Integration of Access Control with Aggregation Query Processing over Outsourced Data

Meghdad Mirabi
Faculty of Computer Science, Technical University of Darmstadt
Darmstadt, Germany
meghdad.mirabi@cs.tu-darmstadt.de

Carsten Binnig
Faculty of Computer Science, Technical University of Darmstadt
Darmstadt, Germany
carsten.binnig@cs.tu-darmstadt.de

ABSTRACT

Despite great advances in cryptography and extensive research in the field of data outsourcing, integrating access control with query processing for secure query evaluation over outsourced data remains an open challenge. In this paper, we propose a fine-grained access control enforcement mechanism tightly integrated with query processing to evaluate aggregation SQL queries over secret-shared data without revealing any information about data, associated access policies, user queries and query results. Our proposed approach can process simple and multi-dimensional SQL queries including aggregation functions "count", "sum", and "avg" over secret-shared data without the need of communication between servers during query execution. It is a privacy-preserving communication-efficient approach and experimental results show that it is computationally efficient for data outsourcing in the case of tuple level policy attachment.

PVLDB Reference Format:
Meghdad Mirabi and Carsten Binnig. Integration of Access Control with Aggregation Query Processing over Outsourced Data. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/.....

1 INTRODUCTION

Cloud computing as a new generation of computing provides a large number of benefits for both companies and individuals willing to store and process their data in a public environment such as high availability, scalability, and efficiency. It offers cheap storage capacity and computing resources and reduces the cost of provisioning of infrastructure and maintenance [40, 41, 44].

In a cloud environment, data owners outsource their data to the cloud and data users can access their intended data irrespective of time and location across multiple platforms (e.g., mobile devices, personal computers, and laptops). However, the major barrier to data outsourcing in a cloud infrastructure is security concerns about the potential disclosure of private data and the leakage of data processing results to other cloud tenants or the service provider itself since data owners do not have direct control over their data and computations [22, 63, 70]. This challenge can be exacerbated when data owners differentiate between data users and define a set of access control policies to restrict access to their data [27, 50]. Basically, an analysis of access control policies may allow observers to clarify the relationships and dependencies between data items and disclose private data. It may even reconstruct the relationships between different data users accessing the outsourced data [16, 33]. In addition, in some cases, there is a system requirement that access control policies to be confidential since data owners do not want to publicly declare to whom they give (or not give) access to their data. Therefore, it is of great significance to prevent an adversary to learn about both the data itself and associated access control policies 1) by just observing cryptographically secure data and associated access control policies and inferring the frequency of each value (i.e., frequent count attack), 2) by just deducting which tuples satisfying (or filtering out from) a user query (i.e., access pattern attack) or counting the number of tuples satisfying (or filtering out from) a query condition (i.e., output size attack) during both query processing and access authorization checking. To prevent these attacks, we need a practical solution to maintain 1) the privacy of user queries, 2) the privacy of data, associated access control policies, and intermediate results during a computation, and 3) the privacy of query results.

A typical solution to protect sensitive data from inappropriate disclosure either by the cloud or external attackers is to encrypt data before being outsourced to the cloud [1, 22, 39, 52, 54, 62]. The general idea is to transfer access to outsourced data into access to secret keys by which data is encrypted before uploading in the cloud. All techniques based on this solution (e.g., Homomorphic Encryption [9, 34, 35, 43, 61], Searchable-Encryption [15, 49, 50, 66], Bucketization [29]) assume that an adversary does not have sufficient computing resources to break the underlying cryptographic mechanisms in a practically short amount of time. These techniques as computationally secure techniques introduce a challenging issue of how to efficiently process different types of queries on encrypted data. Most of these techniques are extremely complex or cannot practically support various types of queries.

An effective technique to provide data confidentiality, data privacy, and a fine-grain access control in cloud computing is Attribute Based Encryption (ABE). In ABE technique [3, 6, 13, 23, 38, 42, 51, 53], a user is authorized to decrypt a ciphertext only if his/her attribute set satisfies the corresponding access control policies. However, the conventional ABE approaches naturally reveal the

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

set of user attributes and access policies to the public and suffer from the inference attack [28, 72].

Recently, several research works have explored information theoretically secure techniques using Multi Party Computation (MPC) which support efficient processing of aggregation SQL queries over outsourced data, while preserving the privacy of outsourced data, user queries, and query results [14, 18, 19, 24]. These techniques make no assumptions about the adversary's computing capabilities and are unconditionally secure. However, the existing proposals [14, 18, 19, 24] assume that data users are fully trusted and can access outsourced data without any limitation. Obviously, it is not a case in the real world where data users have different access privileges and user requests to access unauthorized parts of outsourced data must be filter.

While both computationally and information-theoretically secure techniques have been largely studied for data outsourcing, there is no practical and efficient solution which can integrate access control with query processing to evaluate aggregation SQL queries over outsourced data while preserving the privacy of user queries, outsourced data, associated access control policies, and query results.

Considering the mentioned limitations in the existing works and triggered by its importance, in this paper, we propose a fine-grained access control enforcement mechanism tightly integrated with query processing to evaluate aggregation SQL queries (i.e., "count", "sum", and "avg" queries) over secret-shared data in a privacy preserving manner. In our proposed approach, the data owner splits attribute values of a relation as well as attached access control policies into several secret-shares using Shamir's secret sharing scheme [59] and distributes them to a set of *honest-but-curious* servers. Later, an authorized user, who has submitted an aggregation SQL query, receives the secret-shared query results from a set of servers and performs Lagrange Interpolation operation on the received results to provide the final answer for the query. The key idea to check access authorizations and filter out unauthorized query results is to regard an access authorization as a query condition to be checked during query processing at the server side [37, 45, 46, 54, 55]. In our proposed approach, an aggregation SQL query is first rewritten by every *honest-but-curious* server in such a way that some query conditions are added in the where clause of the query to check the attached access authorizations of a relation in the form of secret-share and then the rewritten query is processed by the corresponding server to provide the authorized query results in the form of secret-share. To specify a set of access control policies with different levels of granularity (i.e., tuple, attribute, and cell), we exploit Attribute-Based Access Control (ABAC) model [31, 64, 69, 71] in this paper. In our proposed ABAC model, a user whose identity attributes satisfy the ABAC policy associated with a specific data item (i.e., a tuple, attribute, or cell) is allowed to perform a particular aggregation function (i.e., "count", "sum", or "avg") on that data item.

In summary, the main contributions of this paper are as follows:

- We propose an ABAC model to specify the set of fine-grained access control policies by adding new attributes to relation to be outsourced in such a way that each value of the new attribute represents a specific ABAC policy. To reduce the number of ABAC policies and support complex access conditions, our proposed ABAC model combines all user attribute conditions together as only one condition set using boolean operators and then automatically maps each condition set to a unique user group. Each user group is later used to specify access control policies on different data items.
- Our proposed approach obliviously rewrites the submitted aggregation SQL query by adding some query conditions in the where clause of the query to check access authorizations and filter out unauthorized data items during query processing at the server side. It does not require any communication between any two servers during query rewriting.
- We design a set of string matching based operators to obliviously process simple and multi-dimensional SQL queries including aggregation functions "count", "sum", and "avg" over secret-shared data. Such operators include bit-wise multiplication followed with an addition over all values of bits of secret-shares to compute the result of a specific aggregation function. There is no need for different servers to communicate with each other during query processing using these string matching based operators.
- We analyze our proposed approach in terms of the number of computation and communication rounds at both server and data user sides. We also experimentally evaluate the performance of our proposed approach by performing several experiments using TPC-H benchmark.

The rest of paper is organized as follows: In Section 2, the preliminary concepts are described. In Section 3, an overview of the system architecture, adversary model, our proposed ABAC model, and policy attachment process are provided. In Section 4, our proposed approach for data outsourcing and oblivious query rewriting and processing is explained. In Section 5, the performance of our proposed approach is experimentally investigated. In Section 6, the existing research works are reviewed and compared with our proposed approach. Finally, this paper is conducted by a conclusion and discussion of future works in Section 7.

2 BACKGROUND

In this section, we briefly review the basic concepts of Shamir's secret sharing scheme and string matching on Shamir's secret sharing scheme. They are as building blocks used in this paper.

2.1 Shamir's Secret Sharing Scheme

Shamir's secret sharing scheme [59] as a threshold secret sharing scheme is secure against adversaries with unlimited computing resources and time. It is known as an information-theoretically secure technique and resists to future developments in computing. The basic idea behind Shamir's secret sharing scheme is that k points are enough to define a k-1 degree polynomial. To share a secret value S among c non-communicating participants/servers, k-1 random coefficients $a_1, a_2, ..., a_{k-1}$ are chosen by the data owner to build a pronominal $f(x) = a_0 + a_1x^1 + a_2x^2 + ... + a_{k-1}x^{k-1}$ where $(k \leq c)$, $f(x) = \mathbb{F}_{\mathbb{P}}[x]$, $\mathbb{P}$ is a prime number, $\mathbb{F}_{\mathbb{P}}$ is a finite field of order $\mathbb{P}$, $a_0 = S$, and $a_i \in \mathbb{N}$ $(\forall 1 \leq i \leq k-1)$. Then, each participant/server i $(\forall 1 \leq i \leq c)$ is given a point $(x_i, f(x_i))$ on the

polynomial. The secret value S can be reconstructed based on any subset of k secret-shares using Lagrange Interpolation operation [2, 7, 11].

2.2 String Matching on Secret Sharing Schemes

Generally, string matching is a basic task in different domains of computer science where a pattern (i.e., string) must be found as the part of text, malware defense, pattern recognition, bio-informatics, and database query. In this scenario, the inputs are a text and a pattern and the pattern is usually much shorter than the text. The goal is to find out whether the pattern appears in the text or not.

Recently, a new string matching method called Accumulating Automata (AA) is proposed by [17] in which the need for cooperation between participants/servers to perform a string matching operation is eliminated. It is a secure multi-party computation method which can be used in our proposed approach to check whether query conditions in the where clause of submitted query can be satisfied or not. Note that the set of query conditions contains 1) conditions of the submitted aggregation SQL query itself and 2) conditions added by the server to check the accessibility of data items during query processing time. Here we describe how to exploit AA for performing string matching over secret-shares.

Assume that S is the value of secret and S_i ($\forall 1 \leq i \leq c$) is the ith secret-shares of S stored at the ith server. AA allows a data user to search a string pattern p by creating c secret-shares of p (i.e., $p_i, \forall 1 \leq i \leq c$) in such a way that the ith server can search the secret-share pattern p_i over the secret-share S_i. The result can be 1 in the form of secret-share if the secret-share pattern p_i matches with secret-share S_i or 0 in the form of secret-share; otherwise. To find a pattern on servers, AA applies the multiplication of secret-shares and the addition of final secret-shares. After receiving the the outputs from k servers, the data user performs Lagrange Interpolation operation to reconstruct the secret value. The process of string matching using AA in shown in Example 2.1.

Example 2.1. Consider the values of *Account Type* attribute in relation *Account* in Table 1 which are "checking" and "saving". These two values can be mapped to "01" and "10" in the unary representation form, respectively since we only have these two values for *Account Type* attribute.

Table 1: Account Relation

Account No.	Account Type	Balance (×1000$)
1	checking	2
2	saving	3
3	checking	1

Now, assume that "01" as the unary representation of value "checking" is outsourced by the data owner. Therefore, it will reveal the value "checking" to the adversary. To prevent data disclosure, two polynomials with an identical degree can be used by the data owner to outsource the value "checking" to the set of *5* non-communicating servers as shown in Table 2.

Now, assume that the data user wants to search for the value "checking". The data user knows that the value "checking" is represented as "01". Then, he/she creates secret-shares for that as shown in Table 3. It should be noted here that the data user does not need to ask from the data owner about any polynomial to build the secret-shares of value "checking".

At the server side, every individual server performs position-wise multiplication of the bits that they have, adds all the multiplication resultants, and sends them to the data owner. This process is shown in Table 4.

After receiving the outputs from the set of *5* non-communicating servers which are $y_1 = 18$, $y_2 = 57$, $y_3 = 118$, $y_4 = 201$, and $y_5 = 306$, the data user performs Lagrange Interpolation operation to reconstruct the secret answer which is 1 (i.e., $b_0 = 1$) to confirm that the string pattern is found. This process is as follows:

$$\frac{(x-x_2)(x-x_3)(x-x_4)(x-x_5)}{(x_1-x_2)(x_1-x_3)(x_1-x_4)(x_1-x_5)} \times y_1 + \frac{(x-x_1)(x-x_3)(x-x_4)(x-x_5)}{(x_2-x_1)(x_2-x_3)(x_2-x_4)(x_2-x_5)} \times y_2 +$$
$$\frac{(x-x_1)(x-x_2)(x-x_4)(x-x_5)}{(x_3-x_1)(x_3-x_2)(x_3-x_4)(x_3-x_5)} \times y_3 + \frac{(x-x_1)(x-x_2)(x-x_3)(x-x_5)}{(x_4-x_1)(x_4-x_2)(x_4-x_3)(x_4-x_5)} \times y_4 +$$
$$\frac{(x-x_1)(x-x_2)(x-x_3)(x-x_4)}{(x_5-x_1)(x_5-x_2)(x_5-x_3)(x_5-x_4)} \times y_5 = \frac{(x-2)(x-3)(x-4)(x-5)}{(1-2)(1-3)(1-4)(1-5)} \times 18 +$$
$$\frac{(x-1)(x-3)(x-4)(x-5)}{(2-1)(2-3)(2-4)(2-5)} \times 57 + \frac{(x-1)(x-2)(x-4)(x-5)}{(3-1)(3-2)(3-4)(3-5)} \times 118 +$$
$$\frac{(x-1)(x-2)(x-3)(x-5)}{(4-1)(4-2)(4-3)(4-5)} \times 201 + \frac{(x-1)(x-2)(x-3)(x-4)}{(5-1)(5-2)(5-3)(5-4)} \times 306 =$$
$$b_0 + b_1x^1 + b_2x^2 + b_3x^3 + b_4x^4, b_0 = 1$$

3 OVERVIEW

In this section, we provide an overview of the system architecture, adversarial model, our proposed ABAC model, and policy attachment process used in this paper.

3.1 System Architecture

We assume three entities in our proposed system architecture as follows: *Data Owner*, *Data User*, and *Non-Communicating Servers*. The interaction between these entities are shown in Figure 1.

In step 1, the data user registers his identity attributes with the data owner. In Step 2, the data owner creates a credential for the data user based on his identity attributes and sends it to the data user. The information about users' credentials is stored in relation *userInfo* at the data owner side and the authentication process of data users is done by their credentials at the server side. In Step 3, the data owner who owns relation R (i.e., including data and associated access control policies) creates c relations R_1, R_2, ..., and

Table 2: Secret-Shares of Value "checking" Created by the Data Owner

Value	Polynomial	S1 (x=1)	S2 (x=2)	S3 (x=3)	S4 (x=4)	S5 (x=5)
0	0+x	1	2	3	4	5
1	1+2x	3	5	7	9	11

Table 3: Secret-Shares of Value "checking" Created by the Data User

Value	Polynomial	S1(x=1)	S2(x=2)	S3(x=3)	S4(x=4)	S5(x=5)
0	0+3x	3	6	9	12	15
1	1+4x	5	9	13	17	21

Table 4: Operations Performed by Non-Communicating Servers

Server 1	Server 2	Server 3	Server 4	Server 5
$1 \times 3 = 3$	$2 \times 6 = 12$	$3 \times 9 = 27$	$4 \times 12 = 48$	$5 \times 15 = 75$
$3 \times 5 = 15$	$5 \times 9 = 45$	$7 \times 13 = 91$	$9 \times 17 = 153$	$11 \times 21 = 231$
$3 + 15 = 18$	$12 + 45 = 57$	$27 + 91 = 118$	$48 + 153 = 201$	$75 + 231 = 306$

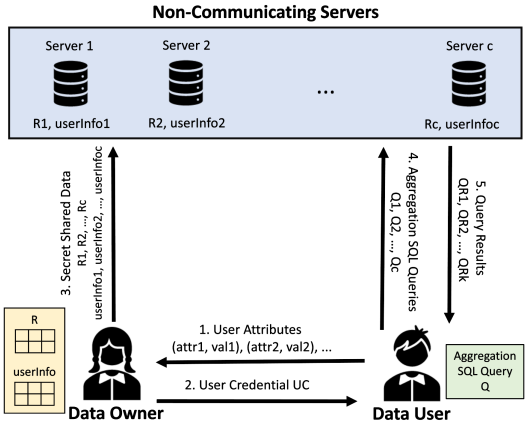


Figure 1: System Architecture

R_c for relation R using Shamir's secret sharing scheme and sends the ith relation to the ith server. She also creates c relations $userInfo_1$, $userInfo_2$, ..., and $userInfo_c$ for relation *userInfo* and sends the ith *userInfo* relation to the ith server. Every individual server S_i needs the information stored in relation $userInfo_i$ during the process of query rewriting. This process will be explained later in Section 4.3. In step 4, the data user who wishes to execute the aggregation SQL query Q on relation R, creates c queries Q_1, Q_2, ..., and Q_c for the query Q by converting each value of query condition to a set of secret-shared values and then sends the ith query including the ith set of secret-shared values to the ith sever. Next, every individual server S_i rewrites and executes the query Q_i. The process of query rewriting in server S_i is done by adding new query conditions in the where clause of the query Q_i. Such query conditions are used to check the set of access authorizations during query processing time. In step 5, the partial outputs from the execution of aggregation SQL queries Q_1, Q_2, ..., and Q_k (i.e., $k < c$) are sent to the data user. Next, the data user performs a simple operation (i.e., Lagrange Interpolation) to obtain the result of query Q. It should be noted that every individual server cannot learn neither the exact values of query conditions in the where clause of query Q nor the query results during query rewriting and query processing times.

3.2 Adversary Model

We consider the *honest-but-curious* adversary model in this paper. In such a model, non-communicating servers correctly compute the assigned tasks (i.e., query rewriting and query processing) and do not attempt to attack the system by modifying data and associated access policies or altering query results. However, every individual server may use side information (e.g., background information, query execution, and output size) to obtain any useful information about outsourced data, associated access control policies, query conditions, and query results.

We assume that the data owner is fully trusted and the adversary cannot lunch any attack against the data owner. We also assume that the adversary cannot access the secret-sharing algorithm and its related information at the data owner side.

In such a model, only authenticated users can request aggregation SQL queries over outsourced data. They wish to learn about unauthorized parts of outsourced data and confidential access control policies. They also try to get any information about the parts of their query results which are filtered out. We assume that authenticated users establish a secure and trust communication with the data owner/non-communicating servers and they use their credentials issued by the data owner for authentication at the server side.

It should be noted here that we follow the restriction of the secret sharing schemes that an adversary cannot collude with the majority of servers or access the communication channel between the data owner/data user and every individual server.

3.3 Attribute Based Access Control Model

In Attribute Based Access Control (ABAC) model, request of a subject to perform an action on an object is granted or denied based on a set of the assigned attributes of subjects and objects [21, 58, 65, 73]. Based on the general basis in this model, we define the following elements in our proposed ABAC model:

- ***Users* (*U*)**. It contains the set of users who are interested in submitting aggregation SQL queries over outsourced data.
- ***Data Items* (*DI*)**. It contains all data items protected by the system. It can be a tuple, attribute, or cell in a relation.
- ***Actions* (*A*)**. It contains the set of aggregation functions provided by the system. They are "count", "sum", and "avg".
- ***Policy* (*P*)**. It contains the set of all policies to access data items in the system.

In such a model, users have a set of identity attributes which explains the properties of users such as "name", "affiliation", "office number", "job title", "role", "trust level", and so on and ABAC policies are associated with data items that specify access conditions in terms of identity attributes of users. A user whose identity attributes satisfy the ABAC policy associated with a data item is allowed to perform a specific action (i.e., an aggregation function) on that data item. In the following, we explain about our proposed ABAC model in more details.

Definition 3.1 (User Attribute (UA)). A user attribute is defined as a pair of ($aName$, $aType$) such that $\forall a \in UA : a = (aName, aType)$, where $aName$ is a unique attribute name and $aType$ is a predefined data type in the system.

Based on Definition 3.1, a user attribute must have a predefined data type (e.g., integer, floating point, boolean, etc.) to enforce consistency in access control policies. It also prevents from any possible ambiguity in access control policies such as type mismatching.

Example 3.2. Three user attributes (*Role*, *String*), (*Age*, *Integer*), (*Sex*, *Boolean*) can be defined in the system based on Definition 3.1.

Definition 3.3 (User Attribute Condition). A user attribute condition is defined in the form of $cond$: $attrName\ compOp\ attrVal$, where $cond$ is a unique condition name, $attrName$ is the name of an identity attribute of users, $compOp$ is a comparison operator such as "=", ">=", "<=", ">", "<", and "!=", and $attrVal$ is a value from the set of values in the predefined datatype $aType$ that can be used by the identity attribute.

Example 3.4. Three user attribute conditions *C1*, *C2*, and *C3* can be defined in the system based on Definition 3.3 as follows: *C1: Role = "Banker"*, *C2: Age > 18*, and *C3: Sex = 0*.

Definition 3.5 (User Attribute Condition Set). A user attribute condition set is defined in the form of $condSet$: $cond$ | ($cond$ $boolOp$ $condSet$) | ($condSet$ $boolOp$ $cond$) | ($condSet$ $boolOp$ $condSet$), where $condSet$ is a unique name for the condition set, $cond$ is a unique condition name, $boolOp$ is a boolean operator which can be "∧" or "∨", and "|" is a discriminator.

Example 3.6. User attribute condition set *CS1: (C1 ∧ (C2 ∨ C3))* can be defined in the system based on Definition 3.5.

Based on Definition 3.5, user attribute conditions specified by the data owner can be combined together as a set of conditions to specify complex conditions. This property in the proposed model supports the modeling of complex real-world situations.

Definition 3.7 (User Group). A user group is defined as a set of users which satisfies user attribute condition set in an ABAC policy.

In our proposed ABAC model, each unique user attribute condition set is mapped to a unique user group based on Definition 3.8.

Definition 3.8 (Mapping of a User Attribute Condition Set to a User Group). Given a set of user attribute condition set $UACS = \{condSet_1, condSet_2, ..., condSet_t\}$, a unique user group G_i can be automatically created by the system for a user attribute condition set $condSet_i$, where $\forall 1 \leq i \leq t$.

Example 3.9. User attribute condition set *CS1* can be automatically mapped to user group G_1 based on Definition 3.8.

Definition 3.10 (ABAC Policy). An ABAC policy is defined as a triple ($userGroup$, $dataItem$, $aggrFunc$), where $userGroup$ is a user group, each member of which has a set of identity attributes which satisfy a specific user attribute condition set, $dataItem$ is a data item in relation R, and $aggrFunc$ is an aggregation SQL function supported by the system.

Example 3.11. An ABAC policy can be defined as follows: (G_1, *Balance*, *sum*) in which a user who is a member of user group G_1 can perform the aggregation function *sum* on the data item *Balance* in relation *Account*.

Based on Definition 3.10, our proposed ABAC model only supports positive access authorizations. Therefore, we assume that access to a specific data item is denied by default.

Ideally, some tools may be needed to verify the correctness of ABAC policies in the system and detect conflicts between different ABAC policies. However, we assume that the set of ABAC policies is checked by several tools to make sure that there are no collisions between them. Therefore, we only focus on how to enforce the set of ABAC policies defined by the data owner to access outsourced data in a privacy-preserving manner.

3.4 The Process of Policy Attachment

The basic idea to attach the set of ABAC policies to relation R (which should be outsourced) is to add new attributes to relation R in such a way that each value of the new attribute represents a specific ABAC policy specified by the data owner to access a data item. In the case of tuple level access control, policy attachment is performed by adding x additional attributes to relation R, where x is the number of aggregation functions supported by the system. In the case of attribute/cell level access control, policy attachment is performed by adding x attributes for each attribute in relation R. It should be noted that each value of the new attribute in relation R as an ABAC policy is labeled with only one user group since the set of user attribute conditions is mapped to a unique user group based on Definition 3.8.

To support the case that no ABAC policy is specified for a specific data item in relation R, we use a use group called *Min User Group* which does not map to any user attribute condition set in the system.

Definition 3.12 (Min User Group). User group G_0 is defined as $Min\ User\ Group$ if it is not mapped to any of the existing user attribute condition sets in the system.

Based on Definition 3.12, it is possible to set the values of new attribute in relation R to G_0 in the case that there is no ABAC policy for that data item in the system (i.e., access denied for all data users).

Example 3.13. Table 5 and 6 show the result of the policy attachment process for relation *Account* (i.e., Table 1) where access control granularity levels are tuple and attribute/cell, respectively.

Table 5: Tuple Level Policy Attachment

Account No.	Account Type	Balance (×1000$)	Count	Sum
1	checking	2	G_1	G_0
2	saving	3	G_2	G_0
3	checking	1	G_1	G_2

As shown in Table 5, all users of user group G_1 can access the first tuple of relation R to perform the aggregation function "count". However, none of users in the system cannot access the first tuple of relation R to perform the aggregation function "sum" since the value of attribute *Sum* for the first tuple is set to G_0 (i.e., $Min\ User\ Group$). In the case of attribute/cell level access control as shown in Table 6, all users of user group G_1 can access the first and third tuples of relation R to perform the aggregation functions "count" and "sum" on the data item *Balance* since the value of attributes *Balance-Count* and *Balance-Sum* for these tuples are set to G_1. As shown in Table 6, the value of attribute *Account Type-Sum* in all tuples of relation *Account* is set to G_0 because the aggregation function "sum" cannot be defined for this attribute in the system.

It is noteworthy that although we can also consider the existence of access control policies for the aggregation function "avg", but we ignore to have ABAC policies for this aggregation function in our proposed ABAC model since the result of this aggregation function can be obtained by dividing the result of aggregation function "sum" to the result of aggregation function "count". If there are ABAC policies for the aggregation function "avg", we may encounter a collision problem among different access control policies in the system. Thus, when a data user submits a request to perform the aggregation function "avg", the corresponding ABAC policies specified for the aggregation functions "sum" and "count" are checked instead of considering the explicit ABAC policy for the aggregation function "avg".

Table 6: Attribute or Cell Level Policy Attachment

Account No.	Account No.-Count	Account No.-Sum	Account Type	Account Type-Count	Account Type-Sum	Balance (×1000$)	Balance-Count	Balance-Sum
1	G1	G0	checking	G1	G0	2	G1	G1
2	G1	G0	saving	G2	G0	3	G2	G2
3	G1	G0	checking	G1	G0	1	G1	G1

4 OUR PROPOSED APPROACH

In this section, we provide a detailed description of our proposed approach including the following steps: 1) Creation and distribution of secret-shares by the data owner, 2) Query submission and distribution by the data user, 3) Query rewriting and processing by non-communicating servers, and 4) Query result collection by the data user. We also consider the complexity of our proposed approach in terms of number of communication and computation rounds at both server and data user sides.

4.1 Creation and Distribution of Secret-Shares

To create a set of secret-shares for relation R (i.e., including data and associated access control policies), we need to represent each value of relation R in a unary form as explained in Section 2.2. Example 4.1 shows how to represent different numerical values in a unary form.

Example 4.1. Assume that a relation contains only numerical values. Generally, a numerical value can be represented by a unary array with 10 bits since we have only 10 numbers from '0' to '9' in decimal form. Hence, the number '1' can be represented as $(1_1, 0_2, 0_3, ..., 0_{10})$, where the subscript indicates the position of the numerical value; since '1' is the first number, the first bit in the unary array is one and others are zero. Similarly, '2' is $(0_1, 1_2, 0_3, ..., 0_{10})$, ..., '9' is $(0_1, 0_2, ..., 0_8, 1_9, 0_{10})$, and '0' is $(0_1, 0_2, ..., 0_9, 1_{10})$.

This process can be followed in a similar way to represent other symbols. However, in AA, if the data user wants to search a string pattern with the length b bits without the need of cooperation between servers in *one communication round* where the data owner and data user both use a polynomial with the degree *one*, the final degree of polynomial will be *2b* due to the need to multiply secret-shares in the process of string matching and therefore, solving such a polynomial will require *2b+1* secret-shares from different servers. By mapping each value of relation R into a unary representation form with fewer number of bits, such a process would not only be more efficient, but also require fewer secret-shares (non-communicating servers). However, this kind of mapping must be pre-agreed between the data owner and the data users (i.e., similar to the marshaling process for message passing between different entities in distributed systems). Example 4.2 shows how such a process can be performed for relation *Account*.

Example 4.2. Table 7 shows the output of unary representation of values in relation *Account* including the tuple level policy attachment (Table 5). We only need to use 3 bits to represent the values of attributes *Account-No.*, *Balance*, *Count*, and *Sum* in the unary representation form since each of them has only three different values. Moreover, we only need to use 2 bits to represent the values of attribute *Account Type* in the unary representation form since it has two different values.

If the data owner outsources the values of a relation in the unary representation form, it will reveal the data of relation. Thus, the data owner uses any b polynomials with an identical degree to create b secret-shares for a particular value, where b is the number of bits in the unary representation form of that value and then these b secret-shares will be distributed to a specific server instead of sending the actual value in the unary representation form. Example 2.1 in Section 2.2 shows how such a process is performed for the value of "checking" in attribute *Account Type* of relation *Account*.

It should be noted that different polynomials are used to create secret-shares for each occurrence of a specific value in relation R and therefore, multiple occurrences of a value have different secret-shares. In this case, an adversary cannot learn about the number of occurrences of a specific value in relation R (i.e., Protection against frequent count attacks).

Assume that R with n tuples and m attributes denoted by A_1, A_2, ..., and A_m is a relation which should be outsourced. In the case of tuple level access control, two new attributes *Count* and *Sum* are added to relation R whose values specify the ABAC policy attached to each tuple of relation R. Therefore, relation R^t (t stands for tuple level access control) with n tuples and $m+2$ attributes denoted by $A_1, A_2, ..., A_m, Count$, and Sum should be outsourced to a set of non-communicating servers. In the case of attribute/cell level access control, two new attributes are added for each attribute in relation R whose values specify the ABAC policies associated with each attribute in relation R. Therefore, relation R^a (a stands for attribute level access control) with n tuples and $(3 \times m)$ attributes denoted by $A_1, A_1^{Count}, A_1^{Sum}, A_2, A_2^{Count}, A_2^{Sum}, ..., A_m, A_m^{Count}$, and A_m^{Sum} should be outsourced to a set of non-communicating servers. Now, assume that v_{ij} is the value of the ith tuple and jth attribute in relations R^t and R^a, and c is the number of non-communicating servers. Therefore, the data owner creates c secret-shares for the value v_{ij} (i.e., $S(v_{ij})$). The result of this step is c secret-shared relations (i.e., R_1^t, R_2^t, ..., and R_c^t in the case of tuple level access control and R_1^a, R_2^a, ..., and R_c^a in the case of attribute/cell level access control). Then, the pth secret-shared relation (i.e., R_p^t in the tuple level access control and R_p^a in the attribute/cell level access control) is outsourced to the pth server.

To rewrite the aggregation SQL query submitted by the data user u at the server side, all servers need the list of user groups of which the data user u is a member of to add new query conditions as access conditions in the where clause of the submitted query. To

Table 7: Unary Representation of Account Relation

Account No.	Account Type	Balance (×1000$)	Count	Sum
100	01	010	100	001
010	10	001	010	001
001	01	100	100	010

provide this information, the data owner creates relation *userInfo* = (*Credential*, *User Group*) to store the values of credentials and user groups for each registered data user in the system. The values of attributes *Credential* and *User Group* are inserted into relation *userInfo* during the user registration process when a specific credential is created for the data user u and all the ABAC policies specified by the data owner are considered to find the list of user groups of which the data user u with a set of identity attributes is member of. However, outsourcing the values of attribute *User Group* in relation *userInfo* violates the privacy of data users and reveals relationships between different data users in the system. It is noted that servers need the exact values of attribute *Credential* in the process of user's authentication. To preserve the privacy of data users and prevent the leakage of access control policies, the data owner creates c secret-shares for each value of attribute *User Group* and then creates c relations $userInfo_1$, $userInfo_2$, ..., and $userInfo_c$ for relation *userInfo* in such a way that the values of attribute *Credential* are unchanged and the values of attribute *User Group* are replaced by their corresponding secret-shares. Finally, the pth relation of *userInfo* (i.e., $userInfo_p$) is send to the pth server. To hide the actual number of user groups which a user is member of, the data owner can generate a set of user groups that has never been used in the system and randomly assign the registered data users to these user groups (i.e., Elimination of inference channels).

Example 4.3. Table 8 (a) shows an example of *userInfo* relation which should be outsourced among different non-communicating servers. As shown in Table 8 (a), a user having a specific credential can be member of different user groups in our proposed ABAC model. Table 8 (b) shows the values of attributes *Credential* and *User Group* which should be outsourced to the pth server, where $S(G_x)_p$ indicates the secret-share of the xth user group (i.e., G_x) in relation $userInfo_p$. However, this table can also include fake user groups to protect the system against the inference of user groups and access policies.

Table 8: Relations $userInfo$ and $userInfo_p$

Credential	User Group
c_1	G_1
c_1	G_3
c_2	G_2
c_3	G_1
c_3	G_2

(a) $userInfo$ Relation

Credential	User Group
c_1	$S(G_1)_p$
c_1	$S(G_3)_p$
c_2	$S(G_2)_p$
c_3	$S(G_1)_p$
c_3	$S(G_2)_p$

(b) $userInfo_p$ Relation

4.2 Query Submission and Distribution

Our proposed approach supports both the simple and multi dimensional aggregation SQL queries as shown in Table 9. In the following, we explain how a data user can submit and distribute an aggregation SQL query to the set of non-communicating servers.

(1) Simple aggregation SQL queries: Assume that the data user u wishes to submit the aggregation SQL query Q_1 in the form of "select $\alpha(A_i)$ from R" to servers. This query will be distributed to each individual server without any changes. The important point here is that the query patterns can be hidden from non-communicating servers in our proposed approach when the submitted aggregation SQL query has at least one query condition. Since there are no query conditions in the query Q_1, we should not worry about query patterns being obvious.

(2) Multi-dimensional aggregation SQL queries: In the case that the data user u wishes to submit the aggregation SQL query Q_2 in the form of "select $\alpha(A_i)$ from R where $(A_k = v_k)$ OP ... OP $(A_l = v_l)$", the actual values in query conditions (e.g., v_k and v_l) should be represented in the unary form and then a set of c secret-shares should be created for each bit of them by the data user u, where c is the number of non-communicating servers. Such a process is explained Example 2.1 in Section 2.2. Assume that the pth set of secret-shares for all bits of values of v_k and v_l are $S(v_k)_p$ and $S(v_k)_p$, respectively. Then, these secret-shares are replaced by the actual values in query conditions to hide the query pattern from every individual server. Thus, the query Q_{2p} in the form of "Select $\alpha(A_i)$ from R where $(A_k = S(v_k)_p)$ OP ... OP $(A_l = S(v_l)_p)$" will be distributed to the pth server.

(3) Multi-dimensional aggregation SQL queries including Group By clause (with or without Having): This process will be performed similar to multi-dimensional aggregation SQL queries.

4.3 Query Rewriting and Processing

This section provides the details on query rewriting and query processing in our proposed approach along with a discussion about non-disclosure of data at the server side. The remarkable point here is that every individual server does not need to communicate with other entities (i.e., the data owner, the data user, or other servers) during query rewriting and query processing.

4.3.1 Query Rewriting. When an aggregation SQL query is submitted by the data user u on relation R, it is necessary to check the set of ABAC policies attached to relation R to restrict the query result to the only data items which the data user u has access to. In the following, we explain in detail how to obliviously rewrite an aggregation SQL query at the server side.

(1) Simple aggregation SQL queries: Assume that the aggregation query Q_p = "select $\alpha(A_i)$ from R" is sent from the data user u to the pth server and F_p is relation R_p^t in the case of tuple level access control and relation R_p^a in the case of attribute/cell level access control outsourced on the pth server. This query is rewritten as follows: $Q_p^{'}$ = "select $\alpha(A_i)$ from F_p where $(\beta = S(UG_1)_p) \vee (\beta = S(UG_2)_p) \vee ... \vee (\beta = S(UG_q)_p)$", where β is the attribute α in relation R_p^t or attribute A_i^{α} in relation R_p^a, and $S(UG_x)_p$ is the secret-share of the xth user group UG ($\forall 1 \leq x \leq q$) in relation $userInfo_p$ which the data user u is a member of. In the process of query rewriting, we need to add a query condition in the where clause of the query $Q_p^{'}$ for each user group which the data user u is member of since the data user u can be a member of different user groups and the set of ABAC policies are mapped into user groups in the system.

Table 9: Types of Supported Aggregation Queries in Our Proposed Approach

Aggregation SQL Query Type	Query Format
Simple Aggregation SQL Queries	select $\alpha(A_i)$ from R
Multi-Dimensional Aggregation SQL Queries	select $\alpha(A_i)$ from R where $(A_k{=}v_k)$ OP ... OP $(A_l{=}v_l)$
Multi-Dimensional Aggregation SQL Queries with Group-By	select A_i, $\alpha(A_j)$ from R where $(A_k{=}v_k)$ OP ... OP $(A_l{=}v_l)$ Group By A_i
Multi-Dimensional Aggregation SQL Queries with Group-By and Having	select A_i, $\alpha(A_j)$ from R where $(A_k{=}v_k)$ OP ... OP $(A_l{=}v_l)$ Group By A_i Having $(A_i{=}v_i)$

Note: α can be "count", "sum", or "avg" and OP can be "$\wedge$" or "$\vee$" operator.

(2) Multi-dimensional aggregation SQL queries: Assume that the aggregation query Q_p = "select $\alpha(A_i)$ from R where $(A_k = S(v_k)_p)$ OP ... OP $(A_l = S(v_l)_p)$" is sent from the data user u to the pth server and F_p is relation R_p^t in the case of tuple level access control and relation R_p^a in the case of attribute/cell level access control outsourced on the pth server. This query is rewritten as follows: $Q_p^{'}$ = "select $\alpha(A_i)$ from F_p where $(A_k = S(v_k)_p)$ OP ... OP $(A_l = S(v_l)_p) \wedge ((\beta = S(UG_1)_p) \vee (\beta = S(UG_2)_p) \vee ... \vee (\beta = S(UG_q)_p))$", where β is the attribute α in relation R_p^t or attribute A_i^{α} in relation R_p^a, and $S(UG_x)_p$ is the secret-share of the xth user group UG $(\forall 1 \leq x \leq q)$ in relation $userInfo_p$ which the data user u is a member of. In this process, a set of query conditions is added in the where clause of the query $Q_p^{'}$ using "$\wedge$" operator. These query conditions are specified for the user groups which the data user u is member of and each two query conditions are combined together using "$\vee$" operator.

(3) Multi-dimensional aggregation SQL queries including Group By clause (with or without Having): This process is similar to the process of query rewriting for multi-dimensional aggregation SQL queries explained before.

Example 4.4. Assume that the data user u_1 with the credential c_1 wishes to submit the query Q in the form of "Select *count*(*Balance*) from *Account* where (*Account Type* = "checking")" on relation *Account* (in Table 5). Such a query is distributed to the pth server in the form of "Select *count*(*Balance*) from *Account* where (*Account Type* = $S(checking)_p$)". Next, this query is rewritten by the pth server as follows: "Select *count(Balance)* from *Account* where (*Account Type* = $S(checking)_p) \wedge ((Count = S(G_1)_p) \vee (Count = S(G_3)_p))$ since the data user u_1 with the credential c_1 is a member of user groups G_1 and G_3 (refer to Table 8).

4.3.2 Query Processing. In this phase, the pth server $(\forall 1 \leq p \leq c)$ executes the pth aggregation query $Q_p^{'}$ over the pth outsourced relation F_p (i.e., relation R_p^t in the case of tuple level access control and relation R_p^a in the case of attribute/cell level access control) and filters the data items that do not satisfy the query conditions in the query $Q_p^{'}$. To find the result of string matching (i.e., which can be "0" or "1" in the form of secret-share) for each query condition in the query $Q_p^{'}$, the operator $\odot$ as the string matching operator is used by the pth server including a bit-wise multiplication followed with an addition over all values of bits of secret-shares in the unary representation form. It is defined as follows:

$$Result_y^z = \begin{cases} S(v(\alpha)_z)_p \odot S(v_y)_p & \text{if } \beta = \alpha \\ S(v(A_y^{\alpha})_z)_p \odot S(v_y)_p & \text{if } \beta = A_i^{\alpha} \\ S(v(A_y)_z)_p \odot S(v_y)_p & \text{otherwise} \end{cases}$$

where $S(v(\alpha)_z)_p$ is the secret-share of attribute α in the zth tuple of relation F_p (or R_p^t), $S(v(A_y^{\alpha})_z)_p$ is the secret-share of attribute A_y^{α} in the zth tuple of relation F_p (or R_p^a), $S(v(A_y)_z)_p$ is the secret-share of attribute A_y in the zth tuple of relation F_p (i.e., R_p^t or R_p^a), and $S(v_y)_p$ is the secret-share of the corresponding query condition in the query $Q_p^{'}$. It should be noted that α is the aggregation function in the query $Q_p^{'}$. Table 4 in Example 2.1 shows how this operator can be used by every individual server for obliviously searching the string pattern "Checking" over outsourced data.

The results of string matching can be used to compute the result of a specific aggregation function in the form of secret-share at the pth server. Such a process varies depending on the type of aggregation functions. In the following, we explain in detail how to exploit the result of string matching to process aggregation functions.

(1) Count Function: The result of an aggregation SQL query including "count" function can be computed by the pth server using the following operation:

$$output = \sum_{z=1}^{n} (((Result_{A_k}^z \circledast ...) \circledast Result_{A_l}^z)) \wedge$$

$$(((Result_{\beta_1}^z \vee Result_{\beta_2}^z) \vee ... \vee Result_{\beta_q}^z))$$

where $\circledast$ is OP (i.e, $\wedge$ or $\vee$), A_k, ..., A_l are the set of attributes in the where clause of query Q_p (and $Q_p^{'}$), and β_i is the ith β in the where clause of query $Q_p^{'}$ $(\forall 1 \leq i \leq q)$. To capture the operation $\wedge$ and compute the final result of $(Result_i^z \wedge Result_j^z)$ in the form of secret-share for each tuple z, the pth server executes the following computation:

$$Result_{ij}^z = Result_i^z \times Result_j^z$$

To capture the operation $\vee$ and compute the final result of $(Result_i^z \vee Result_j^z)$ in the form of secret-share for each tuple z, the pth server executes the following computation:

$$Result_{ij}^z = Result_i^z + Result_j^z - Result_i^z \times Result_j^z$$

The correctness of the output of "count" function can be described as if the zth tuple has "0" in the form of secret-share as a comparison resultant of $(((Result_{A_k}^z \circledast ...) \circledast Result_{A_l}^z))$ or $(((Result_{\beta_1}^z \vee Result_{\beta_2}^z) \vee ... \vee Result_{\beta_q}^z))$, it will produce "0" in the secret-share form as the result of this operation for the zth tuple; therefore, the zth tuple will not counted as the result of this operation. Thus, the correct occurrences over all tuples that satisfy the query's where clause are counted as the result of this operation.

(2) Sum Function: The result of an aggregation SQL query including "sum" function can be computed by the pth server using the following operation:

$$output = \sum_{z=1}^{n} S(v(A_i)_z)_p \times (((Result^z_{A_k} \circledast ...) \circledast Result^z_{A_l}) \wedge (((Result^z_{\beta_1} \vee Result^z_{\beta_2}) \vee ... \vee Result^z_{\beta_q})))$$

where A_i is the attribute which the aggregation function "sum" is applied on, $S(v(A_i)_z)_p$ is the secret-share of the attribute A_i of zth tuple of relation F_p, $\circledast$ is OP (i.e, $\wedge$ or $\vee$), A_k, ..., A_l are the set of attributes in the where clause of query Q_p (and $Q_p^{'}$), and β_i is the ith β in the where clause of query $Q_p^{'}$ ($\forall 1 \leq i \leq q$). The process of computation to find the final results of $(Result_i^z \vee Result_j^z)$ and $(Result_i^z \wedge Result_j^z)$ for each tuple z is similar to this process in "count" function. In addition, the argument for the correctness of "sum" operation is similar to the correctness of the operation "count".

(3) Avg Function: An aggregation SQL query including "avg" function can be processed by dividing the results of corresponding aggregation SQL query including "sum" function to the corresponding aggregation SQL query including "count" function. In this case, every individual server sends the query results for these two queries to the data user u. Upon arrival of the query results, the data user u utilizes them to compute the result of aggregation SQL query including "avg" function.

Example 4.5. Assume that the query $Q_p^{'}$ in the form of "Select *count(Balance)* from *Account* where *(Account Type* = $S(checking)_p) \wedge (Count = S(G_1)_p \vee Count = S(G_3)_p)$ is the output of the query rewriting phase by the pth server (Refer to Example 4.4). Table 10 shows how do the pth server executes this query using string matching based operations. The results of string matching are used to compute the result of the count query $Q_p^{'}$ at the pth server using the following operation:

$$output = \sum_{z=1}^{3} (Result^z_{AccountType} \wedge (Result^z_{Count1} \vee Result^z_{Count2})) =$$
$$(S(1) \wedge (S(1) \vee S(0))) + (S(0) \wedge (S(0) \vee S(0))) + (S(1) \wedge (S(1) \vee S(0))) =$$
$$S(1) + S(0) + S(1) = S(2).$$

Note that this process will be performed on the unary representation form of secret-shares. However, we show cleartext values for simple explanation here.

A Group-By query in combination with an aggregation function in the form of "select A_i, $\alpha(A_j)$ from R where $(A_k{=}v_k)$ OP ... OP $(A_l{=}v_l)$ Group By A_i" can be processed similar to other multi-dimensional aggregation SQL queries. The only requirement here is that the set of possible values for attribute A_i should be known for the data user u in advance. In this case, the data user u can request the pth server to execute the aggregation function α on each tuple of relation R for each group of attribute A_i (1 to g) and returns $<S(v_f)_p, S(output_f)_p>$, where $1 \leq f \leq g$, $S(v_f)_p$ is the name of the fth group of attribute A_i in the form of secret share in the pth server, and $S(output_f)_p$ is the answer to the Group-By query for the fth group in the form of secret-share in the pth server. Upon arrival of the secret-shared query results for each group in attribute A_i, the data user u exploits them to obtain the final answer for that group.

4.3.3 Discussion about Data Leakage. Generally, access patterns are hidden from adversaries (i.e., every *honest-but-curious* server and external attacker) because 1) data and associated access control policies stored on servers, 2) query conditions distributed by the data user, and 3) new query conditions added to check access authorization, are all in the form of secret-share. Also, the actual values of query results and their sizes are masked from adversaries since the output of an aggregation SQL query is in the form of secret-share and contains an identical number of bits. However, an adversary knows the exact type of submitted query (i.e., simple or multi-dimensional queries with/without Group-By and Having) and the exact type of aggregation function used in the query (i.e., "count", "sum", and avg).

4.4 Query Result Collection

After receiving the set of results from servers, the data user performs Lagrange Interpolation operation on the received results to provide the final answer for the query Q. Such a process is explained Example 2.1 in Section 2.2. For multi-dimensional aggregation SQL queries including Group-By clause (with or without Having), the data user u may learn about the number of groups in an attribute, but the data owner can create a number of fake groups for that attribute and share them with the data user along with the actual groups of that attribute. Therefore, the data user u cannot distinguish between actual and fake groups. It should be noted that an unauthorized user cannot infer any information from the result of an aggregation SQL query including Group-By clause when the query result is 0 (for a specific group of the attribute included in the Group-By clause) since it means either the data user does not have access to that group to perform an aggregation function, or the query result for that group is 0 by default.

4.5 Complexity of Our Proposed Approach

Table 11 shows the complexity of our proposed approach to process different types of aggregation SQL queries. As shown in Table 11, it requires only one communication round between the data user and every individual server and one computation round to scan the tuples during the process of querying at the server side for all types of aggregation SQL queries. To interpolate the query results at the data user side, it only needs one computation round for the aggregation SQL queries including "count" and "sum" functions and two computation rounds for the aggregation SQL queries including "avg" function (1 round for "sum" and 1 round for "count"). For the aggregation SQL queries with Group-By, the number of computation rounds at the data user side depends on the type of aggregation function used in the query which is g for the aggregation functions "count" and "sum" and $2{\times}g$ for the aggregation function "avg", where g is the number of actual and fake groups for the attribute including in the Group-By clause. For the aggregation SQL queries including "Group-By" and "Having", the number of computation rounds at the data user side is 1 for the queries including "count" and "sum" functions and 2 for the queries including "avg" function.

Table 10: Execution of Multi-Dimensional SQL Query including "Count" Function

Account Type	Value of the First Condition	$Result^k_{AccountType}$	Count	Value of the Second Condition	$Result^k_{Count1}$	Count	Value of the Third Condition	$Result^k_{Count2}$
$S(checking)_p$	$S(checking)_p$	$S(1)$	$S(G_1)_p$	$S(G_1)_p$	$S(1)$	$S(G_1)_p$	$S(G_3)_p$	$S(0)$
$S(saving)_p$	$S(checking)_p$	$S(0)$	$S(G_2)_p$	$S(G_1)_p$	$S(0)$	$S(G_2)_p$	$S(G_3)_p$	$S(0)$
$S(checking)_p$	$S(checking)_p$	$S(1)$	$S(G_1)_p$	$S(G_1)_p$	$S(1)$	$S(G_1)_p$	$S(G_3)_p$	$S(0)$

Table 11: Complexity of Our Proposed Approach

Aggregation SQL Query	Computation Rounds at the Server Side	Communication Rounds	Computation Rounds at the Data User Side
Aggregation Queries including "count" Function	1	1	1
Aggregation Queries including "sum" Function	1	1	1
Aggregation Queries including "avg" Function	1	1	2
Aggregation Queries including "Group-By"	1	1	g or $2 \times g$
Aggregation Queries including "Group-By" and "Having"	1	1	1 or 2

Note: g is the number of actual and fake groups.

5 EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate the performance of our proposed approach at the data owner, data user and server sides.

5.1 Setup

We implemented our proposed approach in JAVA programming language and used a machine with 3.60 GHz Intel® Core™ i7-7700 CPU and 16 GB RAM in our experiments.

We used seven attributes ORDERKEY, PARTKEY, SUPPKEY, LINENUMBER, QUANTITY, EXTENDEDPRICE, and DISCOUNT of LINEITEM relation in TPC-H benchmark to generate datasets. To prevent an adversary to learn about the distribution of numerical values of these attributes, we first added a set of zeros to the left side of each numerical value in such a way that all numerical values contain identical digits. Then, we represented each digit into a set of ten numbers, as mentioned in Section 4.1, having only 0s and 1s. To generate the secret-shares for each digit, we selected a polynomial with the degree 1 where the coefficient was randomly generated between 1 and 10. We also defined *Accessibility Ratio* (AR) to control the accessibility of data items in LINEITEM relation. It is the fraction of data items that are accessible. In this case, a user group was randomly selected from the set of user groups defined in the system to be mapped to an accessible data item and all inaccessible data items were mapped to the user group G_0 (i.e., *Min User Group*) by default. The list of parameters along with their values in our experiments is shown in Table 12.

Table 12: List of Parameters and Their Values

Parameter	Value
Number of Servers	5, 15 (default), 25
Number of Tuples	100K (default), 250K, 500K
Number of Attributes	3, 4 (default), 5, 7
Number of User Groups	2 (default), 40, 80
Accessibility Ratio (AR)	10%, 50% (default), 90%
Number of Query Conditions	0, 2 (default), 4

The list of aggregation SQL queries used in our experiments is shown in Table 13. The first letter in the query name indicates the type of aggregation function in the query (*C* for "Count", *S* for "Sum", and *A* for "Avg"), the second and third letters in the query name indicate the type of queries (*SI* for "Simple", *CE* for "Conjunctive Equality" and *DE* "Disjunctive Equality"), and the number at the end of the query name indicates the number of query conditions in the where clause of the query. It is noted that the experimental results for the queries listed in Table 13 with Group-By had the same results and we do not report them in this paper.

The previous works on Homomorphic Encryption, Searchable-Encryption, Bucketization, and Attribute-Based Encryption are either not implemented [3, 13, 23, 32, 42, 51, 64] or do not support aggregation SQL queries [6, 9, 15, 22, 28, 29, 33–35, 39, 40, 43, 49, 50, 61, 63, 66, 69–72]. Therefore, we cannot compare our proposed approach with them. MPC-Based methods [8, 10] can support aggregation SQL queries but they have high overhead for secret-share creation and/or query execution. In [24], it is demonstrated that the use of AA method for oblivious query processing over secret-shared data has better performance compared to the existing MPC-Based techniques [8, 10]. Authors in [24] also showed that the use of AA method is more efficient than a simple strategy of downloading encrypted data, decrypting, and then executing an aggregation SQL query. Hence, we only focused on investigating the performance of our proposed approach here.

5.2 Experimental Results

5.2.1 Computation at the Data Owner Side. Table 14 shows the average time to create secret-shares for LINEITEM relation as well as the average size of generated dataset when the number of attributes and servers are 4 and 15, respectively, and no access policy is attached to this relation. As shown in Table 14, the average size of dataset increases as expected due to the unary representation of values in LINEITEM relation but these secret-shares can be created in a short period of time. Moreover, it is obvious that the average time to create secret-shares and the average size of generated dataset increase by increasing the number of tuples in LINEITEM relation. We also have the similar results when the number of attributes increases. Note that attributes can be added to LINEITEM relation to specify access policies, but the number of attributes added depends on the type of policy attachment (i.e., tuple level and attribute level).

Figure 2 shows the impact of different parameters on the computational overhead of our proposed approach to create secret-shares for both types of policy attachments. As shown in Figure 2a, the computational overhead to create secret-shares for the tuple

Table 13: List of Aggregation SQL Queries

Query Name	Query Definition
C-SI0	select count (orderkey) from lineitem
S-SI0	select sum (orderkey) from lineitem
A-SI0	select avg (orderkey) from lineitem
C-CE2	select count (orderkey) from lineitem where (orderkey=1319) and (partkey=36685)
S-CE2	select sum (orderkey) from lineitem where (orderkey=1319) and (partkey=36685)
A-CE2	select avg (orderkey) from lineitem where (orderkey=1319) and (partkey=36685)
C-CE4	select count (orderkey) from lineitem where (orderkey=1319) and (partkey=36685) and (suppkey=1692) and (linenumber=2)
S-CE4	select sum (orderkey) from lineitem where (orderkey=1319) and (partkey=36685) and (suppkey=1692) and (linenumber=2)
A-CE4	select avg (orderkey) from lineitem where (orderkey=1319) and (partkey=36685) and (suppkey=1692) and (linenumber=2)
C-DE2	select count (orderkey) from lineitem where (orderkey=1319) or (partkey=36685)
S-DE2	select sum (orderkey) from lineitem where (orderkey=1319) or (partkey=36685)
A-DE2	select avg (orderkey) from lineitem where (orderkey=1319) or (partkey=36685)
C-DE4	select count (orderkey) from lineitem where (orderkey=1319) or (partkey=36685) or (suppkey=1692) or (linenumber=2)
S-DE4	select sum (orderkey) from lineitem where (orderkey=1319) or (partkey=36685) or (suppkey=1692) or (linenumber=2)
A-DE4	select avg (orderkey) from lineitem where (orderkey=1319) or (partkey=36685) or (suppkey=1692) or (linenumber=2)

level policy attachment decreases when the number of attributes in LINEITEM relation increases. The reason is that only one new attribute should be added to LINEITEM relation for each aggregation function in our proposed approach and therefore, the computational overhead can decrease by increasing the number of attributes in LINEITEM relation. However, increasing the number of attributes in the case of attribute level policy attachment does not have any effect on the computational overhead due to this fact that two new attributes (for "count" and "sum" functions) should be added in our proposed approach to specify ABAC policies for each attribute of LINEITEM relation. It is also clear from Figure 2b that the number of tuples does not have any effect on the computational overhead for both types of policy attachments since new attributes should be added for each tuple of LINEITEM relation in the process of policy attachment. Based on the experimental results illustrated in Figure 2c, the accessibility ratio does not have any effect on the computational overhead for both types of policy attachments since this parameter only changes the values of ABAC policies attached in LINEITEM relation. However, the computational overhead in the case of tuple level policy attachment is much lower than the attribute level policy attachment. In addition, the number of user groups does not have any effect on the computational overhead for both types of policy attachments as shown in Figure 2d, since we added a set of zeros to the left side of each value in LINEITEM relation to prevent an adversary to learn about the distribution of values. Therefore, it makes all user groups in LINEITEM relation with the identical number of bits.

Based on the experimental results shown here, it is obvious that there is a trade off between enforcing finer granularity of access control and the computational overhead for creating secret-shares. Our proposed approach provides finer granularity to specify ABAC policies in the case of attribute level policy attachment but it imposes more computational overhead compared to the tuple level policy attachment for creating secret-shares at the data owner side.

Table 14: Average Time and Size to Create Secret-Shares

Number of Tuples	Time (Sec.)	Size (MB)
100K	12.7832	94.16
250K	32.1551	235.41
500K	64.5962	470.82

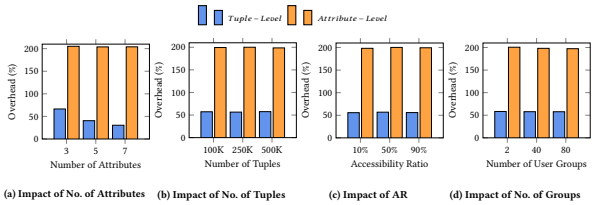


Figure 2: Computational Overhead at the Data Owner Side

5.2.2 Computation at the Server Side. Figure 3 shows the computation time to process aggregation SQL queries shown in Table 13 for both types of policy attachments at the server side. As shown in Figure 3, the computation time to process SQL queries including aggregation function "sum" is longer than SQL queries including aggregation function "count" since we need one more multiplication for each tuple of outsourced relation in this case. It is also clear that the computation time to process SQL queries including aggregation function "avg" is approximately equal to the sum of computation time to process the corresponding SQL queries including aggregation functions "sum" and "count". Based on the experimental results, we can see that as the number of query conditions increases, the computation time also increases, due to increasing the number of multiplications. In addition, the computation time for processing of aggregation SQL queries in the case of attribute level policy attachment is slightly longer than that in the case of tuple level policy attachment because the data fetching time is a bit longer in the case of attribute level policy attachment.

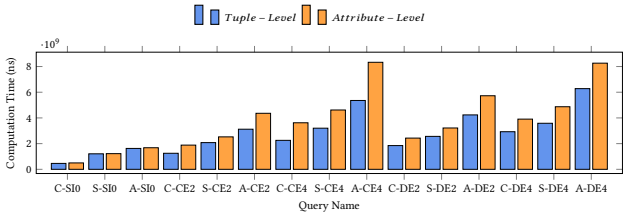


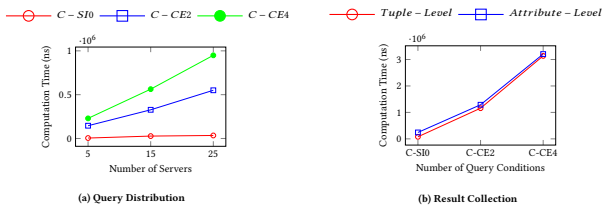
Figure 3: Computation Time for Processing Aggregation SQL Queries

It should be noted that different parameters (i.e., number of attributes, number of tuples, accessibility ratio, and number of user groups) do not have significant impact on the computational

overhead for both types of policy attachments at the server side. The reason is that all tuples in LINEITEM relation are scanned during query processing time and for each tuple, string matching based operators are applied to all attributes involved in query conditions, regardless of the content of their attributes.

5.2.3 Computation at the Data User Side. Figure 4a shows the computation time to create secret-shares for the values in the query conditions of aggregation SQL queries *C-SI0*, *C-CE2*, and *C-CE4* at the data user side when the number of servers were varied. As expected, the computation time increases as the number of servers and the query conditions increase since the data user needs to create more secret-shares in this regard. However, it is clear that this process can be done in a very short period of time.

Figure 4b shows the computation time to perform Lagrange Interpolation operation on the received results from different servers to provide the final answer for the aggregation SQL queries *C-SI0*, and *C-CE2*, *C-CE4* for both types of policy attachments. As expected, the computation time increases as the number of query conditions increases since the data user needs to get the secret-shared query results from more number of servers. It should be noted that similar experimental results have been obtained for other aggregation SQL queries in Table 13.



(a) Query Distribution (b) Result Collection

Figure 4: Computation Time at the Data User Side

6 RELATED WORKS

In theory, it is possible to utilize Fully Homomorphic Encryption (FHE) [9, 34, 35, 43, 61] to perform arbitrary query processing operations on a relation, but it is extremely complex and cannot practically support various types of queries [26, 54, 55]. By contrast, we employ Shamir's secret sharing scheme which can efficiently process aggregation SQL queries over secret-shared data.

Several proposals have been suggested to improve the performance of FHE by utilizing a particular hardware such as GPUs [4, 67, 68, 75], but this technique cannot be applied to low-cost hardware, while our proposed approach can be run to any hardware. Another solution to overcome the shortages of FHE solution is to employ Intel SGX as a hardware-assisted Trusted Execution Environment (TEE) which offers high computational efficiency, generality, and flexibility [5, 12, 56, 60]. However, this solution exposes access patterns due to side channel attacks (i.e., cache time [25, 47, 57], branch shadowing [30], and page fault attacks [20, 48]) on Intel GSX. By contrast, our proposed approach can obliviously evaluate aggregation SQL queries over secret-shared data without revealing access patterns (i.e., hiding the identity of the tuple satisfying a query condition) and query patterns (i.e., hiding which two queries are identical in the case that these queries have at least a condition in the where clause).

Attribute Based Encryption (ABE) is a public key cryptographic technique which can achieve data confidentiality, data privacy and access control in data outsourcing [36, 74]. It is mainly classified into two types, Key-Policy ABE (KP-ABE) and Ciphertext-Policy ABE (CP-ABE). In KP-ABE solutions [3, 23, 38, 51], the ciphertext is based on attributes and the user's secret keys are based on access policies while in CP-ABE solutions [6, 13, 42], the ciphertext is based on access policies and user's secret keys are based on attributes. However, these solutions may naturally expose attributes and access policies to the public and suffer from an inference attack [28, 72]. By contrast, our proposed approach can protect both the privacy of data users and the privacy of access policies.

7 CONCLUSION AND FUTURE WORKS

In this paper, we proposed a fine-grained access control enforcement mechanism tightly integrated with query processing to evaluate aggregation SQL queries (i.e., "count", "sum", and "avg" having single-dimensional, conjunctive, or disjunctive query conditions) over secret-shared data. We designed a set of string matching based operators to obliviously compute query results without the need of communication between servers. Experimental results showed that our proposed approach has lower overhead in the case of tuple level policy attachment and can be used for real-world applications.

In the future, we plan to extend this paper in several directions:

(1) Designing algorithms to process SQL queries including aggregation functions "min" and "max".
(2) Extending our proposed approach to process complex aggregation SQL queries including multiple aggregation functions with inequality in query conditions.
(3) Extending our proposed approach to be used in a malicious adversary model using a proactive secret sharing scheme. In this case, some algorithms should also be proposed to verify the correctness of computation at the server side.

ACKNOWLEDGMENTS

This work was partially funded by the BMWK project SafeFBDC (01MK21002K), the National Research Center ATHENE, and the BMBF project TrustDBle (16KIS1267). We also want to thank hessian.AI at TU Darmstadt as well as DFKI Darmstadt for the support.

REFERENCES

[1] Dinh Tien Tuan Anh and Anwitaman Datta. 2014. Streamforce: outsourcing access control enforcement for stream data to the clouds. In *Proceedings of the 4th ACM conference on Data and application security and privacy*. ACM Press, New York, NY, 13–24.
[2] Varunya Attasena, Jérôme Darmont, and Nouria Harbi. 2017. Secret sharing for cloud data security: a survey. *The VLDB Journal* 26, 5 (June 2017), 657–681. https://doi.org/10.1007/s00778-017-0470-9
[3] Nuttapong Attrapadung, Benoît Libert, and Elie de Panafieu. 2011. Expressive Key-Policy Attribute-Based Encryption with Constant-Size Ciphertexts. In *Proceedings of the International Workshop on Public Key Cryptography*. Springer-Verlag, Berlin, Germany, 90–108.
[4] Ahmad Al Badawi, Bharadwaj Veeravalli, Jie Lin, Nan Xiao, Matsumura Kazuaki, and Aung Khin Mi Mi. 2021. Multi-GPU Design and Performance Evaluation of Homomorphic Encryption on GPU Clusters. *IEEE Transactions on Parallel and Distributed Systems* 32, 2 (Sept. 2021), 379–391.
[5] Sumeet Bajaj and Radu Sion. 2011. TrustedDB: A Trusted Hardware-Based Database with Privacy and Data Confidentiality. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM Press, New York, NY, USA, 205–216.

[6] John Bethencourt, Amit Sahai, and Brent Waters. 2007. Ciphertext-Policy Attribute-Based Encryption. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, USA, 321–334.

[7] Carlo Blundo and Douglas R. Stinson. 1997. Anonymous secret sharing schemes. *Discrete Applied Mathematics* 77, 1 (June 1997), 13–28.

[8] Dan Bogdanov, Sven Laur, and Jan Willemson. 2008. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *Proceedings of the European Symposium on Research in Computer Security*. Springer-Verlag, Berlin, Germany, 192–206.

[9] Dan Boneh, Craig Gentry, Shai Halevi, Frank Wang, and David J. Wu. 2013. Private Database Queries Using Somewhat Homomorphic Encryption. In *Proceedings of International Conference on Applied Cryptography and Network Security*. Springer-Verlag, Berlin, Germany, 102–118.

[10] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. 2010. SEPIA: Privacy-Preserving Aggregation of Multi-Domain Network Events and Statistics. In *the 19th USENIX Security Symposium*. ACM Press, New York, NY, USA, 223–240.

[11] Anirudh Chandramouli, Ashish Choudhury, and Arpita Patra. 2022. A Survey on Perfectly-Secure Verifiable Secret-Sharing. *Comput. Surveys* 54, 11 (Sept. 2022), 1–36. https://doi.org/10.1145/3512344

[12] Yaxing Chen, Qinghua Zheng, Zheng Yan, and Dan Liu. 2021. QShield: Protecting Outsourced Cloud Data Queries With Multi-User Access Control Based on SGX. *IEEE Transactions on Parallel and Distributed Systems* 32, 2 (Feb. 2021), 485–499.

[13] Ling Cheung and Calvin Newport. 2007. Provably secure ciphertext policy ABE. In *Proceedings of the 14th ACM conference on Computer and communications security*. ACM Press, New York, NY, USA, 456–465.

[14] Henry Corrigan-Gibbs and Dan Boneh. 2017. Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. In *14th USENIX symposium on networked systems design and implementation*. ACM Press, New York, NY, USA, 259–282.

[15] Ernesto Damiani, S. De Capitani Vimercati, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. 2003. Balancing confidentiality and efficiency in untrusted relational DBMSs. In *Proceedings of the 10th ACM conference on computer and communications security*. ACM Press, New York, NY, USA, 93–102.

[16] Sabrina De Capitani di Vimercati, Sara Foresti, Sushil Jajodia, Stefano Paraboschi, Gerardo Pelosi, and Pierangela Samarati. 2008. Preserving confidentiality of security policies in data outsourcing. In *Proceedings of the 7th ACM workshop on privacy in the electronic society*. ACM Press, New York, NY, USA, 75–84.

[17] Shlomi Dolev, Niv Gilboa, and Ximing Li. 2019. Accumulating automata and cascaded equations automata for communicationless information theoretically secure multi-party computation. *Theoretical Computer Science* 795, 26 (Nov. 2019), 81–99.

[18] Shlomi Dolev, Peeyush Gupta, Yin Li, Sharad Mehrotra, and Shantanu Sharma. 2021. Privacy-Preserving Secret Shared Computations Using MapReduce. *IEEE Transactions on Dependable and Secure Computing* 18, 4 (Aug. 2021), 1645–1666.

[19] Fatih Emekci, Ahmed Methwally, Divyakant Agrawal, and Amr ElAbbadi. 2014. Dividing secrets to secure data outsourcing. *Information Sciences* 263 (April 2014), 198–210.

[20] Shufan Fei, Zheng Yan, Wenxiu Ding, and Haomeng Xie. 2022. Security Vulnerabilities of SGX and Countermeasures: A Survey. *Comput. Surveys* 54, 6 (July 2022), 1–36.

[21] Keith Frikken, Mikhail Atallah, and Jiangtao Li. 2006. Attribute-Based Access Control with Hidden Policies and Hidden Credentials. *IEEE Trans. Comput.* 55, 10 (Aug. 2006), 1259–1270.

[22] Chunpeng Ge, Willy Susilo, Zhe Liu, Jinyue Xia, Pawel Szalachowski, and Liming Fang. 2021. Secure Keyword Search and Data Sharing Mechanism for Cloud Computing. *IEEE Transactions on Dependable and Secure Computing* 18, 6 (Nov. 2021), 2787–2800.

[23] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. 2006. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM conference on computer and communications security*. ACM Press, New York, NY, USA, 89–98.

[24] Peeyush Gupta, Yin Li, Sharad Mehrotra, Nisha Panwar, Shantanu Sharma, and Sumaya Almanee. 2022. Obscure: Information-Theoretically Secure, Oblivious, and Verifiable Aggregation Queries on Secret-Shared Outsourced Data. *IEEE Transactions on Knowledge and Data Engineering* 34, 2 (March 2022), 843–864.

[25] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*. ACM Press, New York, NY, USA, 1–6.

[26] Mohammad Ali Hadavi, Rasool Jalili, Ernesto Damiani, and Stelvio Cimato. 2015. Security and searchability in secret sharing-based data outsourcing. *International Journal of Information Security* 14 (Nov. 2015), 513–529.

[27] Mohammad Ali Hadavi, Rasool Jalili, and Leila Karimi. 2016. Access control aware data retrieval for secret sharing based database outsourcing. *Distributed and Parallel Databases* 34 (Dec. 2016), 505–534.

[28] Jialu Hao, Jian Liu, Huimei Wang, Lingshuang Liu, Ming Xian, and Xuemin Shen. 2019. Efficient Attribute-Based Access Control With Authorized Search in Cloud Storage. *IEEE Access* 7 (March 2019), 182772–182783.

[29] Bijit Hore, Sharad Mehrotra, and Gene Tsudik. 2004. A privacy-preserving index for range queries. In *Proceedings of the 13th international conference on very large data bases*. Springer-Verlag, Berlin, Germany, 720–731.

[30] Shohreh Hosseinzadeh, Hans Liljestrand, Ville Leppänen, and Andrew Paverd. 2018. Mitigating Branch-Shadowing Attacks on Intel SGX using Control Flow Randomization. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*. ACM Press, New York, NY, USA, 42–47.

[31] Vincent C. Hu, David Ferraiolo, Rick Kuhn, Arthur R. Friedman, Alan J. Lang, Margaret M. Cogdell, Adam Schnitzer, Kenneth Sandlin, Robert Miller, and Karen Scarfone. 2013. *Guide to Attribute Based Access Control (ABAC) Definition and Considerations (Draft)*. NIST Special Publication. National Institute of Standards and Technology (NIST).

[32] Junbeom Hur and Dong Kun Noh. 2011. Attribute-Based Access Control with Efficient Revocation in Data Outsourcing Systems. *IEEE Access* 22, 7 (Nov. 2011), 1214–1221.

[33] Adel Jebali, Salma Sassi, Abderrazak Jemaia, and Richard Chbeir. 2022. Secure data outsourcing in presence of the inference problem: A graph-based approach. *J. Parallel and Distrib. Comput.* 160 (Feb. 2022), 1–15.

[34] Myungsun Kim, Hyung Tae Lee, San Ling, Benjamin Hong Meng Tan, and Huaxiong Wang. 2019. Private Compound Wildcard Queries Using Fully Homomorphic Encryption. *IEEE Transactions on Dependable and Secure Computing* 16, 5 (Oct. 2019), 743–756.

[35] Myungsun Kim, Hyung Tae Lee, San Ling, and Huaxiong Wang. 2018. On the Efficiency of FHE-Based Private Queries. *IEEE Transactions on Dependable and Secure Computing* 15, 2 (May 2018), 357–363.

[36] P. Praveen Kumar, P .Syam Kumar, and P. J. A. Alphonse. 2018. Attribute based encryption in cloud computing: A survey, gap analysis, and future directions. *Journal of Network and Computer Applications* 108, 15 (April 2018), 37–52.

[37] Jae-Gil Lee, Kyu-Young Whang, Wook-Shin Han, and Il-Yeol Song. 2007. The dynamic predicate: integrating access control with query processing in XML databases. *The VLDB Journal* 16, 3 (July 2007), 371–387.

[38] Allison Lewko, Amit Sahai, and Brent Waters. 2010. Revocation Systems with Very Small Private Keys. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, USA, 273–285.

[39] Jingwei Li, Jin Li, Zheli Liu, and Chunfu Jia. 2014. Enabling efficient and secure data sharing in cloud computing. *Concurrency and Computation: Practice and Experience* 26, 5 (April 2014), 1052–1066.

[40] Jingwei Li, Dan Lin, Anna Cinzia Squicciarini, Jin Li, and Chunfu Jia. 2017. Towards Privacy-Preserving Storage and Retrieval in Multiple Clouds. *IEEE Transactions on Cloud Computing* 5, 3 (Oct. 2017), 499–509.

[41] Ming Li, Shucheng Yu, Kui Ren, Wenjing Lou, and Y. Thomas Hou. 2013. Toward privacy-assured and searchable cloud data storage services. *IEEE Network* 27, 4 (Aug. 2013), 56–62.

[42] Xiaohui Liang, Zhenfu Cao, Huang Lin, and Dongsheng Xing. 2009. Provably secure and efficient bounded ciphertext policy attribute based encryption. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*. ACM Press, New York, NY, USA, 343–352.

[43] Murali Mani. 2013. Enabling secure query processing in the cloud using fully homomorphic encryption. In *Proceedings of the Second Workshop on Data Analytics in the Cloud*. ACM Press, New York, NY, USA, 36–40.

[44] Yaser Mansouri, Adel Nadjaran Toosi, and Rajkumar Buyya. 2018. Data Storage Management in Cloud Environments: Taxonomy, Survey, and Future Directions. *Comput. Surveys* 50, 6 (Nov. 2018), 1–51.

[45] Meghdad Mirabi, Hamidah Ibrahim, Leila Fathi, Nur Izura Udzir, and Ali Mamat. 2015. A Dynamic Compressed Accessibility Map for Secure XML Querying and Updating. *Journal of Information Science & Engineering* 31, 1 (Jan. 2015), 59–93.

[46] Meghdad Mirabi, Hamidah Ibrahim, Nur Izura Udzir, and Ali Mamat. 2012. A Compact Bit String Accessibility Map for Secure XML Query Processing. *Procedia Computer Science* 10 (Aug. 2012), 1172–1179.

[47] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems*. Springer-Verlag, Berlin, Germany, 69–90.

[48] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, USA, 1466–1482.

[49] Mohamed Nabeel and Elisa Bertino. 2012. Privacy preserving delegated access control in the storage as a service model. In *Proceedings of the IEEE 13th International Conference on Information Reuse & Integration*. IEEE Computer Society, Washington, DC, USA, 645–652.

[50] Mohamed Nabeel and Elisa Bertino. 2014. Privacy Preserving Delegated Access Control in Public Clouds. *IEEE Transactions on Knowledge and Data Engineering* 26, 9 (April 2014), 2268–2280.

[51] Rafail Ostrovsky, Amit Sahai, and Brent Waters. 2007. Attribute-based encryption with non-monotonic access structures. In *Proceedings of the 14th ACM conference on computer and communications security*. ACM Press, New York, NY, USA, 195–203.

[52] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting confidentiality with encrypted query

processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM Press, New York, NY, USA, 85–100.

[53] Amit Sahai and Brent Waters. 2005. Fuzzy Identity-Based Encryption. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer-Verlag, Berlin, Germany, 457–473.

[54] Muhammad I. Sarfraz, Mohamed Nabeel, Jianneng Cao, and Elisa Bertino. 2015. DBMask: Fine-Grained Access Control on Encrypted Relational Databases. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. ACM Press, New York, NY, USA, 1–11.

[55] Muhammad I. Sarfraz, Mohamed Nabeel, Jianneng Cao, and Elisa Bertino. 2016. DBMask: Fine-Grained Access Control on Encrypted Relational Databases. *Transactions on Data Privacy* 9, 3 (March 2016), 187–214.

[56] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *Proceedings of 2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, USA, 38–54.

[57] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2020. Malware Guard Extension: abusing Intel SGX to conceal cache attacks. *Cybersecurity* 3, 2 (Dec. 2020). https://doi.org/10.1186/s42400-019-0042-y

[58] Daniel Servos and Sylvia L. Osborn. 2017. Current Research and Open Problems in Attribute-Based Access Control. *Comput. Surveys* 49, 4 (Dec. 2017), 1–45.

[59] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (Nov. 1979), 612–613.

[60] Wenhai Sun, Ruide Zhang, Wenjing Lou, and Y. Thomas Hou. 2018. REAR-GUARD: Secure Keyword Search Using Trusted Hardware. In *Proceedings of IEEE Conference on Computer Communications*. IEEE Computer Society, Washington, DC, USA, 801–809.

[61] Benjamin Hong Meng Tan, Hyung Tae Lee, Huaxiong Wang, Shuqin Ren, and Khin Mi Mi Aung. 2021. Efficient Private Comparison Queries Over Encrypted Databases Using Fully Homomorphic Encryption With Finite Fields. *IEEE Transactions on Dependable and Secure Computing* 18, 6 (Jan. 2021), 2861–2874.

[62] Cory Thoma, Adam J. Lee, and Alexandros Labrinidis. 2016. PolyStream: Cryptographically Enforced Access Controls for Outsourced Data Stream Processing. In *Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies*. ACM Press, New York, NY, USA, 227–238.

[63] Cong Wang, Ning Cao, Kui Ren, and Wenjing Lou. 2012. Enabling Secure and Efficient Ranked Keyword Search over Outsourced Cloud Data. *IEEE Transactions on Parallel and Distributed System* 23, 8 (Dec. 2012), 1467–1479.

[64] Guojun Wang, Qin Liu, and Jie Wu. 2010. Hierarchical attribute-based encryption for fine-grained access control in cloud storage services. In *Proceedings of the 17th ACM conference on computer and communications security*. ACM Press, New York, NY, USA, 735–737.

[65] Lingyu Wang, Duminda Wijesekera, and Sushil Jajodia. 2004. A logic-based framework for attribute based access control. In *Proceedings of the 2004 ACM workshop on formal methods in security engineering*. ACM Press, New York, NY, USA, 45–55.

[66] Shiyuan Wang, Divyakant Agrawal, and Amr El Abbadi. 2011. A Comprehensive Framework for Secure Query Processing on Relational Data in the Cloud. In *Workshop on Secure Data Management*. Springer-Verlag, Berlin, Germany, 52–69.

[67] Wei Wang, Zhilu Chen, and Xinming Huang. 2014. Accelerating leveled fully homomorphic encryption using GPU. In *Proceedings of 2014 IEEE International Symposium on Circuits and Systems*. IEEE Computer Society, Washington, DC, USA, 2800–2803.

[68] Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, and Berk Sunar. 2012. Accelerating fully homomorphic encryption using GPU. In *Proceedings of 2012 IEEE Conference on High Performance Extreme Computing*. IEEE Computer Society, Washington, DC, USA, 1–5.

[69] Jianghong Wei, Wenfen Liu, and Xuexian Hu. 2018. Secure and Efficient Attribute-Based Access Control for Multiauthority Cloud Storage. *IEEE Systems Journal* 12, 2 (June 2018), 1731–1742.

[70] Jianghong Wei, Wenfen Liu, and Xuexian Hu. 2018. Secure Data Sharing in Cloud Computing Using Revocable-Storage Identity-Based Encryption. *IEEE Transactions on Cloud Computing* 6, 4 (Oct. 2018), 1136–1148.

[71] Yingjie Xue, Kaiping Xue, Na Gai, Jianan Hong, David S. L. Wei, and Peilin Hong. 2019. An Attribute-Based Controlled Collaborative Access Control Scheme for Public Cloud Storage. *IEEE Transactions on Information Forensics and Security* 14, 11 (April 2019), 2927–2942.

[72] Kan Yang, Qi Han, Hui Li, Kan Zheng, Zhou Su, and Xuemin Shen. 2017. An Efficient and Fine-Grained Big Data Access Control Scheme With Privacy-Preserving Policy. *IEEE Internet of Things Journal* 4, 2 (April 2017), 563–571.

[73] Xinwen Zhang, Yingjiu Li, and Divya Nalla. 2005. An attribute-based access matrix model. In *Proceedings of the 2005 ACM Symposium on Applied Computing*. ACM Press, New York, NY, USA, 359–363.

[74] Yinghui Zhang, Robert H. Deng, Shengmin Xu, Jianfei Sun, Qi Li, and Dong Zheng. 2021. Attribute-based Encryption for Cloud Computing Access Control: A Survey. *Comput. Surveys* 53, 4 (Aug. 2021), 83:1–41.

[75] Özgün Özerk, Can Elgezen, Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. 2022. Efficient number theoretic transform implementation on GPU for homomorphic encryption. *The Journal of Supercomputing* 78 (Feb. 2022), 2840–2872.