa_{r\_a}(t) | t \in R_A\}. \tag{4}$$

She also generates keys *r_b* and *r_ab* and encrypts R_B and R_{AB} for user groups B and AB accordingly. The data owner issues the respective keys to the member of each user group.
The number of keys distributed to a user depends on the number of user groups a user participates in. The maximum number of user groups is $2^n - 1$ which is the superset of the number of users n without the empty set. In real world applications, the total number of user groups is even smaller: its order is bound by the number of users n [19].

4. QUERY PROCESSING OVER AN ENCRYPTED RELATIONAL ALGEBRA

Encrypted query processing over a relational operation can be efficiently supported for single user mode [25]. In particular, this holds for the following primitive and derived relational operations: *selection*, *projection*, *rename*, *cartesian product*, *set union*, *set difference*, *equi join*, and the aggregate functions *group by*, *count (distinct)*, *sum*, *average*, *maximum*, *minimum*, and *sort*.
The introduction of access restrictions on relations interferes with encrypted query processing in three ways:
First, a relational operation is now executed on (potentially) multiple virtual relations depending on the access rights of the user rather than on one relation. To tackle this problem, we introduce query rewriting strategies in Subsection 4.1. Applying these rewriting strategies does not change the application logic: we point out that the user only has to submit the original, unchanged query and its user id. The ENKI Query Adapter rewrites the query and adapts it for encrypted query processing.
Second, proxy re-encryption of virtual relations is necessary to support *count distinct*, *equi-join*, or *set difference* operations on the server. We present a new privacy-preserving encryption scheme to support proxy re-encryption in a multi user setting in Subsection 4.2. It offers proxy re-encryption of attributes or relations encrypted with different keys while preserving the access rights.
Third, some relational operations can only be executed on server-side with significant computational effort, huge storage capacities, or diminishing security. For such cases, we present a client-server split requiring only small data traffic and minimal computational effort on the client while preserving confidentiality in Subsection 4.3.
All introduced techniques are combined in a multi user algorithm to handle the presence of multiple users described in Subsection 4.4. The multi user algorithm takes as input a user id and a query consisting of a combination of relational operations, processes it over virtual relations, and returns the result. It can handle an arbitrary set of users.
To explain the three techniques, we use the small example introduced in Section 3: we part table R in virtual relations R_A, R_B, and R_{AB} and encrypt them as $\kappa_{r\_a}(R_A)$, $\kappa_{r\_b}(R_B)$, and $\kappa_{r\_ab}(R_{AB})$ with keys *r_a*, *r_b*, *r_ab*. Table S is treated accordingly.

4.1 Rewriting Strategies

We introduce rewriting strategies for the relational operations *selection*, *projection*, *rename*, aggregate function *count*, *set union*, and *cartesian product* over encrypted virtual relations. Applying the rewriting strategies allows the straightforward execution of these relational operations over encrypted data. The rest of this subsection presents the rewriting of these relational operations in detail.
Selection. Consider a predicate θ (e.g. $=, <, \leq, >, \geq$) and α, β attributes, constants, or terms of attributes, constants, and data operations. A *selection* $\sigma_{\alpha\theta\beta}(R)$ on relation R issued by user Alice is executed on the encrypted virtual relations $\kappa_{r\_a}(R_A)$ and $\kappa_{r\_ab}(R_{AB})$. The condition $\alpha\theta\beta$ has to be applied on both virtual relations. Therefore, $\alpha\theta\beta$ is encrypted with key *r_a* as $\kappa_{r\_a}(\alpha)\theta\kappa_{r\_a}(\beta)$ and with key *r_ab* as $\kappa_{r\_ab}(\alpha)\theta\kappa_{r\_ab}(\beta)$. It is

$$\begin{aligned}&(\sigma_{\alpha\theta\beta}(R), Alice) \\ =&\sigma_{\kappa_{r\_a}(\alpha)\theta\kappa_{r\_a}(\beta)}(\kappa_{r\_a}(R_A)) \wedge \sigma_{\kappa_{r\_ab}(\alpha)\theta\kappa_{r\_ab}(\beta)}(\kappa_{r\_ab}(R_{AB})) \\ =&\{\kappa_{r\_a}(t) | t \in R_A \wedge \kappa_{r\_a}(\alpha)\theta\kappa_{r\_a}(\beta)\} \\ & \cup \ \{\kappa_{r\_ab}(t) | t \in R_{AB} \wedge \kappa_{r\_ab}(\alpha)\theta\kappa_{r\_ab}(\beta)\}.\end{aligned} \tag{5}$$

Projection. Let R' be a relation with

$$R'(A_{i(1)}, \ldots, A_{i(k)}) \subseteq R(A_1, \ldots, A_n) \tag{6}$$

and R'_A and R'_{AB} the respective virtual relations. A *projection* $\pi_\beta(R)$ with attribute list

$$\beta = (A_{i(1)}, \ldots, A_{i(k)}) \subseteq (A_1, \ldots, A_n) \tag{7}$$

on relation R issued by user Alice is executed on the encrypted virtual relations $\kappa_{r\_a}(R_A)$ and $\kappa_{r\_ab}(R_{AB})$. Therefore, the attribute list β is encrypted with key *r_a* as

$$\kappa_{r\_a}(\beta) = \kappa_{r\_a}(A_{i(1)}), \ldots, \kappa_{r\_a}(A_{i(k)}) \tag{8}$$

and also encrypted with key *r_ab* as

$$\kappa_{r\_ab}(\beta) = \kappa_{r\_ab}(A_{i(1)}), \ldots, \kappa_{r\_ab}(A_{i(k)}). \tag{9}$$

It is

$$\begin{aligned}(\pi_\beta(R), Alice) =& \pi_{\kappa_{r\_a}(\beta)}(\kappa_{r\_a}(R_A)) \cup \pi_{\kappa_{r\_ab}(\beta)}(\kappa_{r\_ab}(R_{AB})) \\ =& \kappa_{r\_a}(R'_A) \cup \kappa_{r\_ab}(R'_{AB}).\end{aligned} \tag{10}$$

Rename. A *rename* ρ of an attribute $A_i \in R$ to Q issued by Alice is executed on the encrypted virtual relations $\kappa_{r\_a}(R_A)$ and $\kappa_{r\_ab}(R_{AB})$. The new attribute name Q is encrypted with key *r_a* as $\kappa_{r\_a}(Q)$ and with key *r_ab* as $\kappa_{r\_ab}(Q)$ respectively. It replaces the encrypted original attribute name A_i in the virtual relations. It is

$$\begin{aligned}(\rho_{Q\leftarrow A_i}(R), \mathit{Alice}) =&\rho_{\kappa_{r\_a}(Q)\leftarrow\kappa_{r\_a}(A_i)}(\kappa_{r\_a}(R_A))\\ &\cup\ \rho_{\kappa_{r\_ab}(Q)\leftarrow\kappa_{r\_ab}(A_i)}(\kappa_{r\_ab}(R_{AB})).\end{aligned} \tag{11}$$

A *rename* is not persisted.

Count. The aggregate function ${}_{\beta}\gamma_{Count(A_i)}(R)$ on a relation R issued by Alice is executed on the encrypted virtual relations $\kappa_{r\_a}(R_A)$ and $\kappa_{r\_ab}(R_{AB})$. It is

$$\begin{aligned}&({}_{\beta}\gamma_{Count(A_i)}(R), \mathit{Alice})\\ =&{}_{\kappa_{r\_a}(\beta)}\gamma_{Count(\kappa_{r\_a}(A_i))}(\kappa_{r\_a}(R_A))\\ &+{}_{\kappa_{r\_ab}(\beta)}\ \gamma_{Count(\kappa_{r\_ab}(A_i))}(\kappa_{r\_ab}(R_{AB})).\end{aligned} \tag{12}$$

with $Count$ the aggregate function executed on server-side. It counts the numbers of attribute values of A_i for the virtual relations R_A and R_{AB} separately and adds these partial results on the server. The output represents the number of attribute values of attribute A_i accessible by Alice.

Set Union. Let relations R and S have the same set of attributes. A *set union* $R \cup S$ issued by Alice is executed on the encrypted virtual relations $\kappa_{r\_a}(R_A)$, $\kappa_{r\_ab}(R_{AB})$, $\kappa_{s\_a}(S_A)$, and $\kappa_{r\_ab}(S_{AB})$. It is

$$\begin{aligned}(R \cup S, \mathit{Alice}) =&\{\kappa_{r\_a}(t)|t \in R_A\} \cup \{\kappa_{r\_ab}(t)|t \in R_{AB}\}\\ &\cup\ \{\kappa_{s\_a}(t)|t \in S_A\} \cup \{\kappa_{r\_ab}(t)|t \in S_{AB}\}.\end{aligned} \tag{13}$$

Cartesian Product. Let r be a tuple of relation R and s be a tuple of relation S. A cartesian product $R \times S$ issued by Alice is executed on the encrypted virtual relations $\kappa_{r\_a}(R_A)$, $\kappa_{r\_ab}(R_{AB})$, $\kappa_{s\_a}(S_A)$, and $\kappa_{s\_ab}(S_{AB})$. It is

$$\begin{aligned}(R \times S, \mathit{Alice}) =&\{\kappa_{r\_a}(r)\kappa_{s\_a}(s) \vee \kappa_{r\_a}(r)\kappa_{s\_ab}(s)\\ &\vee \kappa_{r\_ab}(r)\kappa_{s\_a}(s) \vee\ \kappa_{r\_ab}(r)\kappa_{s\_ab}(s)\\ &|r \in (R_A \vee R_{AB}) \wedge\ s \in (S_A \vee S_{AB})\}.\end{aligned} \tag{14}$$

ENKI also supports queries to update, delete, or insert a tuple as well as queries to modify the table schema. The introduced rewriting techniques can be directly applied to update and delete a tuple. Schema modification and insertion of a tuple require different rewriting strategies. In case a user modifies the schema of one relation, the query must be rewritten to modify the schemas of all its virtual relations. In case a user inserts a tuple in a relation, the tuple can only be inserted in one of the virtual relations the user is allowed to access. To rewrite the insert query, exactly one virtual relation must be specified. This depends on the access restrictions defined by the data owner.

The computational complexity to rewrite and encrypt a query for the multi user mode depends on the number of operands seen in the original query (i.e. number of involved relations) and the number of user groups a user participates in. Assume that the computational complexity is $\mathcal{O}(s)$ for unary and binary operations in the single user mode. The computational complexity to rewrite and encrypt an unary operation is linear given k user groups. Each query is rewritten and encrypted to be executed on k virtual relations resulting in a computational complexity of $\mathcal{O}(s \times k)$. The computational complexity to rewrite and encrypt a binary operation is quadratic given that a user is granted access to k virtual relations of each relation respectively. Each query is rewritten and encrypted to be executed on one of k^2-pairs of the involved virtual relations resulting in a computational complexity of $\mathcal{O}(s \times k^2)$.

4.2 Proxy Re-Encryption as Relational Operation

Virtual relations are encrypted with different encryption keys which prevent comparisons of tuples even if a deterministic encryption scheme is used. However, comparisons are necessary to compute the unary operation *count distinct* or the binary operations *equi-join* and *set difference* over (deterministically) encrypted data. Our goal is to proxy re-encrypt virtual relations on the database server so that all queried attributes share the same encryption key while preserving data confidentiality. To formalize this approach, we define proxy re-encryption as a relational operation.

Definition 2. A *proxy re-encryption* alters an attribute $\kappa_z(A_i)$ encrypted with key z to enable its decryption with another key y. We define a proxy re-encryption $\chi_y(\kappa_z(A_i))$ of attribute A_i as

$$\begin{aligned}\chi_y(\kappa_z(A_i)) :&= \{\chi_y(\kappa_z(t_{i_k}))|t_{i_k} \in A_i \text{ for all } k = 1, \ldots, j\}\\ &= \{\kappa_y(t_{i_k})|t_{i_k} \in A_i \text{ for all } k = 1, \ldots, j\}\\ &= \kappa_y(A_i)\end{aligned} \tag{15}$$

with t_{i_k} the attribute values of A_i for all $k = 1, \ldots, j$.

The proxy re-encryption of a relation $\kappa_z(R)$ is the proxy re-encryption of all attributes. It is

$$\begin{aligned}\chi_y(\kappa_z(R)) :&= R(\chi_y(\kappa_z(A_1)), \ldots, \chi_y(\kappa_z(A_n)))\\ &= R(\kappa_y(A_1), \ldots, \kappa_y(A_n)) = \kappa_y(R).\end{aligned} \tag{16}$$

Definition 3. A proxy re-encryption is called *symmetric* if

$$\chi_b(\kappa_a(A_i)) = \kappa_b(A_i) \Leftrightarrow \chi_a(\kappa_b(A_i)) = \kappa_a(A_i). \tag{17}$$

Definition 4. A proxy re-encryption is called *transitive* if

$$\begin{aligned}&\chi_b(\kappa_a(A_i)) = \kappa_b(A_i) \wedge \chi_c(\kappa_b(A_i)) = \kappa_c(A_i)\\ \Rightarrow&\chi_c(\kappa_a(A_i)) = \kappa_c(A_i).\end{aligned} \tag{18}$$

A symmetric and transitive proxy re-encryption scheme ensures privacy-preserving computations on the database server in the single user mode [25]. However, if you recall the problem statement in Section 2, it does not preserve data confidentiality in the face of multiple users as its application leads to a data compromise. This motivates the introduction of a non-symmetric and non-transitive proxy re-encryption scheme called DETPRE as the cryptographic primitive for *count distinct*, *equi-joins*, and *set differences* in multi user mode.

Definition 5. A deterministic proxy re-encryption scheme is a tuple of algorithms $ParamGen$, $KeyGen$, Enc, $Token$, Pre such that:

- **Parameter Generation.** The probabilistic polynomial time algorithm $ParamGen$ takes as input the security parameter λ and outputs system parameters $params$: $params \leftarrow ParamGen(1^\lambda)$.
- **Key Generation.** The probabilistic polynomial time algorithm $KeyGen$ takes as input the security parameter λ and outputs a key k_i: $k_i \leftarrow KeyGen(1^\lambda)$.
- **Encryption.** The deterministic polynomial time algorithm Enc takes as input a plaintext m and key k_i and outputs a ciphertext: $C = Enc(m, k_i)$.
- **Token.** The deterministic polynomial time algorithm $Token$ takes as input two keys k_i, k_j and outputs a token to proxy re-encrypt k_i to k_j: $T = Token(k_i, k_j)$.

- **Proxy Re-Encryption.** The deterministic polynomial time algorithm Pre takes as input a ciphertext C and a token T and outputs a ciphertext C': $C' = Pre(C, T)$.

We now present our deterministic proxy re-encryption scheme DETPRE by specifying each algorithm.

ParamGen. Given a security parameter λ, $ParamGen$ works as follows: we generate a prime p and two groups $\mathbb{G}_1, \mathbb{G}_2$ of order p, and a bilinear, non-degenerated, computable map $e : \mathbb{G}_1 \times \mathbb{G}_1 \longrightarrow \mathbb{G}_2$. We choose a generator $G \in \mathbb{G}_1$ uniformly at random.

KeyGen. We choose $k_i \in \mathbb{Z}_p$ uniformly at random.

Enc. We encrypt a plaintext m with a key k_i and compute a ciphertext C as

$$C = G^{mk_i} \in \mathbb{G}_1. \tag{19}$$

Token. We generate a token T that proxy re-encrypts a ciphertext encrypted with a key k_i to be encrypted with a key k_j and compute

$$T = G^{\frac{k_j}{k_i}} \in \mathbb{G}_1. \tag{20}$$

Pre. We proxy re-encrypt a ciphertext C encrypted with a key k_i to a ciphertext C' encrypted with a key k_j and compute

$$\begin{aligned} C' = e(C, T) =& e(G^{mk_i}, G^{\frac{k_j}{k_i}}) = e(G, G)^{mk_i \frac{k_j}{k_i}} = e(G, G)^{mk_j} \\ =& g^{mk_j} \in \mathbb{G}_2. \end{aligned} \tag{21}$$

DETPRE is single-hop meaning that a ciphertext can only be proxy re-encrypted once. This restricts its usability as the key of a once proxy re-encrypted ciphertext is persisted. Therefore, we propose the following strategy which allows to benefit from the application of DETPRE while maintaining its re-usablity:

1. We encrypt all attribute values using the algorithm Enc. These encrypted attribute values are called *base values*.

2. If a proxy re-encryption is required, we use the algorithm Pre and proxy re-encrypt the base values with a temporary key c. The proxy re-encrypted results are called DETPRE *values*.

3. We store the DETPRE *values* temporarily as a concatenation to the base values and use them to process a relational operation.

4. After the user logs out, the DETPRE *values* are deleted.

We now describe our adversary model to informally explain the security guarantees our proxy re-encryption schemes provides.

We consider a passive adversary, i.e., the adversary can read all encrypted attribute values of all users, but does not modify them. We assume that the adversary has also compromised the application and its database proxy and observes executed operations. In particular, if a user is compromised during the attack, the adversary learns the user's masterkey, the encryption keys stored in the key store, and the used tokens of a user. Our goal is to prevent an adversary from using this information to learn private data of non-compromised users.

Therefore, we assume a number of users distributed to n user groups with each user group endowed with an encryption key $d_1, \ldots, d_n$ which are kept private.

We allow the adversary to compromise all but one encryption key. The adversary could have learned these keys as a result of a collusion between service provider and personnel working for the data owner. Therefore, she has access to keys $\{d_1, \ldots, d_{n-1}\}$ but not to key d_n. It implies that the adversary can decrypt all database entries encrypted with the keys $d_1, \ldots, d_{n-1}$. In particular, if d_n is the private key of a single user i.e. of a user group with only one member and this user also participates in additional user groups with compromised members, then the adversary can decrypt all tuples encrypted for these user groups but cannot decrypt the tuples encrypted with d_n.

The adversary can compute or learn tokens $Token(d_{i^*}, d_i)$ for all compromised keys $d_{i^*} \in \{d_1, \ldots, d_{n-1}\}$ to be proxy re-encrypted to an arbitrary key d_i. Thereby, she can proxy re-encrypt the database entries encrypted with the compromised encryption keys $d_1, \ldots, d_{n-1}$.

The database entries encrypted with key d_n are not compromised and the adversary cannot access these database entries. She also cannot compute or learn tokens $Token(d_n, d_i)$ which proxy re-encrypt the database entries encrypted with key d_n to an arbitrary key d_i. However, she can compute tokens $Token(d_i, d_n)$ such that a database entry encrypted with an arbitrary key d_i can be proxy re-encrypted to a key d_n.

Given all these information, the adversary should not be able to proxy re-encrypt an attribute value encrypted with the encryption key d_n to another key. We refer to this property as non-reversion.

Next, we study a security game to formally define the described security guarantees and proof our claims based on a known hardness assumption.

Let $\mathcal{A}$ be a probabilistic time adversary modeled as described above. Let $\mathcal{C}$ be the challenger. Then consider the following security game for a security parameter λ:

Setup. $\mathcal{C}$ takes a security parameter λ, runs algorithm $ParamGen$, and returns the system parameters $params$ to $\mathcal{A}$. $\mathcal{C}$ also runs algorithm $KeyGen$ and outputs keys $d_1, \ldots, d_n$. $\mathcal{C}$ sends $d_1, \ldots, d_{n-1}$ to $\mathcal{A}$ and keeps d_n as a secret. $\mathcal{C}$ runs algorithm $Token$ and outputs $Token(d_i, d_j)$ for all $i, j = 1, \ldots n$ that allow a database entry encrypted with the key d_i to be proxy re-encrypted to the key d_j. $\mathcal{C}$ sends Token(d_{i^*}, d_i) with $i^* = 1, \ldots, n-1$ and $i = 1, \ldots, n$ to $\mathcal{A}$.

Phase 1. $\mathcal{A}$ performs actions $q_1, \ldots, q_m$ where q_i is one of the following type:

Enc $\mathcal{A}$ chooses an arbitrary value s and runs algorithm Enc. Thereby, $\mathcal{A}$ encrypts s with key d_{i^*} for $i^* = 1, \ldots, n-1$. (He knows the keys $d_1, \ldots, d_{n-1}$ of all compromised users.) Although he does not know key d_n, he can also encrypt an arbitrary value with key d_n as he can compute

$$Token(d_{i^*}, d_n) d_{i^*} = (G^{\frac{d_n}{d_{i^*}}})^{d_i^*} = G^{d_n} \tag{22}$$

given a $Token(d_{i^*}, d_n)$ and a key d_{i^*} to encrypt a chosen value s under the uncompromised key d_n as G^{md_n}.

Pre $\mathcal{A}$ runs algorithm Pre to proxy re-encrypt a ciphertext $C = G^{d_i m}$ with a token $Token(d_i, d_j) = G^{\frac{d_j}{d_i}}$ as

$$\begin{aligned} Pre(C, T) = e(G^{d_i m}, G^{\frac{d_j}{d_i}}) = e(G^{d_i m}, G^{\frac{d_j}{d_i}}) \\ = e(G, G)^{d_i m \frac{d_j}{d_i}} = e(G, G)^{d_j m} = g^{d_j m}. \end{aligned} \tag{23}$$

Challenge. $\mathcal{A}$ chooses a key d and sends it to $\mathcal{C}$. $\mathcal{C}$ picks a random value r and encrypts it with key d_n as G^{rd_n}. It sends G^{rd_n} to $\mathcal{A}$ and asks him to proxy re-encrypt $C = G^{rd_n}$ to key d

as C_d.

Phase 2. $\mathcal{A}$ performs further actions $q_{m+1}, \ldots, q_n$ of the types described above.

Guess. $\mathcal{A}$ outputs its guess C'_d and wins the security game if and only if $C_d = C'_d$.

The advantage of $\mathcal{A}$ in the security game is defined as

$$Pr[C_d = C'_d] = \epsilon.$$

We proof the security of DETPRE in the Appendix. The remaining of this section presents the proxy re-encryption and rewriting strategies to execute the relational operations *count distinct*, *set difference*, and *equi-join*.

Count Distinct The aggregate function count distinct

$$_\beta\gamma_{CountDistinct(A_i)}(R)$$

on a relation R issued by Alice is executed on the encrypted virtual relations $\kappa_{r\_a}(R_A)$ and $\kappa_{r\_ab}(R_{AB})$. As these virtual relations are encrypted with different keys, it is not possible to apply a count distinct. So, we adjust the key of both virtual relations to key c. It is

$$\begin{aligned}\chi_c(\kappa_{r\_a}(R_A)) &= R_A(\chi_c(\kappa_{r\_a}(A_1)), \ldots, \chi_c(\kappa_{r\_a}(A_n))) \\ &= R_A(\kappa_c(A_1), \ldots, \kappa_c(A_n)) = \kappa_c(R_A)\end{aligned} \tag{24}$$

and $\chi_c(\kappa_{r\_ab}(R_{AB})) = \kappa_c(R_{AB})$ respectively. The aggregate function count distinct is then computed as

$$\begin{aligned}&(_\beta\gamma_{CountDistinct(A_i)}(R), Alice) \\ &=_{\kappa_c(\beta)}\gamma_{CountDistinct(\kappa_c(A_i))}(\kappa_c(R_A) \cup \kappa_c(R_{AB})).\end{aligned} \tag{25}$$

It counts the number of different attribute values of attribute A_i for all virtual relations accessible by user Alice.

Set Difference. Let relations R and S have the same set of attributes. A set difference $R \backslash S$ of relation S in relation R issued by Alice is executed on the encrypted virtual relations $\kappa_{r\_a}(R_A)$, $\kappa_{r\_ab}(R_{AB})$, $\kappa_{s\_a}(S_A)$, and $\kappa_{s\_ab}(S_{AB})$. As these virtual relations are encrypted with different keys, it is not possible to apply a set difference. Therefore, we adjust the keys of all virtual relations to key c. It is

$$\begin{aligned}\chi_c(\kappa_{r\_a}(R_A)) &= R_A(\chi_c(\kappa_{r\_a}(A_1)), \ldots, \chi_c(\kappa_{r\_a}(A_n))) \\ &= R_A(\kappa_c(A_1), \ldots, \kappa_c(A_n)) = \kappa_c(R_A)\end{aligned} \tag{26}$$

and $\kappa_c(R_{AB})$, $\kappa_c(S_A)$, and $\kappa_c(S_{AB})$ respectively. Then, we apply the set difference on the proxy re-encrypted virtual relations. It is

$$R \backslash S = \{\kappa_c(t) | t \in (R_A \vee R_{AB}) \wedge t \notin (S_A \vee S_{AB})\}. \tag{27}$$

Equi-Join. An *equi-join* issued by user Alice between two relations $R = R(A_1, \ldots, A_n)$ and $S = S(B_1, \ldots, B_m)$ on their respective attributes A_i and B_j is executed on the encrypted virtual relations $\kappa_{r\_a}(R_A)$, $\kappa_{r\_ab}(R_{AB})$, $\kappa_{s\_a}(S_A)$, and $\kappa_{s\_ab}(S_{AB})$.

However, the attributes A_i and B_j are encrypted with different keys within the relations (A_i is encrypted with key r_a in relation R_A and with key r_ab in relation R_{AB} and B_j is encrypted with key s_a in relation S_A and with key s_ab in relation S_{AB}). To apply the condition $A_i = B_j$ on the accessible subsets of R and S, we have to proxy re-encrypt all virtual relations and encrypt the condition with a shared encryption key. It is

$$\chi_c(\kappa_{r\_a}(R_A)) = \kappa_c(R_A). \tag{28}$$

and $\kappa_c(R_{AB})$, $\kappa_c(S_A)$, and $\kappa_c(S_{AB})$ respectively.

We encrypt the condition $A_i = B_j$ as $\kappa_c(A_i) = \kappa_c(B_j)$ and execute the equi-join as

$$\begin{aligned}(R \bowtie_{A_i=B_j} S, Alice) =&\{\kappa_c(r)\kappa_c(s) | r \in (R_A \vee R_{AB}) \\ &\wedge\ s \in (S_A \vee S_{AB}) \wedge \kappa_c(r_i)\theta\kappa_c(s_j)\}.\end{aligned} \tag{29}$$

The computational complexity to proxy re-encrypt a query for the multi user mode depends on the computational complexity of our introduced proxy re-encryption scheme and on the cardinality j of the involved k virtual relations. The computational complexity of DETPRE is the computation of one pairing operation for each of the $j \times k$ attribute values. The computational effort increases compared to the operations applied in the single user mode as the pairing operation is more expensive [22, 25]. In addition, optimization strategies can minimize the number of attribute values that have to be re-encrypted in the single user mode [17]. These are not feasible in the multi user mode.

4.3 Client-Server Split

Aggregate functions *count*, *count distinct*, *group by*, *sum*, *average*, *maximum*, *minimum*, and *sort* compute key figures over a whole relation. The encrypted processing of aggregation results is supported on the server in the single user mode [25]. In Subsection 4.1 and Subsection 4.2, we introduced the server-side execution of *count* and *count distinct* in the multi user mode.

Now, we explain the execution of the rest of these aggregate functions. Introducing virtual relations to specify access restrictions, aggregate functions cannot be executed on the whole relation as this relation is split into different virtual relations encrypted with different keys. In order to evaluate an aggregate function, a user has to process the aggregate function over all virtual relations she is allowed to access. These virtual relations are encrypted with different keys. Typically, the evaluation of aggregation results requires that all invoked virtual relations are encrypted with a shared encryption key.

One possible solution is a proxy re-encryption on the server to compute the aggregation results. Such proxy re-encryption schemes must be suitable for the encryption scheme required by the aggregate function. Unfortunately, some can be hard to construct [25] while others require notable computational effort and execution time [13].

Another naive solution processes the aggregate functions on the client. This generates significant data traffic and increases storage capacity as all data has to be transferred to and stored on the client. In addition, the client needs sufficient computational capacity to evaluate the aggregate function.

This in mind, we opt for a client-server split where a significant amount of computational effort is executed on encrypted data and small encrypted partial result sets are issued to the client where they are decrypted and further processed to receive the final result. Therefore, we split the execution of these aggregate functions between server and client as follows:

- On the server: Computation of the encrypted results for each virtual relation. These are the *partial results*.
- On the client: Decryption of the partial results and computation of a function $\mathcal{F}_{Agg}$ which takes as input the unencrypted partial results and computes the *final result* depending on the underlying aggregate function.

To illustrate this approach, consider an aggregate function $F(A_i)$ which computes *maximum*, *minimum*, *average*, *sum*, or *sort* over an attribute A_i. Let $\beta = (A_1, \dots, A_k)$ be an attribute list to *group* the results. If $\beta = \emptyset$, then there is no group-by function defined. An aggregate function ${}_{\beta}\gamma_{F(A_i)}(R)$ on a relation R issued by Alice is executed on the encrypted virtual relations $\kappa_{r\_a}(R_A)$ and $\kappa_{r\_ab}(R_{AB})$. Therefore, the attribute list β is encrypted with key r_a as $\kappa_{r\_a}(\beta)$ and with key r_ab as $\kappa_{r\_ab}(\beta)$. The function $F(A_i)$ is also encrypted with key r_a as $F(\kappa_{r\_a}(A_i))$ and with key r_ab as $F(\kappa_{r\_ab}(A_i))$. We compute the partial result for virtual relation R_A on the server as

$$\kappa_{r\_a}(Res(R_A)) = {}_{\kappa_{r\_a}(\beta)}\, \gamma_{F(\kappa_{r\_a}(A_i))}\kappa_{r\_a}(R_A) \tag{30}$$

and the partial result for virtual relation R_{AB} as

$$\kappa_{r\_ab}(Res(R_{AB})) = {}_{\kappa_{r\_ab}(\beta)}\, \gamma_{F(\kappa_{r\_ab}(A_i))}\kappa_{r\_ab}(R_{AB}). \tag{31}$$

The partial results $\kappa_{r\_a}(Res(R_A))$ and $\kappa_{r\_ab}(Res(R_{AB}))$ are sent to the client where they are decrypted. On the client, we compute the function $\mathcal{F}_{Agg}$ which takes as input the unencrypted partial results. It is

$$\mathcal{F}_{Agg} = Agg(Res(R_A), Res(R_{AB})) \tag{32}$$

the final result. Depending on the underlying aggregate function, the definition of $\mathcal{F}_{Agg}$ varies.

The rest of this subsection shows how to compute the partial and final result sets.

Maximum/Minimum. We compute the partial results on the server and decrypt them on the client. It is

$$Res(R_A) = Max(R_A) \text{ and } Res(R_{AB}) = Max(R_{AB}). \tag{33}$$

On the client, we compute the final result $\mathcal{F}_{Agg} = \mathcal{F}_{Max}$ as

$$\mathcal{F}_{Max} = Max(Max(R_A), Max(R_{AB})). \tag{34}$$

This also holds for the computation of the minimum.

Sum. We compute the partial results on the server and decrypt them on the client such that $Res(R_A) = Sum(R_A)$ and $Res(R_{AB}) = Sum(R_{AB})$. We compute $\mathcal{F}_{Agg} = \mathcal{F}_{Sum}$ as

$$\mathcal{F}_{Sum} = Sum(R_A) + Sum(R_{AB}) \tag{35}$$

on the client.

Average. During the query rewriting, the aggregate function *average* is replaced by the aggregate functions *sum* and *count*. This provides the partial results

$$\begin{aligned} Res(R_A) &= \{Sum(R_A), Count(R_A)\} \\ Res(R_{AB}) &= \{Sum(R_{AB}), Count(R_{AB})\}. \end{aligned} \tag{36}$$

We compute $\mathcal{F}_{Agg} = \mathcal{F}_{Avg}$ on the client as

$$\mathcal{F}_{Avg} = \frac{Sum(R_A) + Sum(R_{AB})}{Count(R_A) + Count(R_{AB})}. \tag{37}$$

Sort. The server-side computation and client-side decryption results in the partial results $Res(R_A) = Sort(R_A)$ and $Res(R_{AB}) = Sort(R_{AB})$. Both are sorted lists. On the client, we compute $\mathcal{F}_{Agg} = \mathcal{F}_{Sort}$ as

$$\mathcal{F}_{Sort} = Merge\_sorted\_lists(Sort(R_A), Sort(R_{AB})). \tag{38}$$

Group By. The aggregate function *group by* provides as partial results the grouped results for virtual relations R_A and R_{AB}. There are

$$\begin{aligned} Res(R_A) &= \{Agg(R_A) \text{ grouped by } R_A.A_i\} \\ Res(R_{AB}) &= \{Agg(R_{AB}) \text{ grouped by } R_{AB}.A_i\}. \end{aligned} \tag{39}$$

On the client, we process the partial results Res(R_A) and Res(R_{AB}) as follows. If $R_A.A_i = R_{AB}.A_i$, we merge these groups of R_A and R_{AB} and include it in the final result. If $R_A.A_i \neq R_{AB}.A_i$, we overtake the partial result in the final result.

The client-server split does not increase the computational complexity of a query as this technique only distributes the computations between client and server. However, the data traffic increases the communication complexity.

4.4 Multi User Algorithm

We apply these introduced techniques and present a multi user algorithm allowing a user to execute a query over a set of access restricted relations.

It takes as input a user id and an unencrypted query and returns the final result of the query as output. The user id is an identifier unique for each user. A query is a combination of relational operations over one or more relations. The final result is the decrypted result of the query.

Consider a relation R with attributes $A_1, \dots, A_n$ and a relation S with attributes $B_1, \dots, B_m$. The data owner splits the relation R in virtual relations $R_1, \dots, R_k$ and encrypts them with keys $v_1, \dots, v_k$. She also splits the relation S in virtual relations $S_1, \dots, S_l$ and encrypts them with keys $w_1, \dots, w_l$ respectively.

The data owner handles n user. Each user is equipped with a user id. The data owner defines the user group mapping where each user id is related to its user groups. She also defines the virtual relation mapping where each pair of user group and relation is assigned to a virtual relation. Here, we focus on a user which is member of $i + j$ different user groups. For relation R, the user is member of user groups which are assigned to the virtual relations $\kappa_{v_1}(R_1), \dots, \kappa_{v_i}(R_i)$ and for relation S, the user is member of user groups which are assigned to the virtual relations $\kappa_{w_1}(S_1), \dots, \kappa_{w_j}(S_j)$.

With respect to the specific query, the multi user algorithm requires six steps:

1. Look Up: Determine the required virtual relations given query and user id and check if the required attributes are encrypted with the necessary encryption layer.
2. Proxy Re-Encryption: Initiate a proxy re-encryption if the query contains *count distinct*, *equi join*, or *set difference*.
3. Query Encryption: Encrypt all elements of the rewritten query like attributes, conditions, attribute lists.
4. Query Rewriting: Run the query rewriting algorithm presented in Algorithm 1 to adapt the query to multi user mode.
5. Server-side Execution: Process the rewritten query over encrypted data and return the encrypted results to the client.
6. Client-side Execution: Decrypt the returned results and do further processing if required.

We now explain the details of these steps given an arbitrary query and a user id.

Look Up. ENKI Query Adapter checks the user group mapping to identify all user groups containing this user id. Given these user groups and the relation(s) contained in the query, it determines all virtual relations for each pair of user group and relation(s) in the virtual relation mapping.

Proxy Re-Encryption. If the query contains *equi-join*, *set difference*, or *count distinct*, an UDF adjusts the keys of all involved virtual relations to a temporary key c.

Query Encryption. ENKI Query Adapter encrypts all attributes. If there exists a condition $a\theta b$ with a, b attributes, then it encrypts the attributes a and b of relations accessible by this user with the same key c. If a is an attribute and b is a constant, it encrypts $a\theta b$ with all keys $v_1, \ldots, v_i$. The attribute list $\beta = A_{i_{(1)}}, \ldots, A_{i_{(k)}}$ of a projection or an aggregate function is encrypted with keys $v_1, \ldots, v_i$ of the virtual relations $R_1, \ldots, R_i$ accessible by this user. This step also encrypts the aggregate function $F(A_i)$ with keys $v_1, \ldots, v_i$.

Query Rewriting. ENKI Query Adapter includes a query rewriting algorithm that modifies the original query to be executable over the required virtual relations

$$\kappa_{v_1}(R_1), \ldots, \kappa_{v_i}(R_i) \text{ and } \kappa_{w_1}(S_1), \ldots, \kappa_{w_j}(S_j).$$

We describe the details of this algorithm in Algorithm 1. It takes as input the query Q which can contain one or more unary or binary operations over relation R (and relation S). It returns a rewritten query sQ to be executed on the server and in some cases also a rewritten query cQ to be executed on the client.

Server-side Execution. The server executes the rewritten, encrypted query sQ and returns the encrypted results to the client. If client-side processing is necessary, the server also returns a query cQ

Client-side Execution. The client receives the encrypted results and decrypts them. If the client does not receive a query cQ, the query processing is finished. If the client receives a query cQ, it executes the query over the decrypted partial results receiving the final result.

Algorithm 1 Query Rewriting Algorithm

```
Require: $\kappa_{v_1}(R_1), \ldots, \kappa_{v_i} R_i, (\kappa_{w_1}(S_1), \ldots, \kappa_{w_j}(S_j))$: virtual relations
    $Q$: query containing one or more relational operations $\Delta$
for all $avg \in Q$ do
    rewrite ${}_\beta\gamma_{Avg(A_i)}(R)$ as $Q_k = {}_\beta\gamma_{Sum,Count(A_i)}(R)$
    generate cQ
end for
for all $\rho \in Q$ do
    rewrite $\rho(R)$ as $Q_t = \rho(\kappa_{v_k}(R_k))$ for all $k, t = 1, \ldots, i$
end for
for all unary $\sigma \in Q$ do
    rewrite $Q_t = \sigma(R)$ as $Q_t = \sigma(\kappa_{v_k}(R_k))$ for all $k, t = 1, \ldots, i$
end for
for all $\pi \in Q$ do
    rewrite $\pi(R)$ as $Q_t = \pi(R_k)$ for all $k, t = 1, \ldots, i$
end for
for all $max \vee min \vee sum \in Q$ do
    rewrite $\gamma_{F(R.A_i)}{}^{\Delta}R$ as
    $Q_t = \gamma_{F(R_k.A_i)}{}^{\Delta}(\kappa_{v_k}(R_k))$ for all $k, t = 1, \ldots, i$
    if $\exists cQ$ then
        modify cQ
    else
        generate cQ
    end if
end for
for all $\cup \vee \setminus \vee \times \vee \bowtie \in Q$ do
    rewrite $\Delta(R, S)$ as $Q_t = \Delta(R_k, S_l)$ for all $k = 1, \ldots, i$, $l = 1, \ldots, j$, and $t = 1, \ldots, i * j$
end for
for all $sort \vee group \in Q$ do
    rewrite ${}_\beta\gamma_{F(R.A_i)}{}^{\Delta(Q_t)}$ as ${}_\beta\gamma_{F(R_k.A_i)}{}^{\Delta(Q_t)}$
    for all $t = 1, \ldots, i$ in case Q unary or $t = 1, \ldots, i * j$ in case Q binary
    if $\exists cQ$ then
        modify cQ
    else
        generate cQ
    end if
end for
if Q unary then
    rewrite $\Delta R$ as $sQ = \bigcup_{t=1,\ldots,i} Q_t$
end if
if Q binary then
    rewrite $\Delta(R, S)$ as $sQ = \bigcup_{t=1,\ldots,i*j} Q_t$
end if
for all $count\ distinct \vee count \in Q$ do
    rewrite ${}_\beta\gamma_{F(R_k.A_i)}(sQ)$ as $sQ = {}_\beta\gamma_{F(R_k.A_i)}sQ$
end for
return sQ, (cQ)
```

5. KEY MANAGEMENT AND DYNAMIC ACCESS CONTROL POLICIES

ENKI enforces access policies through selective encryption leading to different keys for each user. However, access policies (and thereby keys) might change: a data owner grants access rights to new users or revokes access rights from others. Adding or deleting users of a user group can be formalized as changes in a user hierarchy.

Definition 6. Given the set of users $S = \{s_1, \ldots, s_n\}$ a user hierarchy $\mathcal{U}$ is a pair $(\mathcal{P}^*(S), \prec)$ where $\mathcal{P}^*(S)$ is the powerset without the empty set of S and $\prec$ is a partial order such that for all sets of users $p_i, p_j \in \mathcal{P}^*(S)$, $p_i \prec p_j$ if $p_j \subseteq p_i$ for all $i, j = \{1, \ldots 2^{n-1}\}$.

All user groups $p_i \in \mathcal{P}^*(S)$ with a non-empty object set such that

$$O(p_i) = \{o | o \in O \wedge QS_o = p_i\} \neq \emptyset \tag{40}$$

are called *busy user groups*. These are user groups granted access to a set of objects specified by an access policy. These busy user groups might also change when adding or deleting users.

Adding a new user s_{n+1} changes the original user hierarchy $\mathcal{U}$ by adding sets of users. These are a set with only one element s_{n+1} and sets $p_i \cup s_{n+1}$ for all $i = \{1, \ldots 2^n - 1\}$. The result is a new user hierarchy.

Consider all busy user groups p_i^{orig} in the original user hierarchy. Then for each user group p_i^{orig}, the new hierarchy contains a user group $p_i^{new} = p_i^{orig}$ and a user group $p_i^{orig} \cup s_{n+1}$. Consider two cases: First, a busy user group p_i^{orig} evolves to a non busy user group p_i^{new} in the new user hierarchy and user group $p_i^{orig} \cup s_{n+1}$ is busy in the new user hierarchy. Thereby, the new user s_{n+1} has access to all objects accessible by user group p_i^{orig}. It is

$$O(p_i^{orig}) = O(p_i^{orig} \cup s_{n+1}). \tag{41}$$

The data owner shares the key of user group p_i^{orig} with user s_{n+1}.

Second, a busy user group p_i^{orig} of the original hierarchy is still a busy user group p_i^{new} in the new user hierarchy and user group $p_i^{orig} \cup s_{n+1}$ is also busy. Thereby, the new user s_{n+1} has access to a subset of objects accessible for user group p_i^{orig}. The object set $O(p_i^{orig})$ is split such that

$$O(p_i^{orig}) = O(p_i^{new}) \cup O(p_i^{orig} \cup s_{n+1}) \tag{42}$$

with

$$O(p_i^{new}) \cap O(p_i^{orig} \cup s_{n+1}) = \emptyset. \tag{43}$$

The data owner downloads the object set $O(p_i^{orig} \cup s_{n+1})$ and re-encrypts it with a new key.

We differentiate three scenarios where access rights are revoked from a user. First, a user is revoked from all access rights. Second, a user is revoked from a user group. Third, a user is revoked from certain objects of a user group.

Consider the first scenario where all access rights of a user are revoked. The original user hierarchy changes as the set of users S is reduced by one element s_n. This is to reduce

$$\mathcal{P}^*(\mathcal{S}) = \mathcal{P}^*(\mathcal{S} \backslash s_n) \cup (\mathcal{P}(\mathcal{S} \backslash s_n) \cup s_n) \tag{44}$$

to $\mathcal{P}^*(\mathcal{S} \backslash s_n)$ resulting in a new user hierarchy.

Consider all busy user groups $p_i^{orig} \cup s_n$ in the original user hierarchy. These user groups are deleted from the new hierarchy. Their object sets are then accessible by the user groups $p_i^{new} = p_i^{orig}$ and merged with the respective object sets as

$$O(p_i^{new}) = O(p_i^{orig} \cup s_n) \cup O(p_i^{orig}). \tag{45}$$

The data owner downloads all object sets $O(p_i^{orig} \cup s_n)$ and re-encrypts them with the respective keys of user groups p_i^{orig}.

Consider the second scenario where the user is revoked from a user group. This does not change the user hierarchy, but changes the busy user groups. Consider the respective busy user group $p_i^{orig} \cup$

s_n in the original user hierarchy. It is non busy in the new hierarchy. Its object set is then accessible by user group $p_i^{new} = p_i^{orig}$ and merged with the respective object set as

$$O(p_i^{new}) = O(p_i^{orig} \cup s_n) \cup O(p_i^{orig}). \tag{46}$$

The data owner downloads object set $O(p_i^{orig} \cup s_n)$ and re-encrypts it with the key of user group p_i^{orig}.

Consider the third scenario where the user is revoked from access of certain objects accessible by a user group. Consider the respective busy user group $p_i^{orig} \cup s_n$ in the original hierarchy. Revoking user s_n from accessing certain objects requires to split $O(p_i^{orig} \cup s_n)$ as

$$O(p_i^{orig} \cup s_n) = O(p_i^{new} \cup s_n) \cup O(p_i^{new}) \tag{47}$$

with

$$O(p_i^{new} \cup s_n) \cap O(p_i^{new}) \neq \emptyset. \tag{48}$$

Thereby, the user s_n has access to the object set $O(p_i^{new} \cup s_n)$ but cannot access the object set $O(p_i^{new})$. This results in two busy user groups $(p_i^{new} \cup s_n)$ and p_i^{new}. The data owner downloads object set $O(p_i^{new})$ and re-encrypts it with the key of user group p_i^{orig}.

The data owner updates the user group and virtual relation mapping according to the changes of user hierarchy and busy user groups to keep track of the changing users, user groups, and virtual relations. She also distributes the encryption keys to the respective users while updating their key stores when they are logged in.

In most cases, changing keys implies that the data has to be downloaded and re-encrypted in a trusted environment. In particular, each onion layer of the adjustable encryption has to be removed and re-encrypted using a new key. This is time-consuming and increases the data traffic. A proxy re-encryption on the server would save this overhead but has to prevent the untrusted service provider from learning the new encryption keys, computing arbitrary proxy re-encryptions, or gaining information about the onion layers. Currently no solution exists, particularly for order-preserving encryption.

6. EXPERIMENTAL EVALUATION

We implemented ENKI as the extension of an existing single user solution to support the multi user setting. We use a modified JDBC driver for the single user mode which receives unencrypted SQL queries, modifies their operator tree, performs the onion selection, and encrypts the results [15]. As described in Figure 1, ENKI is an additional modification of the JDBC driver to perform query rewriting for the multi user mode and provides a client add-on to execute the post-processing. Our experimental setup consists of a server and a client. The server is a HANA database server with 252 GByte RAM and 8-core 2.6 GHz processor. It hosts an unmodified SAP HANA database. The client has 16 GB RAM and 2-core 2.8GH processor. It hosts a modified JDBC proxy and an ENKI Query Adapter as well as a SQL-lite database. The queries are executed on the unmodified SAP HANA database [10] where UDFs execute cryptographic operations. We implemented DETPRE in C using pbc and gmp libraries providing the mathematical operations underlying pairing based encryption [20, 14]. We evaluate functionality and performance of ENKI on the TPC-C benchmark and three real world use cases described in Subsection 6.1. In Subsection 6.2, we analyze which types of queries and access policies can be supported. In Subsection 6.3, we evaluate the performance overhead consumed by the necessary modifications of ENKI.

6.1 Use Cases

IS-H. IS-H is the healthcare management solution of SAP for patient management. In our observed query trace, we see 7 tables with 477 columns in total. As all tables contain personal information, we assume that all tables must be treated confidential. The users accessing this application are typically associated to different roles which are sets of organizational units. Patient information is associated with the organizational units of her encounters. To protect sensitive patient information, access policies prevent users from accessing medical details of patients if they are not associated to the set of organizational units of the patient.

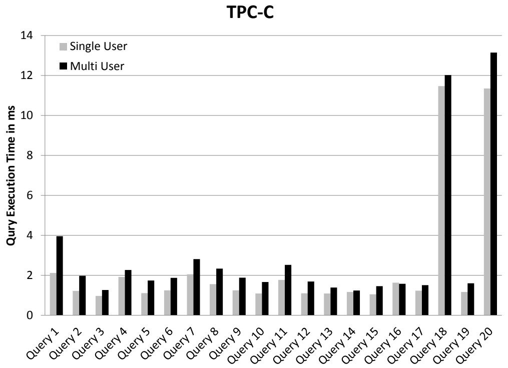

Figure 4: TPC-C: Query Execution Time for Single and Multi User Mode

Table 1: Overview of the query types in the use cases

Query Type	IS-H	LSM	TPC-C	SFIN
Equal	x	x	x	x
Range	x		x	x
Equal Join (binary)	x	x	x	x
Equal Join (tertiary)		x		
Range Join				
Aggregate	x	x	x	x

LSM. LSM is an internal SAP solution which supports facility management to plan resources. Peers on a certain SAP management level include confidential planning information for their area. The peers are only allowed to access the data they committed themselves but not the data of other peers. Facility management has access to all data and calculates figures for future resource planning which are sensitive. Focusing on our evaluation, we use the access policy specified for the facility management such that a user participates in n user groups given n users. The application contains of 25 tables and 173 columns.

TPC-C. TPC-C is an OLTP benchmark consisting of 9 tables and 92 columns. We assume that all tables and columns are sensitive and define an access policy for a two user scenario where each user has certain private data and other data is shared.

SFIN. Simplified Financials (SFIN) is part of SAP ERP application relying on SAP HANA as a database backend. In our use case, this application analyzes consumers' data sets consisting of 9 tables and 741 columns. We assume that all tables and columns are sensitive and define an access policy for a two user scenario where each user has certain private data and other data is shared.

6.2 Functional Evaluation

We analyzed the applications described in Subsection 6.1 to evaluate which queries and access policies ENKI can support.

Queries. Table 1 shows the issued query types for each application. ENKI supports all observed queries including equal and order

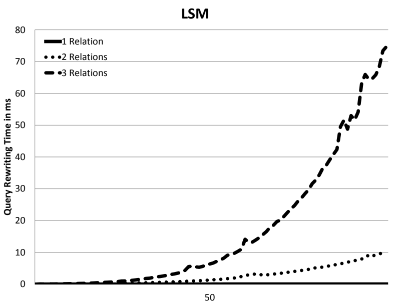

Figure 5: LSM: Query Rewriting for unary, binary, and tertiary relation(s) given $n = 5, \ldots, 100$ user groups

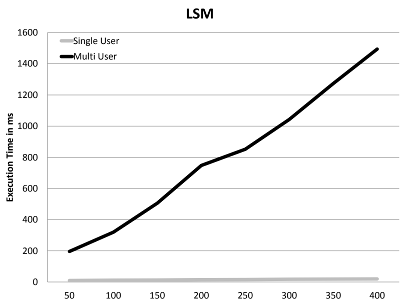

Figure 6: LSM: Query Execution Time for Single and Multi User Mode given $n = 50, \ldots, 400$ user groups

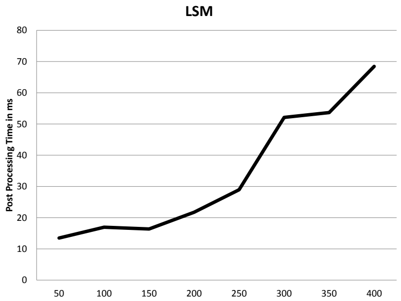

Figure 7: LSM: Post-Processing for Multi User Mode given $n = 50, \ldots, 400$ user groups

selections, equal joins, aggregations and combinations of these. In addition, ENKI also supports update, insert, and delete statements. Therefore, ENKI provides enhanced functionalities compared to existing solutions [30, 25, 24].

ENKI cannot support the execution of range joins on the database server if a range join includes columns of different virtual relations encrypted with different keys. To our knowledge, there is no proxy re-encryption scheme for OPE encryption available. Therefore, ENKI would execute range joins only on client-side. However, we consider this as acceptable as we did not observe a range join in any of our four applications.

Access Policies. ENKI supports the access policies specified for the IS-H and LSM application as its tuple-wise access restrictions on tables match well with the described requirements. This tuple-wise access enables the implementation of most of the access policies specified by authorization views [26]. An exception are those policies which only allow aggregated views on columns e.g. a user is only allowed to see the average of all attribute values of a column but not the unaggregated attribute values.

6.3 Performance Evaluation

We investigate two questions in order to evaluate the performance of ENKI:

- What is the performance penalty of our algorithm for the multi user mode compared to the single user mode?
- What is the performance impact of our proxy re-encryption scheme?

In the experiments to answer the first question we assume that the proxy re-encryption has already taken place.

Compared to the single user mode where an encrypted query is executed on encrypted data, our algorithm rewrites, executes, and post processes an encrypted query for the multi user mode. We analyze the time consumed by query rewriting, query execution, and post processing to better understand the performance penalty of the multi user mode.

Figure 5 shows the time consumption to rewrite unary, binary, and tertiary relational operations given the LSM access policy such that the number of involved virtual relations accessed by one user increases linearly with the number of additional users n. Figure 5 illustrates that the effort is $\mathcal{O}(n)$ for unary operations, $\mathcal{O}(n^2)$ for binary operations, and $\mathcal{O}(n^3)$ for tertiary operations.

Changing the access policy might increase the number of virtual relations assigned to a user such that she participates in more than n user groups. However, we did not observe such a worst case access policy in one of our use cases. In addition, we studied the literature, but did not find any case referring to this requirement [19]. In accordance with current literature, we even assume that the number of actual user groups is smaller than the number of users [19]. This implies that the evaluation of the LSM access policy with a linearly increasing number of user groups per user represents an upper boundary of the time consumption to rewrite unary, binary, and tertiary relational operations.

We further analyze the execution time for a mix of relational operations given the LSM access policy where the number of user groups linearly increases with the number of users. Figure 6 presents the consumed execution time given an increasing number of user groups $n = 50, \ldots, 400$. In the multi user mode, each additional user adds one more user group. Hence the query expands by an additional subquery for each additional virtual relation. This effort is reflected by the execution time ranging from 0.196 s for 50 user groups to 1.5s for 400 user groups. The query execution time in single user mode is nearly constant as the same query set is executed on a growing amount of data.

Figure 7 shows the effort to post-process unary relational operations including aggregation functions over an increasing number of user groups $n = 50, \ldots, 400$. These numbers contain the computational time for the client-server split as well as the necessary merge of the result sets on the client. Although the client-server split does not increase the computational complexity, we observe an overhead for post-processing which is moderately growing given an increasing number of user groups. On the client, the computations of *maximum*, *minimum*, and *sum* require $n - 1$ operations and the computation of *average* requires $2n - 1$ operations. The time to compute *sort* depends on the number of virtual relations n, but also on the maximum cardinality of all invoked virtual relations. It is $\mathcal{O}(m \log n)$ to merge n sorted lists with a total of m attribute values. The time to post-process the *group by* operation is also $\mathcal{O}(m \log n)$, a merge of m groups of n virtual relations. In addition, it is $\mathcal{O}(m - 1)$ to aggregate the partial results if similar groups exist.

Figure 4 shows the time consumption to rewrite, execute, and post process a mix of 20 select queries seen in the TPC-C benchmark and compares them to their execution time in the single user mode. Table 1 shows the query types of TPC-C. For the single user mode we execute the encrypted queries according to [25], i.e. without any access policy. For the multi user mode we use the same access policy as in the examples in this paper: there are two users and three user groups. Each user has access to his private data and both user have access to shared data. In order to execute a multi user mode query, we need to rewrite, execute, and post-process. We measure the time of these steps and compare their total to the execution time of the single user mode. Figure 4 presents the results for the 20 TPC-C queries. The multi user mode incurs an average overhead

Table 2: Microbenchmark of Encryption, Token Computation, and Proxy Re-Encryption of DetPre and JOIN-ADJ [25] over 10.000 Iterations

	DetPre	JOIN-ADJ
Encrypt	1.585956 ms	1.6058 ms
Token	0.0331148 ms	0.0014051 ms
PRE	1.019126 ms	0.000347 ms

of 36.98% (median overhead of 33.797%) compared to the single user mode. This is an absolute performance penalty of 0.6181 ms on average.

Figure 4 illustrates that Query 18 (including sum operator) and Query 20 (including range condition) both consume a significant larger amount of execution time compared to all other unary and binary relational operations in the single as well as in the multi user mode. This is caused by our implementation of the respective encryption schemes, but their performance could be further optimized [25].

For the second performance question we conduct another experiment.

We measure the execution time of our new encryption scheme DetPre used to process *count distinct*, *set difference*, and *join* securely over data encrypted with different keys in the multi user mode and compare it to the encryption scheme Join-Adj used in the single user mode [25].

We present a micro benchmark in Table 2 which contains the time to compute the three algorithms of the scheme: encryption, token computation, and proxy re-encryption. The time to encrypt data is almost equal in both schemes with DetPre consuming 1.5860 ms and Join-Adj consuming 1.6058 ms. The computation of the token consumes 0.03311 ms compared to Join-Adj with 0.0014 ms. The proxy re-encryption consumes 1.0191 ms in DetPre and 0.0003 ms in Join-Adj. This proxy re-encryption time multiplies with the cardinalities of all columns which have to be proxy re-encrypted.

It is possible to perform some computations in advance i.e. during the user logs in saving time during the execution. However, it is not possible to substitute DetPre with Join-Adj in the multi user case as proxy re-encryption in multi user mode which privacy-preserves data must be non-symmetric and non-transitive.

We see roughly a 40% increase on average per user group in multi-user mode for query rewriting, query execution, and post processing. We know from the literature that the total number of user groups scales linearly with the total number of users. In our experiments the absolute increase per user group was on average roughly 0.6 ms. If we assume that a user is not willing to wait longer than say 1 sec, we can accommodate 1500 user groups per query. This is sufficient for many practical examples as in our experiments.

The total execution time in multi user mode consists of 4% for query rewriting, 82% for query execution, and 14% for post processing. As the percentage of query execution is most significant, we will focus on its optimization strategies in future work.

If a query requires proxy re-encryption, our experiments show an absolute increase of roughly 1 ms per proxy re-encryption for one item. This number needs to be multiplied by the number of non-null rows, i.e. 100 sec for 100.000 items and 16.5 min for 1.000.000 items. We propose to perform the proxy re-encryption in advance saving time during the query execution. While the user is logged in, these DetPre values can be easily cached, but will not necessarily be persisted after the user logs out. Another option is to precompute the DetPre values based on an expected set of queries. In conclusion, our system scales well to a realistic number of user groups, but for large-sized databases proxy re-encryption should be persisted.

7. RELATED WORK

Queries over encrypted data. Encryption schemes supporting certain relational operations include key word search [28] or range queries [2, 5, 23, 18]. Some works provide data confidentiality using tuple-wise encryption and execute queries using indexes organized in buckets [16, 8]. Ciriani et al. satisfy privacy-constraints by (partial) encryption and fragmentation of data and rely on the application logic to process a query [6]. Popa et al. introduce adjustable query-based symmetric encryption to process queries on server-side [25]. Tu et al. propose an extension of this system to execute complex queries e.g. nested subqueries as seen in the TPC-C Benchmark by partitioning the query execution between server and client. We see no obstacle to combine this technique with the client-server split introduced by ENKI. Query processing with multiple keys without sharing data is presented in [25] and using searchable encryption in [30, 24, 3]. Ferretti et al. introduce a proxy concept to handle multiple users in the CryptDB setting, but do not present an experimental evaluation or a security analysis to proof their claims [11].

Joins over encrypted data. Deterministic encryption schemes that offer symmetric and transitive proxy re-encryption are presented in [22, 25]. Hacigumus et al. require extensive query rewriting to compute joins [16]. Agrawal et al. propose an interactive approach [1]. Furukawa et al. provide a non-transitive and non-symmetric approach to compute a join such that a probabilistic encryption is degraded to be deterministic in the single user mode [12]. The encryption scheme presented in [3] also handles join operations, but no confidentiality guarantees are provided.

Access Control. A system for encryption enforced access control for outsourced data is proposed in [9], but this solution does not support query execution on encrypted data. Rizvi et al. introduce authorization views that enable the specification of access policies using SQL queries on the application level [26]. This restricts the access of users but does not prevent a service provider or an intruder from learning the data stored on the database.

Key Management. The key management strategies introduced in [4, 7] can be combined with our access control model. ENKI can benefit from these strategies by a reduced number of keys a user has to store.

8. CONCLUSION

This paper presented ENKI, a system for securely executing relational operations on encrypted, access restricted data. ENKI introduces an encryption based access control model to enforce access restrictions on encrypted data using different encryption keys. ENKI uses query rewriting and post-processing to process relational operations over data encrypted with different encryption keys. It applies a newly introduced encryption scheme to execute the relational operations count distinct, set difference, and join while protecting data confidentiality. Our evaluation shows that its performance depends on the specified access policies and on the type of relational operation. It achieves modest overhead for the select queries of the TPC-C benchmark and the LSM use case.

9. REFERENCES

[1] R. Agrawal, A. Evfimievski, and R. Srikant. Information sharing across private databases. In *SIGMOD '03*, 2003.

[2] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *SIGMOD '04*, 2004.

[3] M. R. Asghar, G. Russello, B. Crispo, and M. Ion. Supporting complex queries and access policies for multi-user encrypted databases. In *Proceedings of the 2013 ACM Workshop on Cloud Computing Security Workshop*, CCSW '13, pages 77–88, New York, NY, USA, 2013. ACM.

[4] M. J. Atallah, K. B. Frikken, and M. Blanton. Dynamic and efficient key management for access hierarchies. In *CCS '05*, 2005.

[5] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. *IACR Cryptology ePrint Archive*, 2012.

[6] V. Ciriani, S. De Capitani Di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Keep a few: Outsourcing data while maintaining confidentiality. In *Proceedings of the 14th European Conference on Research in Computer Security*, ESORICS'09, pages 440–455, Berlin, Heidelberg, 2009. Springer-Verlag.

[7] E. Damiani, S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Key management for multi-user encrypted databases. In *StorageSS '05*, 2005.

[8] E. Damiani, S. D. C. Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing confidentiality and efficiency in untrusted relational dbmss. In *CCS '03*, 2003.

[9] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Over-enrcryption: Management of access control evolution on outsourced data. In *VLDB*, 2007.

[10] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The sap hana database – an architecture overview. *IEEE DATA Engineering Bulletins*, 35(1):28–33, 2012.

[11] L. Ferretti, M. Colajanni, and M. Marchetti. Access control enforcement on query-aware encrypted cloud databases. In *Proceedings of the 2013 IEEE International Conference on Cloud Computing Technology and Science - Volume 01*, CLOUDCOM '13, pages 717–722, Washington, DC, USA, 2013. IEEE Computer Society.

[12] J. Furukawa and T. Isshiki. Controlled joining on encrypted relational database. In *Pairing'12*, 2013.

[13] C. Gentry and S. Halevi. Implementing gentry's fully-homomorphic encryption scheme. In K. G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Sciences*, pages 129–148. Springer, 2011.

[14] T. Granlund and GMP Developement Team. *Manual of The GNU Multiple Precision Arithmetic Library*, 6.0.0 edition, March 2014.

[15] P. Grofig, M. Härterich, I. Hang, F. Kerschbaum, M. Kohler, A. Schaad, A. Schröpfer, and W. Tighzert. Experiences and observations on the industrial implementation of a system to search over outsourced encrypted data. In *Sicherheit*, pages 115–125, 2014.

[16] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *SIGMOD '02*, 2002.

[17] F. Kerschbaum, M. Härterich, P. Grofig, M. Kohler, A. Schaad, A. Schröpfer, and W. Tighzert. Optimal re-encryption strategy for joins in encrypted databases. In L. Wang and B. Shafiq, editors, *Data and Applications Security and Privacy XXVII*, volume 7964 of *Lecture Notes in Computer Science*, pages 195–210. Springer Berlin Heidelberg, 2013.

[18] F. Kerschbaum and A. Schröpfer. Optimal average-complexity ideal-security order-preserving encryption. In *to appear in: CSS '14*, 2014.

[19] M. Komlenovic, M. V. Tripunitara, and T. Zitouni. An empirical assessment of approaches to distributed enforcement in role-based access control (rbac). In *CODASPY*, 2011.

[20] B. Lynn. *PBC Library Manual*. Stanford University, 2007.

[21] Oracle Inc. Transparent data encryption: New technologies and best practices for database encryption. http://www.oracle.com/us/products/database/sans-tde-wp-178238.pdf,
2010.

[22] S. Pohlig and M. Hellman. An improved algorithm for computing logarithms over and its cryptographic significance (corresp.). *IEEE Trans. Inf. Theor.*, 2006.

[23] R. A. Popa, F. H. Li, and N. Zeldovich. An ideal-security protocol for order-preserving encoding. In *IEEE Symposium on Security and Privacy*, 2013.

[24] R. A. Popa and N. Zeldovich. Multi-key searchable encryption. *IACR Cryptology ePrint Archive*, 2013:508, 2013.

[25] R. A. Poppa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: Proctecting confidentiality with encrypted query processing. In *SOSP '11*, 2011.

[26] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD '04*, 2004.

[27] G. Saunders, M. Hitchens, and V. Varadharajan. Role-based access control and the access control matrix. *SIGOPS Oper. Syst. Rev.*, 35(4):6–20, Oct. 2001.

[28] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *SP '00*, 2000.

[29] F. Vercauteren, editor. *Final Report on Main Computational Assumptions in Cryptography*, number ICT-2007-216676. European Network of Excellence in Cryptography II, January 2013.

[30] Y. Yang, F. Bao, X. Ding, and R. H. Deng. Multiuser private queries over encrypted databases. *Int. J. Appl. Cryptol.*, 1(4):309–319, Aug. 2009.

APPENDIX

Definition 7. DETPRE is secure if for a probabilistic time adversary $\mathcal{A}$

$$Pr[\mathcal{A} \text{ wins the Security Game of } \textsc{DetPre}]$$

is negligible in λ.

ASSUMPTION 1. *The l-Bilinear Diffie-Hellman Inversion Assumption (l-BDHI) holds if for any probabilistic polynomial time (PPT) adversary $\mathcal{A}$ the probability that $\mathcal{A}$ on input*

$$G^a, G^{a^2}, G^{a^3}, \ldots, G^{a^l}$$

outputs W such that $W = g^{\frac{1}{a}}$ is negligible in a security parameter λ [29].

THEOREM 1. *If the L-BDHI assumption holds, then our proxy re-encryption is secure.*

PROOF. Assuming that an adversary can solve the described security game correctly, we construct a polynomial time algorithm which can solve the underlying problem of the l-Bilinear Diffie-Hellman Inversion assumption. This algorithm receives an instance of the l-BDHI problem with

$$G^a, G^{a^2}, \ldots, G^{a^l} \in \mathbb{G}_1$$

and has to compute $e(G,G)^{\frac{1}{a}} = g^{\frac{1}{a}} \in \mathbb{G}_2$.

Setup. Receive an instance of the l-BDHI problem as

$$p, e, \mathbb{G}_1, \mathbb{G}_2, G, g, G^a, G^{a^2}, \ldots, G^{a^l}$$

Choose $d_i \in \mathbb{Z}_p$ uniformly at random. Run algorithm $Token$ to compute $Token(d_i, d_j)$ with $i, j = 1, \ldots, n$. Send system parameters $p, \mathbb{G}_1, \mathbb{G}_2, e, G, g$, encryption keys $d_1, \ldots, d_{n-1}$, and tokens $Token(d_{i^*}, d_i)$ with $i^* = 1, \ldots, n-1$ and $i = 1, \ldots, n$ to $\mathcal{A}$.

Phase 1. $\mathcal{A}$ performs the following actions:

Enc $\mathcal{A}$ runs algorithm Enc to encrypt arbitrary messages m with keys $d_1, \ldots, d_{n-1}$. To encrypt message m with encryption key d_n, which is not known to $\mathcal{A}$, the adversary exploits its knowledge of encryption keys $d_1, \ldots, d_{n-1}$ and tokens $Token(d_{i^*}, d_n)$ to compute

$$G^{\frac{d_n}{d_{i^*}} d_{i^*}} = G^{d_n}. \tag{49}$$

Using this result, $\mathcal{A}$ computes G^{md_n}.

Pre Adv runs algorithm Pre to proxy re-encrypt ciphertext C encrypted with key d_{i^*} with $i^* = 1, \ldots, n-1$ to be encrypted with key d_i with $i = 1, \ldots, n$.

Challenge. $\mathcal{A}$ chooses a key $d \notin \{d_1, \ldots, d_n\}$ and sends it to $\mathcal{C}$. $\mathcal{C}$ picks a valid ciphertext as

$$C = \mathrm{Enc}(m, k) = G^{\tilde{m}a} = G^r \tag{50}$$

and sends $C = G^r$ to $\mathcal{A}$. $\mathcal{C}$ asks him to guess the proxy re-encryption of C to key d as

$$V = g^{\frac{r}{a}d}. \tag{51}$$

Phase 2. $\mathcal{A}$ performs further actions as described above.

Guess. $\mathcal{A}$ returns its guess for V as V' to $\mathcal{C}$. $\mathcal{C}$ computes

$$W = V^{\frac{1}{rd}} \tag{52}$$

to solve the instance of the l-BDHI problem as

$$W = V^{\frac{1}{rd}} = (g^{\frac{r}{a}d})^{\frac{1}{rd}} = g^{\frac{1}{a}}. \tag{53}$$

The probability that this algorithm solves the l-BDHI problem is the same as the advantage of the adversary in the security game. It is $Pr[V = V'] = \epsilon$.

If the l-BDHI assumption holds, this advantage can only be negligible. Therefore, the adversary can only achieve this attack with a negligible advantage. □