MASTER THESIS

Access Pattern Hiding Aggregation over Encrypted Databases

Y.A.M. Kortekaas (Yoep)

Faculty of Electrical Electrical Engineering, Mathematics, and Computer Science
Services and Cyber Security

EXAMINATION COMMITTEE
dr. A. Peter (Andreas)
dr. C.G. Zeinstra (Chris)
dr.ing. F.W. Hahn (Florian)

October 7th, 2020

UNIVERSITY OF TWENTE.

Abstract—**Outsourcing storage and computation to cloud service providers is becoming increasingly more popular for enterprises. Multiple secure aggregation schemes, such as CryptDB, Seabed, and SAGMA, have been proposed to protect enterprise data while allowing function evaluation over the data. However, multiple leakage-abuse attacks on the access pattern have been published that can also be used to attack these existing schemes. On top of that, without pre-computation these schemes allow only for additive aggregation functions. We propose and implement a secure aggregation scheme based on leveled-fully homomorphic encryption that is not vulnerable to leakage-abuse attacks that exist for the current secure aggregation schemes, and can evaluate both additive and multiplicative aggregation functions. We compare our scheme against SAGMA, and with a security level of 112 bits, multiplicative depth of 13 and plaintext space of 58 bits, we achieve between 2x and 8x better query evaluation wall-time.**

Index Terms—**Secure aggregation, Fully Homomorphic Encryption, Somewhat Homomorphic Encryption, cloud computing**

I. Introduction

The number of enterprises using cloud computing services has been steadily increasing over the last years. In 2018, statistics released by Eurostat revealed that in Europe, 26% of all enterprises used cloud computing services [1], of which 48% host their entire database in the cloud. On the other hand, more and more companies are creating revenue out of their data, it is even giving them a competitive advantage. A study by the European Commission [2] forecasts that by 2020, the total value of the EU data economy will increase to 739 billion euro, being worth approximately 4% of the EU GDP. The combination of these two statistics leads to the important question of whether companies can trust external cloud service providers to securely store and process their valuable business intelligence.

One naive approach to mitigate this issue of trust is to encrypt the entire database using semantically secure encryption, and store it encrypted on the cloud platform. However, one can easily see that with this naive solution that when the client wants to actually evaluate functions over the data, the client needs to download the entire database, decrypt it and evaluate the function. This approach is unfeasible and would completely remove all benefits that enterprises gain from cloud computing.

A. Problem Statement

Looking at traditional, relational databases we see two main functionalities, namely searching and aggregation. Secure and efficient searching through databases has been actively researched over 20 years now, starting with the work of Song, Wagner and Perrig in 2000 [3]. Aggregation, in the context of aggregation in relational databases, has been a less active field of research, starting with the work by Hacigümüş et al in 2002 [4].

In secure aggregation we have two main parties, the *query issuer* and the *aggregator*. The query issuer opens a transaction and sends aggregation queries to the aggregator, which will evaluate the query over the encrypted database. The encrypted result will then be sent back by the aggregator to the query issuer, which can then decrypt the result, finishing the transaction. A schematic overview of this interaction is shown in Figure 1.

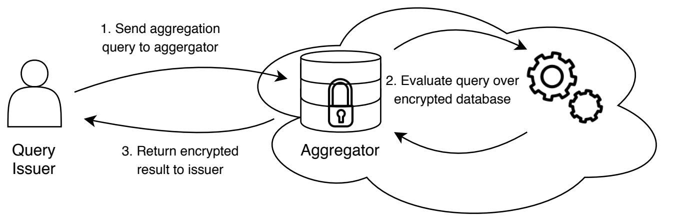

1. Send aggregation query to aggergator
2. Evaluate query over encrypted database
3. Return encrypted result to issuer
Query Issuer
Aggregator

Fig. 1. Basic overview of interaction between the query issuer and the aggregator

Recent works looking into storing databases at external parties while providing some level of security against the hosting provider are CryptDB [5], Seabed [6], and SAGMA [7]. These secure aggregation systems use either deterministic encryption (CryptDB, Seabed[1]) or searchable encryption (SAGMA) to facilitate grouping and filtering in aggregation queries. However, these filtering and grouping techniques are vulnerable to attacks that leak the access pattern, such as the attacks on property preserving encryption by Naveed et al. [8] and Grubbs et al. [9], and the attacks on searchable encryption by Islam et al. [10] and Cash et al. [11]. Both Seabed and SAGMA do propose mitigation techniques against frequency analysis attacks, however, these techniques are based on attribute frequencies of vulnerable columns at the setup of the database, and do not hold up for dynamic databases where the attribute frequencies can change.

Next to being vulnerable to leakage-abuse attacks, these schemes all lack functionality compared to traditional plaintext databases. This is caused either by limitations of the underlying cryptographic scheme (CryptDB, Seabed, SAGMA), and in the case of pre-computation (Seabed) by limitations on the storage space.

Leveled-Fully homomorphic encryption will allow us to construct an aggregation scheme that is semantically secure and hides the access pattern, protecting query issuers from the leakage-abuse attacks on current secure aggregation systems. On top of that, it allows us to evaluate more expressive queries, consisting of additive *and* multiplicative functions, without the storage increase of pre-computation, while having restricted interactivity between query issuer and aggregator.

B. Contributions

In this work we introduce a secure aggregation solution that makes use of a leveled-Fully Homomorphic Encryption scheme, that has both security guarantees similar to those of traditional encryption, is not vulnerable to previous attacks on the access pattern, and through the use of homomorphic

[1]Not in all cases, see Section III

encryption enables computation over encrypted data, therefore allowing enterprises to enjoy the benefits gained by moving the server infrastructure into the cloud. More concretely:

- We define and implement a dynamic secure aggregation system with filtering and grouping that is not vulnerable to leakage-abuse attacks that affect current secure aggregation systems. We define our system in such a way that it can be extended with other leveled-FHE schemes, and provide several default functionalities, such as calculation of the sum, average and standard deviation. (Section IV)

- We perform a rigorous parameter selection for our scheme for the BFVrns cryptographic scheme, and compare our scheme against the SAGMA system by Hackenjos et al. [7]. We conclude that our aggregation scheme provides: 1) protection against attacks leaking the access pattern; 2) more expressive queries than possible in SAGMA, containing additions *and* multiplications; 3) better aggregation time in terms of wall-time, including two times better performance evaluating grouped COUNT queries, four times better performance evaluating grouped SUM queries, and eight times better performance evaluating AVG queries. (Section V-C, Section V-D, and Section V-F)

- We evaluate the effect of the multiplicative depth, the message space, and the security level on computational and storage efficiency of our aggregation scheme in combination with the BFVrns cryptographic scheme. For our chosen parameters: 1) computational and storage efficiency scales linearly with the multiplicative depth; 2) the message space does not directly influence the computational and storage efficiency of our scheme, the number of CRT primes required to represent a ciphertext element directly influences the efficiency; 3) lower security levels (80 - 192 bits) do not influence the efficiency of our solution, for a security level of 256 bits we do see an overall decrease in computational and storage efficiency. (Section V-E)

C. Organization

The remainder of this work is structured as follows. In Section II we will discuss some relevant background information regarding the research, and in Section III we discuss several other notable works regarding secure aggregation over databases. In Section IV we will introduce and discuss the structure of our solution and different constructions that are used in the proposed solution. In Section V we evaluate the performance of our solution, and compare it with the performance and security of the current state of the art. We conclude by giving a brief overview of our work and highlight some interesting future directions of research in this field in Section VI.

II. Background & Notation

A. Databases and Aggregation

Relational databases and relational database management systems (RDBMS) such as PostgreSQL, MariaDB and MySQL are widely used in industry. For example, at the time of writing, PostgreSQL has had more than 30 years of active development going on, by over 600 of open source developers, and has millions of users[2]. Almost all current relational databases can be queried using (a dialect of) the Structured Query Language (SQL). An example of an SQL statement can be seen in Listing 1. SQL uses SELECT statements for information retrieval, and supports insertion, deletion and updating of records through (similarly structured) INSERT, DELETE and UPDATE statements.

```
SELECT column1, column2, ...
FROM table1, table2, ...
[WHERE condition is true]
[ORDER BY column1 [ASC/DESC], ...]
```

Listing 1. Example SQL SELECT statement

Next to retrieving data, results can also be aggregated using *aggregation functions*, as seen in Listing 2. For our definition of aggregation functions we use the definition by Oracle: *"Aggregate functions return a single result row based on groups of rows, rather than on single rows"*[3]. Commonly used aggregation functions are COUNT, AVG, and SUM, which respectively (as the name suggests) count the number of records matching a filter, and compute the average and sum of a column. By default, aggregation functions compute their aggregation for a whole column. Aggregation can be customized in three ways. First we can add a DISTINCT clause to only consider the aggregation of unique rows. Second we can filter the values that will be aggregated with a WHERE clause. Third, aggregations can be made more specific by adding a GROUP BY clause to the SQL statement. With the GROUP BY statement we can select one or more categorical columns which we will group our computation by. For example, if we query a COUNT and we group by a categorical column with categories c_1 and c_2, the RDBMS will return a count of all rows with c_1 in the grouping column and a count of all rows with c_2 in the grouping column.

Next to grouping the aggregation results with a GROUP BY clause, we can also filter the computed aggregation results with a HAVING clause. The HAVING clause is similar to the WHERE clause, but instead of filtering the results *before* aggregation, HAVING filters the results *after* aggregation.

[2]https://www.postgresql.org/about/

[3]https://docs.oracle.com/database/121/SQLRF/functions003.htm#SQLRF20035

```
SELECT column1, agg_func(column2),
       agg_func(DISTINCT column2), ...
FROM table1, ...
[GROUP BY column1, ...]
[HAVING agg_func(column2) > x]
```

Listing 2. Example SQL aggregation statement

B. Homomorphic Encryption

We create a secure aggregation scheme with more functionality and better protection against information leakage than previous schemes, with the help of leveled-Fully *Homomorphic* Encryption. Homomorphic encryption is a special subclass of encryption, where some mathematical structure is present in the ciphertexts that allows for computation on the encrypted data. This mathematical structure is used to define two operators $\square$ and $\circ$. In a homomorphic scheme, we have the property that $Enc(m_1 \square m_2) = Enc(m_1) \circ Enc(m_2)$.

A homomorphic encryption scheme consists of four probabilistic polynomial time algorithms, namely:

- **KeyGen(λ).** The key generation algorithm takes as input security parameter λ, and produces public encryption key pk, secret decryption key sk, and public evaluation key evk.
- **Encrypt(pk, m).** The encryption algorithm takes as input the public key pk and the message m, and outputs ciphertext c, the encryption of message m.
- **Decrypt(sk, c).** The decryption algorithm takes as input the public key sk and the ciphertext c that is encrypted under the public key pk corresponding to sk, and outputs message m', the decryption of ciphertext c.
- **Eval($evk, f, c_1, ..., c_n$).** The evaluation algorithm takes as input the evaluation key evk, the function to evaluate $f : \{c\}^n \rightarrow c_f$, the inputs $(c_1, ..., c_n)$ to function f, and outputs the result c_f of evaluating f on the inputs. We require that the homomorphic encryption scheme is *compact*, i.e. the size of c_f is independent of f and the number of inputs n to f.

We classify homomorphic encryption schemes in three different classes, partial homomorphic encryption schemes (PHE), somewhat homomorphic encryption schemes (SWHE) and (leveled-)fully homomorphic encryption schemes (FHE). For PHE schemes, the *Eval* algorithm can only evaluate an arbitrary number of functions with a single operator, addition or multiplication. In SWHE schemes the *Eval* algorithm can evaluate functions with both operators, addition and multiplication, but it cannot evaluate an arbitrary number of addition *and* multiplication operators. In (leveled-)FHE schemes, the *Eval* algorithm can evaluate an arbitrary number of functions with both operators, addition and multiplication. The main difference between leveled- and pure-FHE schemes is the size of the evaluation key. Pure FHE schemes assume circular security, i.e. assuming the encryption of the secret key under the corresponding public key is secure, while leveled-FHE schemes do not make that assumption. This leads to leveled-FHE schemes having an evaluation key that is linear in size in relation to the function depth that the scheme needs to evaluate, while pure-FHE schemes have an evaluation key size that is constant in relation to the function depth that the scheme needs to evaluate.

C. Learning With Errors

Several of the newer FHE schemes, like BV [12], BFV [13], and BGV [14] base their security on the (ring) learning with errors assumption. The Learning With Errors (LWE) problem was first introduced by Oded Regev in 2005 [15], and later extended to a ring-based version by Lyubashevsky et al. (R-LWE) [16]. The LWE problem is a generalization of the so-called *learning from parity with errors* problem. Brakerski, Gentry and Vaikuntanathan propose a more general Learning With Errors (G-LWE) problem in [14], which we also make use of in this paper:

Definition 1: (G-LWE - Definition 5 in [14]) For security parameter λ, let $n = n(\lambda)$ be an integer dimension, let $f(x) = x^d + 1$ where $d = d(\lambda)$ is a power of 2, let $q = q(\lambda) \geq 2$ be a prime integer, let $R = \mathbb{Z}[x]/(f(x))$ and $R_q = R/qR$, and let $\chi = \chi(\lambda)$ be a distribution over R. The GLWE$_{n,f,q,\chi}$ problem is to distinguish the following two distributions: In the first distribution, one samples $(\boldsymbol{a_i}, b_i)$ uniformly from R_q^{n+1}. In the second distribution, one draws $s \leftarrow R_q^n$ uniformly, and then samples $(\boldsymbol{a_i}, b_i) \in R_q^{n+1}$ by sampling $\boldsymbol{a_i} \leftarrow R_q^n$ uniformly, $e_i \leftarrow \chi$, and setting $b_i = \langle \boldsymbol{a_i}, \boldsymbol{s} \rangle + e_i$. The GLWE$_{n,f,q,\chi}$ assumption is that the GLWE$_{n,f,q,\chi}$ problem is infeasible.

With this G-LWE definition, we can instantiate standard LWE by taking $d = 1$, and instantiate Ring-LWE by taking $n = 1$.

III. Related Work

In the following section we will discuss several works that have already explored the concept of secure aggregation. This section aims to give the reader a clear understanding of what the current state of the art regarding secure aggregation is, and what benefits and drawbacks current works have.

CryptDB ([5]): The CryptDB system was introduced by Popa et al [5]. CryptDB combines, using so-called *"onions of encryption"*, several cryptographic schemes in order to perform filtering and aggregation at the server-side. Aggregation functions are evaluated using the Pallier *additively homomorphic* encryption scheme. For grouping and filtering, CryptDB resorts to deterministic and order preserving encryption, enabling frequency-analysis attacks on the database, as shown by Naveed et al. [8] and Grubbs et al. [9].

In short, CryptDB combines several different encryption schemes to create an 'SQL-aware' secure aggregation solution, which can support multiple users. In their research they find that over a trace of 126 million SQL queries, that their scheme can securely answer 99.5% of them with an efficiency penalty of 14.5% in the best case and 26% in the worst case.

Seabed ([6]): Seabed was introduced by Papadimitriou et al. in 2016 as a reaction to (among others) the CryptDB system. Papadimitriou et al. noticed that for aggregation and analytics, CryptDB used the Pallier asymmetric homomorphic encryption scheme, which compared to symmetric encryption schemes has a quite high computational cost. Pallier is useful in the setting where the data aggregator does not fully trust the data producer, but if the producer is trusted, Papadimitriou et al. show with their Additive Symmetric Homomorphic Encryption (ASHE) that using symmetric encryption is sufficient. Next to ASHE, Papadimitriou et al. also introduce Splayed ASHE (SplASHE) for used for grouping the aggregation results, while mitigating frequency analysis attacks. Both ASHE and SplASHE are used in Seabed, the system that Papadimitriou et al. introduce for large scale secure data analytics/aggregation.

An interesting note is that homomorphic operations in (spl)ASHE grow the ciphertext, and slow down the decryption algorithm. With each homomorphic addition we update an index set that describes which rows have been combined. Databases can be optimized for this apriori, such that we store ranges of rows instead of individual rows, but as more records are inserted the effect of this optimization will diminish.

In SplASHE, grouping columns are encoded using one-hot encoding, or 'spreading out' the column over multiple 'binary' columns. To be a bit more concrete, if we have a column C with d distinct values ($\{c_1, ..., c_d\}$), we replace C with d new columns, and if a row had value c_i in column C, it will now have that value in column i of the new columns, and will have encryptions of 0 in the other columns.

In order to limit the blow-up of columns when d is large for column C, Papadimitriou et al. propose to only apply SplASHE on the k most common values of column C, and store the other values in a single column, encrypted with deterministic encryption. The rows of the spread-out column are used to make sure the deterministically encrypted variable is distributed uniformly. The advantage of this improvement is that the column blow-up is limited, but if the distribution of values in column C changes over time, frequency analysis attacks, such as those we saw for CryptDB by Naveed et al. [8] and Grubbs et al. [9] become possible.

SAGMA ([7]): The SAGMA (Secure Aggregation Grouped by Multiple Attributes) system was introduced by Hackenjos et al. in 2020, and aims to have aggregation with grouping, without the column blow-up that Seabed has, and without relying on deterministic encryption. For filtering, SAGMA uses row-wise encryption in combination with searchable encryption. SAGMA is not completely SQL-aware like CryptDB, but instead focuses on aggregation like Seabed does. The SAGMA system supports the COUNT, SUM and AVERAGE aggregation functions, and allows grouping on t different attributes, where t is a parameter that is fixed at initialization. For aggregation, SAGMA divides the grouping attributes in buckets, and uses packed ciphertexts to aggregate per bucket. Bucket membership is determined with searchable encryption, and within the bucket, aggregation is done by shifting the encrypted value using a mapping function, and summing the shifted values. The cryptographic scheme that Hackenjos et al. use for this is the BGN scheme, that allows one multiplication (for shifting) and multiple additions (for aggregation). In the paper, Hackenjos et al. propose three different bucketization/partition methods to create equally distributed buckets, which are necessary to minimize information leakage that is introduced by bucketization.

The main shortcoming with SAGMA however, is use of the BGN cryptographic scheme. In order to decrypt BGN ciphertexts, the aggregator needs to compute the discrete logarithm of the message, which is assumed to be computationally hard. Furthermore, due to BGN only supporting one multiplication, the mapping/shifting function (modeled as a polynomial) is stored on the server as a combination of plaintext coefficients and encrypted monomials. The plaintext coefficients can be multiplied with the monomials without 'using up' the single homomorphic multiplication, and the shift can then be determined from summing up the monomials multiplied by the coefficients. However, SAGMA requires the server to store $(B-1)^i$ monomials in order to group on i different attributes with bucket size B, which means that the amount of monomials we need to store for grouping grows exponentially with the amount of attributes. A less significant issue (but an issue none-the-less) is the limited aggregation functionality of SAGMA, as it only supports the COUNT, SUM, and AVG operation.

IV. Constructions

For this work we have implemented three types of numerical constructions, that are used to implement the secure aggregation solution. These constructions are build up in such a way that our secure aggregation solution can be instantiated with any leveled-FHE scheme that supports homomorphic addition, homomorphic subtraction, and homomorphic multiplication. We have also provided 'packed' versions of the constructions, in the case that the cryptographic scheme supports ciphertext packing.

The constructions we implement for this work can be grouped in two different categories, numerical constructions that we discuss in Section IV-A and database related data structures that we discuss in Section IV-B. We end this section with a description of how query construction works, and a general application flow.

A. Numerical Constructions

For our work we created three main numerical constructions in two different versions, one version with ciphertext packing and one version without ciphertext packing, to facilitate our secure aggregation. We implement an integer construction, a bit construction based on that integer construction, and a binary integer construction based on the bit construction. Each

TABLE I
COMPUTATION COST OF BIT CONSTRUCTION OPERATIONS

Operation	Mult. Depth	Add	Sub	Mult
NOT	0	0	1	0
AND	1	0	0	1
NAND	1	0	1	1
OR	1	1	1	1
NOR	1	1	2	1
XOR	1	1	2	1
XNOR	1	1	3	1

of these constructions will be discussed in their own respective subsection.

Each of these constructions can be instantiated with a leveled-FHE scheme that supports at least homomorphic addition, homomorphic subtraction, and homomorphic multiplication. If the encryption scheme also supports some form of ciphertext packing, special 'packed' versions of these constructions can be instantiated. These work the same as their non-packed variants, but differ in the fact that they encrypt and decrypt a vector of numbers, instead of a single number. Furthermore for the packed ciphertexts we require some ability to permute/shift the 'slots' of the ciphertexts[4], in order to homomorphically sum up the elements encoded into a packed ciphertext.

1) Integer Construction: The integer construction is the most straightforward construction we implement. The integer construction is used in our solution to represent plain integers, and is a wrapper around the ciphertext of the underlying cryptographic scheme which standardizes the function names. Next to that, the integer construction also supports handling multiple instantiations of the scheme, in the case that the scheme only supports a limited plaintextspace that is too small for the aggregation, and we want to extend the plaintextspace using the Chinese remainder theorem.

2) Bit Construction: The bit construction is used in our solution to represent, as the name implies, an encrypted bit value. The bit construction is based on the integer construction, with its domain restricted to 0 and 1 values. These limitation allows us to define a set of unary and binary operations, such as inversion and the set of binary logic gates (NOT, (N)AND, (N)OR, and X(N)OR). The cost of these logic gates can be found in Table I

Bit constructs can also be applied to/multiplied with integer and binary integer constructs. This is used in the secure aggregation as a way to include or exclude values from aggregation.

3) Binary Integer Construction: The binary integer construction in our solution represents integer values with a collection of encrypted bit constructions. This representation is significantly more computation and memory intensive than the normal integer construction, but does allow for more functionality.

[4]Intuitively, we want to be able to homomorphically rearrange/shift the vector that is encoded in the ciphertext

TABLE II
COMPUTATION COST OF BINARY INTEGER CONSTRUCTION COMPARISON OPERATIONS, FOR BINARY INTEGER CONSTRUCTIONS OF k BITS.

Operation	Mult. Depth	Add	Sub	Mult
$A == B$	$\lceil \log_2(k) \rceil + 1$	k	$3k$	$2k-1$
$A < B$	$2\lceil \log_2(k) \rceil + 1$	$2k-1$	$6k-2$	$5k-3$
$A <= B$	$2\lceil \log_2(k) \rceil + 1$	$2k-1$	$6k-1$	$5k-3$
$A > B$	$2\lceil \log_2(k) \rceil + 1$	$2k-1$	$6k-2$	$5k-3$
$A >= B$	$2\lceil \log_2(k) \rceil + 1$	$2k-1$	$6k-1$	$5k-3$

We use the binary integer construction to facilitate comparison of two values, for example testing equality and order.

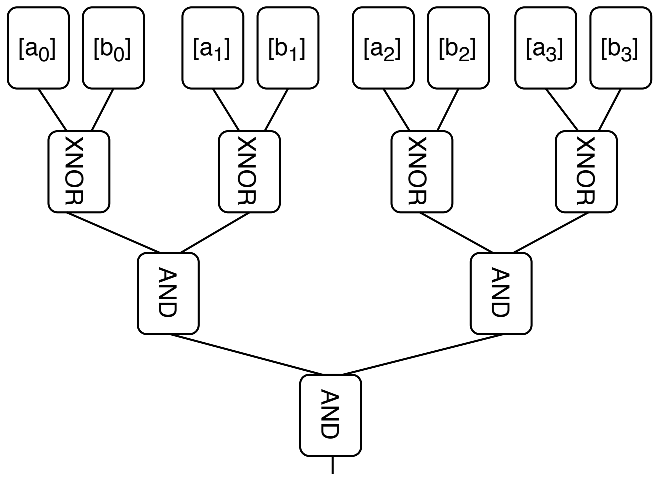


Fig. 2. Example circuit for testing equality of two 4-bit numbers

Due to the way noise builds up in most leveled-FHE schemes, we cannot always use standard textbook binary circuits for comparison. In order to minimize noise growth, we need to use binary circuits with a minimal depth. For the equality comparison this is straightforward, since we only need to XNOR the corresponding bits and compute the logical AND of all bits, as seen in Figure 2. Circuits to test for lesser/greater than are not as straightforward to implement, therefore we implement the comparison circuit proposed by Garay et al. [17]. The computational cost for computing the comparison circuits is shown in Table II.

B. Database Data Structures

Next to implementing constructions to represent values, we have also implemented several data structures to perform the filtering, grouping, and aggregation. Just like with the numeric constructions from the previous section, we also provide 'packed' versions of some data structures in the case that the cryptographic scheme that is used supports ciphertext packing.

For representation of table data in our aggregation solution, we have two main data structures. We have a 'row' data structure, and its packed ciphertext counterpart, the 'block' data structure. These data structures use a combination of integer and binary integer constructions to represent (a block of) rows in the table we aggregate over. We classify the columns of the aggregation table in two categories: aggregation columns which are represented with integer constructions, and filtering/grouping columns which are represented with binary constructions. As their name implies, we evaluate our aggregation functions over the aggregation columns, and we perform filtering/grouping on our filtering/grouping columns. In order to support both aggregation and filtering/grouping functionality for a column, we need to represent that column in our solution with both an integer construction based aggregation column, and a binary integer construction based filtering/grouping column.

Technically, it is possible to implement an addition/subtraction/multiplication circuit for the binary integer construction, which would allow us to use only one numerical construction in the case we want to enable both aggregation and filtering/grouping of a column in our database, and save some memory in the process. We chose not to implement this, since for a normal integer construction, evaluating addition, subtraction, and multiplication only takes one homomorphic operation, and evaluating an entire addition/subtraction/multiplication circuit for the binary integer construction slows down the aggregation too much for the memory we save, since we are optimizing for evaluation speed.

C. Mapping, Aggregation & Combination Functions

In the previous sections we saw how to store and represent data in our solution, however, we need to be able to evaluate functions homomorphically over our encrypted data. For this we define three different types of functions: mapping functions, aggregation functions, and combination functions. We use mapping functions to permute the input to the aggregation functions, aggregation functions to combine the entries of a column in the database to a single aggregation result, and combination functions to combine multiple aggregation results.

We evaluate the mapping functions for every row in the database. All mapping functions have the same structure, taking a row as input and returning an Integer Construction as output. Simple mapping functions return the Integer Construction of a single column, i.e. the mapping function of query `SUM(col 5)` returns just the Integer Construction of column 5. More complex mapping functions can also be specified, for example for query `SUM(col 5 * col 6)` the mapping function returns the Integer Construction corresponding to the multiplication of the Integer Construction of column 5 and column 6.

Like the mapping functions, aggregation functions in our solution all have the same structure. An aggregation function takes three arguments, a pointer to where the current encrypted aggregation result is stored, the output of the mapping function of the current row/block that is processed, and an encrypted bit construction indicating whether to include the current row/block in the aggregation or not.

The aggregation functions are so-called 'online' functions, meaning that the functions do not need the entire table/column as input, but process the data piece-by-piece. We choose to use online aggregation functions, since the encrypted rows can get rather large in memory size. With online functions we do not require the entire data set loaded in memory, allowing machines with less memory available than needed to store the entire table in memory to evaluate the secure aggregation functions.

Last, we discuss the combination functions in our solution. Combinations functions are used to manipulate the results of the aggregation function(s) after aggregation has been performed. The most notable use for the combination functions is to implement a variance/standard deviation calculation, which is calculated by combining the sum, count and sum of squares value of a column.

The mapping, aggregation, and combination functions are combined in the AggregationCombinationTarget data structure. The data structure specifies a mapping function and a set of aggregation functions the database should evaluate, and how to combine those aggregation results into the final result.

D. Query Construction and Flow

In the previous section we explained how we represent data in our solution, both on a low and on a high level, and we explained how our aggregation and combinations functions are structured and interact with each other. In this section we explain how we structure queries in our solution, and the general application flow that is executed when a query is evaluated.

1) Query Structure: A query in our solution consists of three parts, it has:

- One or more AggregationCombinationTargets
- Zero or more Filters
- Zero or more Groupings

The AggregationCombinationTargets specify for a row how the row is transformed into an Integer Construction with the mapping function, which aggregation functions need to be evaluated, how the results of these aggregation functions are combined into a final result, and whether the client should perform any actions after decryption. Some operations are difficult to do homomorphically, but easy to do on plaintext, such as division or taking a square root. Our solution provides pre-made AggregationCombinationTargets for calculating the SUM, COUNT, AVERAGE, SUM_OF_SQUARES, VARIANCE, and STANDARD DEVIATION, but also allows for creation of custom AggregationCombinationTargets.

The Filter specifies for a column how to filter the input to the aggregation functions. The Filter specifies the comparison function to use, e.g. equality (==), lesser than (<), lesser than or equal (<=), greater than (>), or greater than or equal (>=), together with an encrypted binary integer construction which value to compare the column values against.

The Grouping specifies which columns the database should group the aggregation on. When grouping on a column, our aggregation solution takes the domain of that column, and creates an aggregation 'bucket' for each value in that domain. In the case that there are 'gaps' in the domain (i.e. the domain is [1, ..., 4], but only values 1, 2, and 4 are used as grouping values) this method is not very efficient, since we also evaluate the aggregation result for when the grouping value is 3. We can solve this inefficiency by leaking some information in form of disclosing to the server exactly which values the grouping column can have[5], e.g. in the case of the previous example, specifying that the value '3' is not used when grouping.

2) Application Flow: After constructing a query as discussed in the previous section, the database evaluates the query. For ease of explanation, we first define two subroutines `compute_filter` and `compute_aggregation_combination`.

The `compute_filter` subroutine takes as input the encrypted database, and a list of Filters. The subroutine loops through the database, and based on the target column, condition, and target value stored in the Filter, the subroutine computes for each row an encrypted bit construction that indicates whether the current row matches the specified Filter. The `compute_filter` subroutine outputs the vector of encrypted bit constructions that, per row, indicate whether to include a value in aggregation or not. If the list of Filters provided to `compute_filter` is empty, the subroutine returns a vector of encrypted 'true/1' values.

The `compute_aggregation_combination` subroutine takes as input the encrypted database, a list of AggregationCombinationTargets, and a vector of encrypted bit construction values that indicate whether a row/block should be included in aggregation, i.e. the output of the `compute_filter` subroutine. The subroutine unpacks each AggregationCombinationTarget, and extracts the mapping functions, aggregation functions, and combination functions. For each AggregationCombinationTarget, the subroutine loops through the rows in the database, computing the mapping function on each row. For each output of the mapping function, the aggregation function is called with the current running aggregation result, the current mapping function output and the corresponding filtering bit. The aggregation function will conditioned on the filtering bit, using homomorphic multiplication, add the value or zero to the current result. After all aggregation functions have been evaluated for an AggregationCombinationTarget, the aggregation results are combined using the combination function. The output of the subroutine are the results of the combination function of each aggregation combination target.

[5]Note we do not disclose the value frequencies, only the value domain

We explain the application flow in two scenario's. In the first scenario we *do not* perform grouping, and in the second scenario we *do* perform grouping. In Figure 3 we provide an overview of the execution flow for both scenarios.

Scenario 1:

1. Compute the encrypted filtering based on the supplied Filter(s) with the `compute_filter` subroutine
2.a Compute the results using the `compute_aggregation_combination` subroutine
3.a Store the result
4.a Send back results to the query issuer.

Scenario 2:

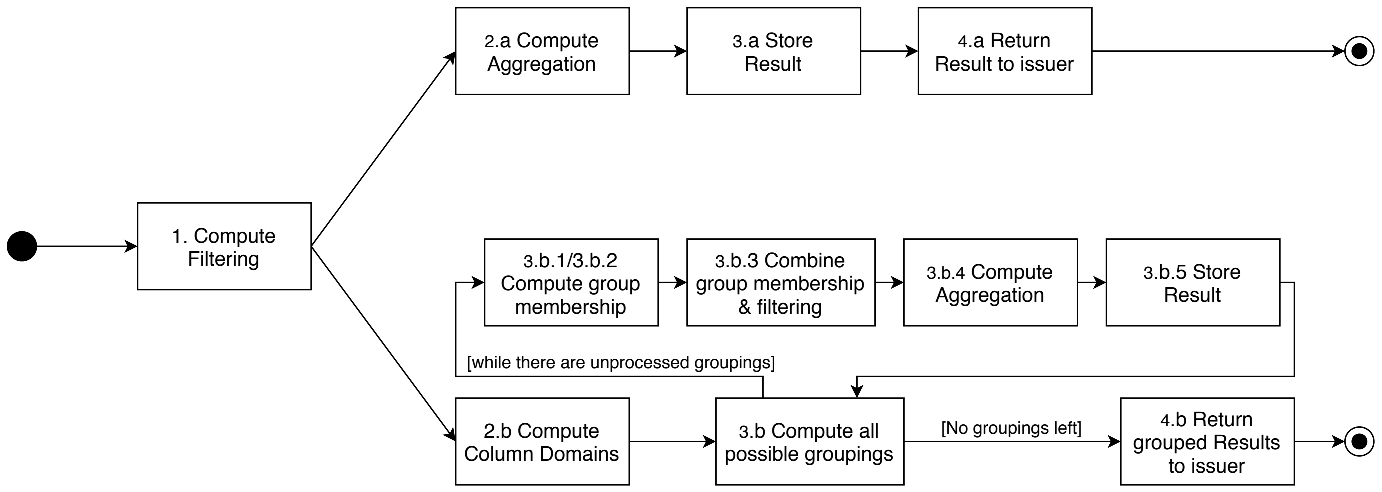
1. Compute the encrypted filtering based on the supplied Filter(s) with the `compute_filter` subroutine
2.b Compute the column domains of the columns we are grouping on
3.b Compute all possible combinations of groupings
3.b.1 Create Filters for the current grouping combination (e.g. Filter(col_8, ==, 1) $\wedge$ Filter(col_9, ==, 2))
3.b.2 Compute an encrypted filtering for the created Filters using the `compute_filter` subroutine.
3.b.3 Combine the filtering from step 1. with the filtering from step 3.b.2, by computing the AND (multiplication) of the bits of corresponding rows
3.b.4 Compute the results for the current grouping combination using the `compute_aggregation_combination` subroutine.
3.b.5 Store the results, and move on to the next grouping combination
4.b Send back all grouped results to the query issuer.

V. Results

A. Cryptographic Scheme

Before we can evaluate the performance of our aggregation solution, we need to select a cryptographic scheme to instantiate our solution with. The most prominent FHE libraries available at this moment in time are HElib which implements BGV and CKKS, Microsoft SEAL which implements BFV and CKKS, PALISADE which implements BFV, BGV, CKKS and FHEW, and FV-NFLlib which implements BFV.

The Microsoft SEAL, HElib and FV-NFLlib have been compared by Melchor et al [18]. Looking at the results they get, there is not a clear winner that performs better than all other libraries. Depending on the plaintext modulus and multiplicative depth, there are performance differences between the libraries, and sometimes even between different versions of the same library. However, next to (computational) performance there are also secondary factors we should keep in mind, such as whether the library/scheme only supports binary circuits, or also modular arithmetic circuits, and the documentation of the library.



2.a Compute Aggregation
3.a Store Result
4.a Return Result to issuer
1. Compute Filtering
3.b.1/3.b.2 Compute group membership
3.b.3 Combine group membership & filtering
3.b.4 Compute Aggregation
3.b.5 Store Result
[while there are unprocessed groupings]
2.b Compute Column Domains
3.b Compute all possible groupings
[No groupings left]
4.b Return grouped Results to issuer

Fig. 3. Flowchart describing application flow for submitted queries

Based on these secondary factors, the PALISADE library [19] seems like the most suitable FHE library available. The PALISADE library is open source, and was originally funded by DARPA PROCEED[6]. PALISADE has active development going on and a large amount of documentation available. The 'most developed' scheme in the PALISADE library is a Chinese remainder theorem variant of the BFV scheme by Halevi et al. [20], which we use to evaluate our solution. This Chinese remainder theorem variant of the BFV scheme is called 'BFVrns' in the library, so this is how we refer to it from now on.

B. Evaluation Scenario

In order to evaluate the secure aggregation solution we created, we first define a realistic scenario in which a secure aggregation solution is desirable. We believe benchmarking our solution in a realistic scenario is beneficial since it helps determine practical feasibility of our aggregation solution in the real world. The scenario will introduce several requirements, which are summarized in Table IV.

In our scenario we have a manufacturing company, 'Business Intelligence Incorporation' or 'BI Inc' for short, which contemplates moving their server infrastructure to the cloud. BI Inc keeps track of all the orders their clients have made, and creates analytics reports of this data overnight (I) in order to intelligently make decisions on issues such as inventory management and production planning. The information stored on the server infrastructure of BI Inc can give competitors insight in which clients BI Inc has, what kind of orders BI Inc is handling and at which price, and overall how profitable BI Inc is. Because of the value of this data, BI Inc requires following of NIST guidelines [21], and storing the data with at least 112 bits of security (II).

In order to model BI Inc for evaluation of our secure aggregation scheme, we look at the TPC-H database[7], and more specifically the LineItem table, since it captures the most essential data for aggregation. The TPC-H database was created as part of an industry-standard benchmark for decision support applications [22], and was created in such a way that it models any international manufacturing and/or distribution business.

The version of the LineItem table[8] we are aggregating over for BI Inc contains approximately 6 million rows (III)[9], and has 16 columns, two of which we use for aggregation, eleven of which we use for filtering/grouping, two we perform both aggregation and filtering/grouping for, and one which we don't model (the COMMENTS column) (IV). A full overview of the LineItem table can be found in Table III.

Next to database layout, the TPC-H design document also defines several queries to evaluate performance of a database. We chose to evaluate our database on queries Q1 and Q6, since they best fit our use case, aggregation, without requiring functionality like table joins or nested queries, which are not part of this research. The queries Q1 and Q6 are stated in Listing 3 and Listing 4. Queries Q1 and Q6 require us to group the aggregation on the 'returnflag' and 'linestatus' columns (V), to filter on the 'shipdate', discount', and 'quantity' columns (VI), and to evaluate a sum, average, and count calculation (VII).

C. Parameter Selection

With the requirements for BI Inc's secure aggregation solution, we can find an optimal parameter set for the scenario. We have already determined the needed security level, and based on the

[6]https://www.darpa.mil/program/programming-computation-on-encrypted-data

[7]http://www.tpc.org/tpch/

[8]The LineItem table can be generated with different parameters, for this research we take the default parameters

[9]We generated the table with the TPC-H dbgen tool, and a scaling factor of 1

TABLE III
TABLE LAYOUT LINEITEM TABLE

Column ID	Column Type	Representation
ORDERKEY	Grouping/Filtering	32 bits BIC
PARTKEY	Grouping/Filtering	32 bits BIC
SUPPKEY	Grouping/Filtering	16 bits BIC
LINENUMBER	Grouping/Filtering	4 bits BIC
QUANTITY	Aggregation	Integer Construct
QUANTITY	Grouping/Filtering	8 bits BIC
EXTENDEDPRICE	Aggregation	Integer Construct
DISCOUNT	Aggregation	Integer Construct
DISCOUNT	Grouping/Filtering	4 bits BIC
TAX	Aggregation	Integer Construct
RETURNFLAG	Grouping/Filtering	2 bits BIC
LINESTATUS	Grouping/Filtering	1 bit BIC
SHIPDATE	Grouping/Filtering	16 bits BIC
COMMITDATE	Grouping/Filtering	16 bits BIC
RECEIPTDATE	Grouping/Filtering	16 bits BIC
SHIPINSTRUCT	Grouping/Filtering	2 bits BIC
SHIPMODE	Grouping/Filtering	4 bits BIC

```
SELECT returnflag, linestatus,
    SUM(quantity), SUM(extendedprice),
    SUM(extendedprice * (1 - discount)),
    SUM(extendedprice * (1 - discount)
                       * (1 + tax)),
    AVG(quantity), AVG(extendedprice),
    AVG(discount), COUNT(*)
FROM lineitem
WHERE shipdate <= ('1998-12-01' - $DAY)
GROUP BY returnflag, linestatus
```

Listing 3. Query Q1 from TPC-H

table layout and the queries we need to support, we determine the minimum multiplicative depth and minimum plaintext space. From these three parameters, the PALISADE library can determine the other parameters, such as the ciphertext modulus q, ring dimension d and error distribution χ, which are needed for the Ring-LWE assumption from Definition 1.

For determining the multiplicative depth, we look at three different steps, computing the filtering/WHERE clause, computing the grouping/GROUP BY clause, and computing the aggregation itself.

For Q1, we filter on the 'shipdate' column. This requires evaluation of a less-than-or-equal circuit between two 16 bit Binary Integer Constructions. Referring to Table II, we see we need a depth of at least 9. Furthermore, we need to evaluate a group by on the 'returnflag' and the 'linestatus' columns. For this we need an multiplicative depth of at least 3, since the equality comparison for the 'returnflag' column requires a depth of 2, the equality comparison for the 'linestatus' column requires a depth of 1, and combining the two comparisons adds one layer of depth to the deepest depths of both columns. Combining the grouping and the filtering requires another layer of depth, thus we require a minimum depth of 10. For the aggregation itself, we look at the most expressive function, `SUM(extendedprice * (1 - discount) * (1 + tax))`. Evaluating the expression within the SUM requires a depth of 2, and applying the filter to the aggregation adds one multiplicative layer. Therefore, to evaluate Q1 we need at least a multiplicative depth of 11.

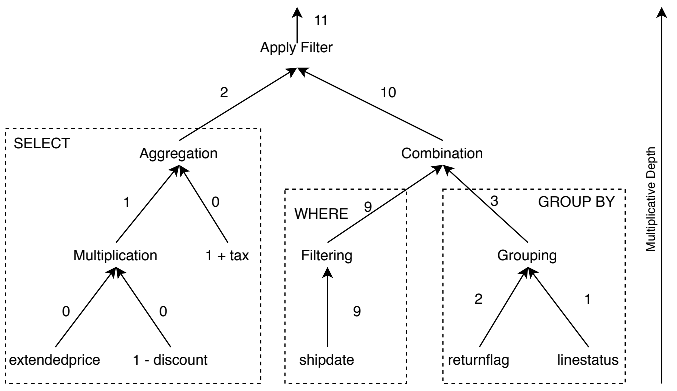


Fig. 4. Diagram describing how query depth of LineItem's Q1 is calculated

Similarly, we determine a minimum multiplicative depth of 12 for Q6. Therefore, the minimum depth required to evaluate both Q1 and Q6 with the same parameter set is 12. However, we set the minimum depth requirement for the parameter set to 13, since with a depth of 13, we allow for the freedom to evaluate all aggregation functions we specify in section IV-C.

In order to calculate the plaintext space for the aggregation, we take a more pragmatic approach. We evaluate Q1 and Q6 in a plaintext database, and determine the largest plaintext space needed for Q1 and Q6, respectively 58 bits and 47 bits. Due to

```
SELECT SUM(extendedprice * discount)
FROM lineitem
WHERE shipdate >= $DATE AND
    shipdate < $DATE + 1 year AND
    discount  > $DISCOUNT - 0.01 AND
    discount < $DISCOUNT + 0.01 AND
    quantity < $QUANTITY
```

Listing 4. Query Q6 from TPC-H

TABLE IV
PARAMETER REQUIREMENTS

	Requirement
I	Query evaluation time should take no longer than 8 hours (the query set will be evaluated overnight).
II	At least 112 bits of security
III	$\sim$ 6 million rows
IV	15 columns, 4 for aggregation, 13 for filtering/grouping
V	Grouping on the returnflag and linestatus column(s)
VI	Filtering on (a combination of) the shipdate, discount, and quantity column(s)
VII	Evaluation of at least the SUM, COUNT and AVG functions

a limitation in the PALISADE library, the plaintext modulus cannot exceed 32 bits. Therefore we create a plaintext space of 58 bits using the Chinese remainder theorem and two 29 bit co-prime plaintext moduli.

In conclusion, the optimal parameter set consists of a security level of 112 bits, a multiplicative depth of 13, and a plaintext space of 58 bits.

D. Evaluation Performance

For evaluation, we run our implementation on a server with four Intel Xeon E7-8890 v4 processors, and 2 TB of RAM. Each Xeon processor has 24 cores / 48 threads that run on 2.20 GHz, meaning that we have a total of 192 threads available.

For evaluation we run, next to Q1 (Listing 3) and Q6 (Listing 4), also a SUM, COUNT, and AVG query, grouped on the same columns as Q1. We run each query 20 times and report the mean and standard error to limit the influence of noise and other sources of randomness. The mean evaluation times and standard error we recorded are stated in Figure 5.

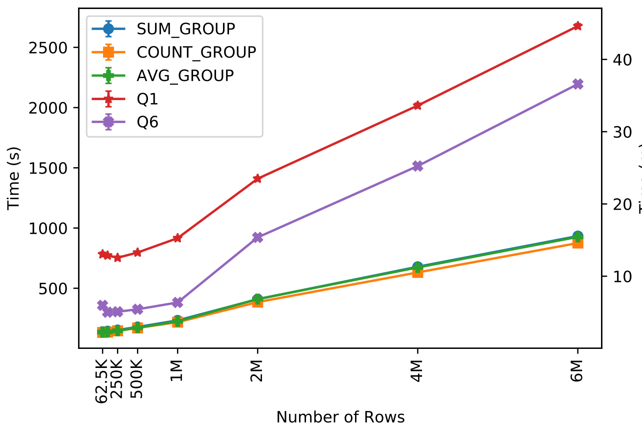


Fig. 5. Query evaluation time for different database sizes, for a security level of 112 bits, multiplicative depth of 13, and a plaintext space of 58 bits

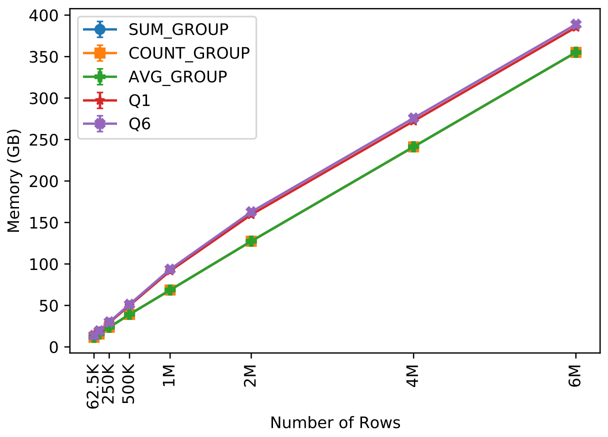


Fig. 6. Query evaluation memory usage for different database sizes, for a security level of 112 bits, multiplicative depth of 13, and a plaintext space of 58 bits

As shown in Figure 5, for the full 6,001,215 rows, Q1 completes on average in 44 minutes and 23 seconds, and Q6 completes on average in 36 minutes and 19 seconds. Based on the evaluation time restriction we set in Section V-B, we see that our solution easily evaluates queries Q1 and Q6 in the required time. Looking at Figure 5, we see slight exponential increase in aggregation time, which turns into a linear increase. This behaviour is explained by fact that for a lower number of rows, we cannot fully utilize the processors available to the system, resulting in a “slow startup”.

Next to evaluation time, we look at required RAM and storage space. For the required storage space we only concern ourselves with the size of the ciphertexts, and the amount of ciphertexts needed to represent the database, whereas for the required RAM we also need to determine the memory needed to store the cryptographic parameters, and the memory needed to store the intermediate results needed for the evaluation of a query. The static memory cost, i.e. the storage cost of the database, is $\sim$ 330.11 GB, a significant increase from the 1 GB of plaintext database we use as source data. However, in order to preserve flexibility in the queries we can evaluate, in the current implementation we encrypt every column with the leveled-FHE scheme. If we only encrypt the columns needed for evaluation of Q1 and Q6, the storage cost of the database decreases to approximately 75.21 GB.

The memory needed to evaluate our queries is graphed in Figure 6. From this graph we can see that the amount of RAM needed to evaluate our most expensive query, Q6, is $\sim$388.90 GB. Contrary to ‘slow start’ we saw for the evaluation time, the memory cost increases linearly.

E. Other Scenarios

In the previous sections we painted a realistic, “average case” scenario for our solution, which we use to evaluate the performance. However, we can imagine that in certain scenarios other requirements are needed. In this section we discuss the effect on storage space and evaluation time of changing parameters such as the security level, the multiplicative depth of the circuits that the homomorphic encryption scheme can evaluate, and the plaintext space available for aggregation operations.

Again we evaluate the performance of the different parameter sets on the LineItem table of the TPC-H database, with the SUM, COUNT, AVG, Q1 and Q6 queries. We compare with our previous results to put the changes of the parameters into context.

1) Security Level: One can imagine there are scenarios where we need more (or less) than 112 bits of security, such as health care applications where a higher level of security is desirable. On the other hand, one can also image scenarios where a lesser levels of security is acceptable, for example when dealing with data that only is valuable for a limited amount of time. Looking at the NIST Recommendation for Key Management [23], we see for data that needs to be secure after 2030, we need to have a security strength of 128, 192, or 256 bits. The recommendation also shows us that the previous recommended security strength was 80 bits, but is no longer adequate. However, some quick math[10] shows us that a current state of the art computer, operating with 128 threads at 4.3 GHz, which we for ease of computation translate this to 4.3 billion breaking operations / second, still needs $\approx 69,000$ years to break this level of security, meaning that it might be suitable for low- or temporarily-valuable data.

In Figure 7 and Figure 8 we see the performance and memory consumption of our aggregation system for a security level of 80, 112, 128, 192, and 256 bits. The multiplicative depth and plaintext space remain unchanged with a depth of 13 and a plaintext space of 58 bits. We see a similar runtime and memory consumption for the security levels 80, 112, 128, and 192 bits, and we see an increase in runtime and memory for the security level of 256 bits (except for Q6). The reason that we see similar performance for the lower security levels can be explained by the parameter generation of the PALISADE library. Based on the security level, multiplicative depth, and plaintext modulus, PALISADE determines a ring dimension and a ciphertext modulus which provide *at least* the needed security and plaintext space. However, due to restrictions on the ring dimension and ciphertext modulus to support ciphertext packing, these parameters increase in steps, instead of in a continuous way. We conclude that our default parameter set enjoys the benefits of 192 bit security instead of 112 bit security, at no extra cost. In the case of a security level of 256 bits, the ciphertext modulus has increased and ring dimension has doubled, meaning that we can store twice the amount of plaintext values per packed ciphertext, but use more storage overall. We see a decrease in performance for all queries, except for query Q6. This exception can be explained by the way we implemented grouping in our aggregation solution. We iterate through the database for every group. This means that in all queries except Q6 (since Q6 doesn't group), we iterate through the database six times, meaning that we need to move needed ciphertexts to the CPU six times. In the case of Q6 the performance gained by processing two times the number of plaintext values per ciphertext outweighs the performance lost by the increase in ciphertext size, while in the case of the other queries performance loss is greater than the performance gained by doubling the amount of plaintext values per ciphertext.

2) Multiplicative Depth: Just like one can imagine scenarios with different security needs, one can also imagine scenarios with different requirements for multiplicative depth. For example, we can imagine a scenario where no grouping and filtering is needed, but some multiplicative depth is needed to evaluate complex queries, such as a standard deviation or a (co-)variance computation. On the other hand, we can also imagine a scenario where we need a large multiplicative depth for filtering and grouping on multiple attributes with a large domain.

[10] $time_{years} = 2^{80}/(4,300,000,000 * 128)/(60 * 60 * 24 * 365.25) \approx 69,601$

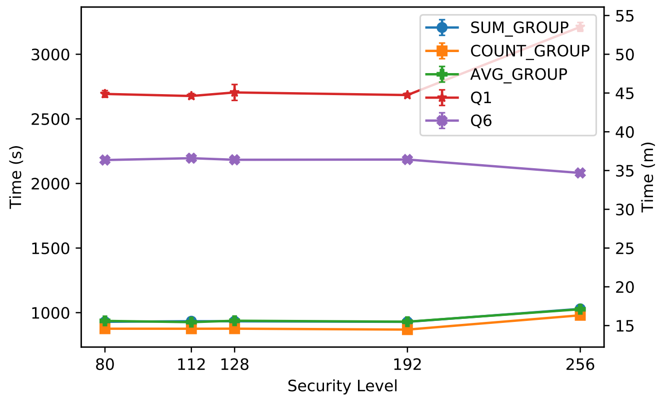


Fig. 7. Query evaluation time for different security levels, for a multiplicative depth of 13, and a plaintext space of 58 bits

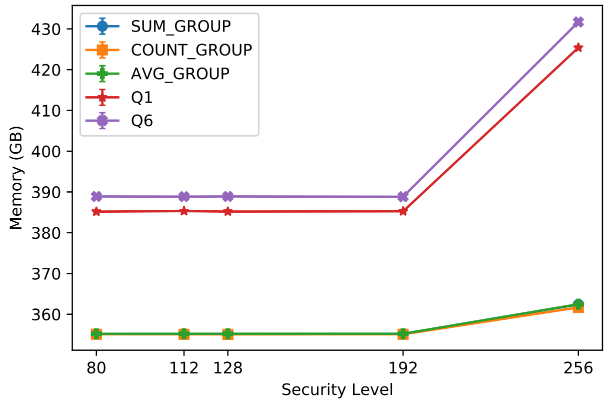


Fig. 8. Query evaluation memory usage for different security levels, for a multiplicative depth of 13, and a plaintext space of 58 bits

In Figure 9 and Figure 10 we see respectively the runtime performance and memory consumption of our aggregation system for a multiplicative depth of 5 up to 17, increasing with steps of 2. The security level and plaintext space remains unchanged with a level of 112 bits and a plaintext space of 58 bits. We see a linear increase in both aggregation time and memory consumption when the multiplicative depth is increased. We compare the aggregation time and memory consumption of Q1 and Q6 only for depths greater than respectively 11 and 13, since these queries cannot be executed with parameters with lesser multiplicative depth.

By default, the PALISADE library determines a ring dimension of 2^{14} for the configurations with depth 5 - depth 9, and a ring dimension of 2^{15} for the configurations with depth 11 - depth 17. During evaluation we encountered some instability with these parameters, as can be seen in Figure 13 in the appendix. Therefore we have manually changed the ring dimension for the lower depths to 2^{15} for the results in Figure 9 and Figure 10.

Table VI and Table VII in the appendix give a small overview of the possibilities these multiplicative depths have.

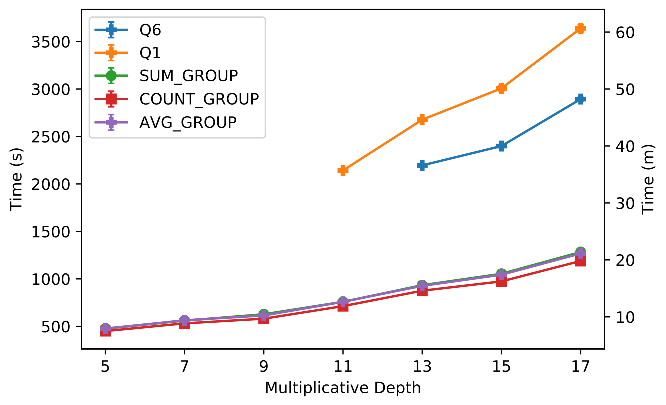


Fig. 9. Query evaluation time for different multiplicative depths, for a security level of 112 bits, and a plaintext space of 58 bits

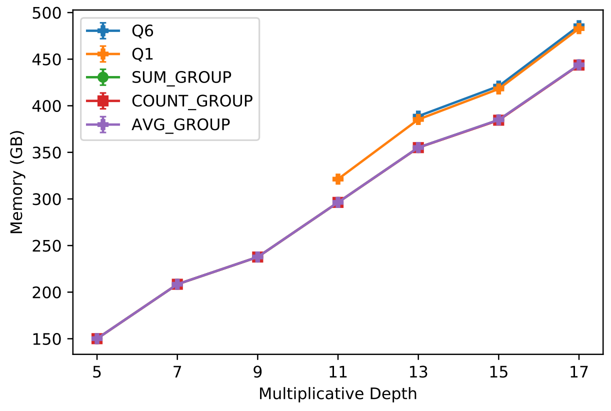


Fig. 10. Query evaluation memory requirements for different multiplicative depths, for a security level of 112 bits, and a plaintext space of 58 bits

3) Plaintext Space: The plaintext space is the final variable we want to discuss is the plaintext space. One can imagine scenario's where only 32 bits of plaintext space is needed, but also scenarios where smaller or larger plaintext space is needed, for example 24 bits or 64 bits. As we mentioned earlier, we achieve the larger plaintext spaces by using multiple ciphertexts with co-prime ciphertext moduli, and the Chinese remainder theorem. Empirical evaluation showed us that using the CRT to combine two plaintext moduli into one larger modulus, increased runtime and memory to be the sum of the two individual moduli. Therefore we will evaluate the runtime and memory consumption for several 'smaller' plaintext moduli, and in the case a larger combined modulus is desired the expected runtime and memory consumption can be determined by addition of the individual results.

In Figure 11 and Figure 12 we see the runtime performance and memory consumption of our aggregation system for a plaintext space of 16, 20, 24, 28, 29, and 32 bits. The security level and multiplicative depth remain unchanged with a level of 112 bits and a depth of 13. In both aggregation runtime and memory usage we see a similar, albeit strangely shaped increase. The shape is the result of the limited amount plaintext spaces we examine, and the characteristics of the BFVrns scheme. The BFVrns scheme uses the Chinese remainder theorem and multiple smaller primes that fit in system-native integers to increase runtime performance, opposed to a 'standard' BFV implementation that uses slower multiple precision integers. Looking at the internal parameters, we see that the 16-bit parameters use 9 CRT primes internally, the 20-bit and 24-bit parameters use 10 CRT primes internally, the 28-bit parameters use 11 CRT primes internally and the 29-bit and 32-bit parameters use 12 CRT primes internally. In the appendix in Figure 14 we plot the CRT primes needed for a plaintext modulus of every bit length in the range 16-32, and we see that this correlates directly with the shape of our graph. The fact that a 32-bit plaintext modulus is represented with the same amount of CRT primes as a 29-bit plaintext modulus, combined with the fact that the two parameter sets have the same ring dimension, means that we can achieve similar performance with a total plaintext space of 64 instead of 58 bits in our default scenario. However, this also means that we can achieve better performance than the standard parameter set, by combining a 28-bit and a 32-bit plaintext modulus to form a combined 60-bit plaintext modulus.

F. Comparison

We compare our solution against the SAGMA solution by Hackenjos, Hahn and Kerschbaum [7], since they claim less access pattern leakage than CryptDB [5] and better storage efficiency and filtering than [6]. We provide an overview of how our solutions compares against the other related work, including SAGMA, in Table V.

The SAGMA system uses the homomorphic BGN [24] scheme with optimizations by Hu et al [25]. The BGN scheme allows for an arbitrary number of additions, and one multiplication. This multiplication is used by SAGMA to facilitate grouping, as we saw in Section III. However, this means that SAGMA (without precomputation of e.g. the square of an value) can only evaluate additive aggregation functions. Since the BFVrns scheme we use is leveled-fully homomorphic, we can generate parameters such that we can evaluate any aggregation function.

Since SAGMA can only evaluate additive aggregation functions, we compare only the grouped SUM, COUNT, and AVG performance results from Section V-D with the SUM, COUNT, and AVG performance from Section 6.1 (Figure 5) in the SAGMA paper [7]. SAGMA allows evaluation of the AVG query by combining the SUM and COUNT query, thus the performance of the AVG query is the sum of the performance of the SUM and COUNT query. We acknowledge that the results from the SAGMA paper were obtained from a system with 16 cores / 32 threads (instead of 96 cores / 192 threads), thus we divide the evaluation time of the SAGMA solution by 6. The SAGMA paper only provides evaluation time for up to 10.000 rows of data, however, as the paper already indicates the increase is linear, meaning that we can extrapolate the runtime for a query on the approximately 6 million rows

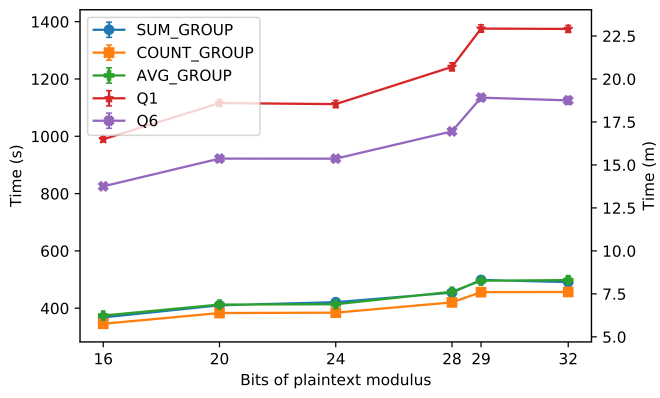


Fig. 11. Query evaluation time for different plaintext space sizes, for a security level of 112 bits, and a multiplicative depth of 13

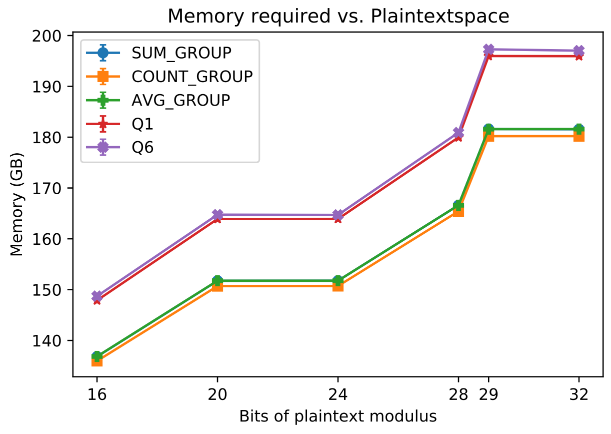


Fig. 12. Query evaluation memory requirement for different plaintext space sizes, for a security level of 112 bits, and a multiplicative depth of 13

of the TPC-H LineItem table. By extrapolation, we get that evaluation of the SUM, COUNT, and AVG queries on the full 6,001,215 rows of the LineItem table takes respectively 91.14, 29.42, and 120.56 minutes, whereas our runtimes for these queries are respectively 15.56, 14.59, and 15.44 minutes. However, as the SAGMA paper already emphasizes, SAGMA uses searchable symmetric encryption (SSE) to facilitate filtering and grouping, and therefore can evaluate their queries in sub-linear time when there are additional filtering queries present, whereas our solution always evaluates queries in linear time. This means that SAGMA performs equally well in the case that a WHERE clause reduces the amount of rows to aggregate over to respectively 2 million for the COUNT query, and 1 million for the SUM query.

The sublinear runtime enabled by use of SSE, however, also means that we trade in security (in the form of leaking the access pattern) for performance, which is undesirable for sensitive use cases. There are two different types of information leaked: information regarding the filtering clauses and information regarding the group values. Research by Islam et al. [10], and Cash et al. [11] shows us that the information leaked by the SSE scheme can be exploited to reconstruct sensitive filtering queries performed by the client, and information on (the frequency of) group values which can be exploited to reconstruct the grouping attributes and queries. Hackenjos et al. propose several techniques to reduce the information leakage. However, in the case of dynamic databases their proposed techniques either have limited effect, or disproportionally increase runtime (in the case of adding dummy records).

Compared to SAGMA, our solution does not need these complicated techniques which slow down aggregation to minimize the information leaked about the group values. Due to the use of the BFV scheme [13] with optimizations by Halevi et al. [20], we can compute group membership for each group value without leaking information on the access pattern, since the ciphertexts created by the BFV scheme (and created by evaluating the homomorphic functions) are IND-CPA secure, meaning that an adversary cannot distinguish whether the output of the group membership computation is true or false. Filter conditions are computed in a similar way, meaning that our solution also does not leak information on whether a row meets a filtering condition or not. However, while our solution leaks significantly less information when evaluating queries, it also requires linear time instead of sub-linear time w.r.t. the database size to evaluate queries.

VI. Conclusion and Future Work

In this paper we propose a dynamic secure aggregation system based on leveled fully homomorphic encryption that supports evaluation of complex aggregation queries, secure filtering, and secure grouping. Compared to previous solutions, our solution achieves better query evaluation time, can evaluate more complex queries, and supports filtering and grouping while better protecting the access pattern.

The protection against access pattern leakage does come at a cost however. Compared to previous works, our solution achieves an aggregation time that is linear in the database size, instead of sub-linear aggregation time as achieved by CryptDB [5] and SAGMA [7].

Nevertheless, our solution still achieves a notable speedup for aggregation in wall-clock time compared to e.g. SAGMA, due to an extensive parameter selection and the use of packed ciphertexts. We evaluate the effect of different parameter sets on aggregation time and memory consumption, and elaborate on the benefits of these different parameters, such as supported queries (query expressiveness and filtering/grouping capability), and achieved security level.

A. *Future Research*

To conclude our research, there are three research directions we would like to highlight which we believe deserve more attention. The first direction is the incorporation of the Chimera homomorphic encryption solution proposed by Boura et al [26]. Chimera is a hybrid solution that combines the BFV [13], CKKS [27], and TFHE [28] schemes for respectively integer

TABLE V
COMPARISON OF RELATED WORK:
CIRCLES ILLUSTRATE PROTECTION AGAINST ACCESS PATTERN LEAKAGE, WHERE AN EMPTY CIRCLE INDICATES LOW PROTECTION, AND A FULL CIRCLE INDICATES HIGH PROTECTION

Scheme	Aggregation Functions	Filtering	Grouping	Access Pattern Hiding *static*	Access Pattern Hiding *dynamic*	Multiple Attributes
CryptDB [5]	additive	Order Preserving Encryption	Deterministic Encryption	○	○	✓
Seabed (ASHE) [6]	additive	Order Preserving Encryption	Deterministic Encryption	○	○	×
Seabed (SPLASHE) [6]	additive	Order Preserving Encryption	SPLASHE	◔	◔	×
SAGMA [7]	additive	Searchable Encryption	Searchable Encryption	◐	◔	✓
This paper	additive & multiplicative	leveled-FHE	leveled-FHE	●	●	✓

arithmetic, floating point arithmetic and binary arithmetic. Chimera makes use of switching algorithms/bridges between the different schemes to process the data in utilize the schemes it is best suited for. We believe that using Chimera can significantly speed up the filtering and grouping computations of our solution.

The second direction is increasing the interactivity of our solution. We set as design goal to have a single-round aggregation solution, but we believe increasing interactivity can drastically speed up the grouping/filtering process. Increasing interactivity would allow us to have a separate scheme for binary arithmetic with plaintext modulus 2, eliminating the need for multiplications for computing the XOR gate, and having the query issuer decrypt the binary arithmetic results and send back an encrypted integer arithmetic result. This does increase interactivity quite a bit, since for a database with n rows, for every group the aggregator has to send n binary arithmetic ciphertexts to the query issuer, and the query issuer needs to send n integer arithmetic ciphertexts back to the aggregator.

The last research direction we want to highlight is the creation of a standardized database/aggregation benchmark for homomorphic encryption scheme's, similar to the standardized database benchmarks for plaintext database systems created by the TPC organization. Such a standardized benchmark not only allows for effective comparisons between homomorphic encryption schemes, but also gives insight in practical feasibility of homomorphic encryption schemes for use in encrypted databases.

ACKNOWLEDGMENT

REFERENCES

[1] M. Kaminska and M. Smihily, "Cloud computing - statistics on the use by enterprises," tech. rep., Eurostat, dec 2018.

[2] The European Commission, *European Data Market*. The European Commission, 2017.

[3] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*, pp. 44–55, IEEE, 2000.

[4] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra, "Executing SQL over encrypted data in the database-service-provider model," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data - SIGMOD '02*, ACM Press, 2002.

[5] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: Protecting confidentiality with encrypted query processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), p. 85–100, Association for Computing Machinery, 2011.

[6] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan, "Big data analytics over encrypted datasets with seabed," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, (USA), p. 587–602, USENIX Association, 2016.

[7] T. Hackenjos, F. Hahn, and F. Kerschbaum, "Sagma: Secure aggregation grouped by multiple attributes," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD, ACM, 2020.

[8] M. Naveed, S. Kamara, and C. V. Wright, "Inference attacks on property-preserving encrypted databases," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, (New York, NY, USA), p. 644–655, Association for Computing Machinery, 2015.

[9] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart, "Leakage-abuse attacks against order-revealing encryption," in *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 655–672, 2017.

[10] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: ramification, attack and mitigation.," in *Ndss*, vol. 20, p. 12, Citeseer, 2012.

[11] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*, ACM Press, 2015.

[12] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) lwe," in *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pp. 97–106, Oct 2011.

[13] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." Cryptology ePrint Archive, Report 2012/144, 2012. https://eprint.iacr.org/2012/144.

[14] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, (New York, NY, USA), p. 309–325, Association for Computing Machinery, 2012.

[15] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," in *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing - STOC '05*, ACM Press, 2005.

[16] V. Lyubashevsky, C. Peikert, and O. Regev, “On ideal lattices and learning with errors over rings.” Cryptology ePrint Archive, Report 2012/230, 2012. https://eprint.iacr.org/2012/230.

[17] J. Garay, B. Schoenmakers, and J. Villegas, “Practical and secure solutions for integer comparison,” in *Public Key Cryptography – PKC 2007* (T. Okamoto and X. Wang, eds.), (Berlin, Heidelberg), pp. 330–342, Springer Berlin Heidelberg, 2007.

[18] C. Aguilar Melchor, M.-O. Kilijian, C. Lefebvre, and T. Ricosset, “A comparison of the homomorphic encryption libraries helib, seal and fv-nfllib,” in *Innovative Security Solutions for Information Technology and Communications* (J.-L. Lanet and C. Toma, eds.), (Cham), pp. 425–442, Springer International Publishing, 2019.

[19] “PALISADE Lattice Cryptography Library (release 1.7.4).” https://palisade-crypto.org/, Jan. 2020.

[20] S. Halevi, Y. Polyakov, and V. Shoup, “An improved rns variant of the bfv homomorphic encryption scheme.” Cryptology ePrint Archive, Report 2018/117, 2018. https://eprint.iacr.org/2018/117.

[21] E. Barker and A. Roginsky, “Transitioning the use of cryptographic algorithms and key lengths,” tech. rep., Mar. 2019.

[22] M. Poess and C. Floyd, “New TPC benchmarks for decision support and web commerce,” *ACM SIGMOD Record*, vol. 29, pp. 64–71, Dec. 2000.

[23] E. Barker, “Recommendation for key management:,” tech. rep., May 2020.

[24] D. Boneh, E.-J. Goh, and K. Nissim, “Evaluating 2-dnf formulas on ciphertexts,” in *Theory of Cryptography* (J. Kilian, ed.), (Berlin, Heidelberg), pp. 325–341, Springer Berlin Heidelberg, 2005.

[25] Y. Hu, W. J. Martin, and B. Sunar, “Enhanced flexibility for homomorphic encryption schemes via crt,” in *Applied Cryptography and Network Security (ACNS)*, 2012.

[26] C. Boura, N. Gama, M. Georgieva, and D. Jetchev, “Chimera: Combining ring-lwe-based fully homomorphic encryption schemes.” Cryptology ePrint Archive, Report 2018/758, 2018. https://eprint.iacr.org/2018/758.

[27] J. H. Cheon, A. Kim, M. Kim, and Y. Song, “Homomorphic encryption for arithmetic of approximate numbers,” in *Advances in Cryptology – ASIACRYPT 2017* (T. Takagi and T. Peyrin, eds.), (Cham), pp. 409–437, Springer International Publishing, 2017.

[28] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, “Tfhe: Fast fully homomorphic encryption over the torus.” Cryptology ePrint Archive, Report 2018/421, 2018. https://eprint.iacr.org/2018/421.

Appendix

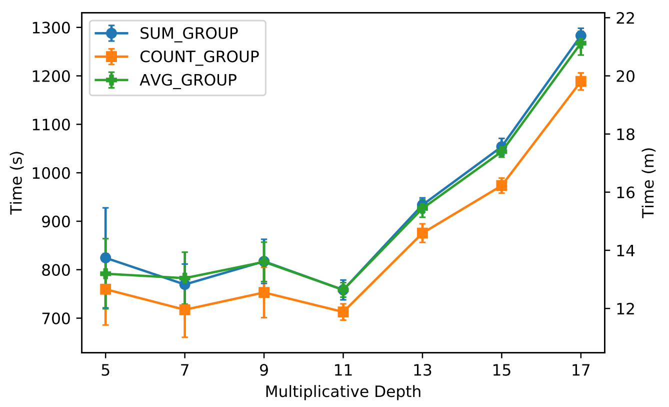


Fig. 13. Query evaluation time for different multiplicative depths, with instable results for depths 5 - 9

TABLE VI
QUERY FUNCTIONALITY ENABLED BY MULTIPLICATIVE DEPTH (PART 1)

Multiplicative Depth	Possibilities
3	• Evaluate our solution’s build-in functions SUM, COUNT, and AVG with a mapping function of depth 1 • Evaluate our solution’s build-in functions SUM-SQUARES with a mapping function of depth 0 AND • Evaluate equality filtering/grouping on two 1-bit columns, or one 2-bit column • Or, evaluate non-equality filtering one two 1-bit columns • Or, evaluate grouping on one 1-bit column *and* non-equality filtering on one 1-bit column • Evaluate our solution’s build-in functions VAR and STDEV without filtering/grouping
5	• Evaluate our solution’s build-in functions SUM, COUNT, and AVG with a mapping function of depth 3 • Evaluate our solution’s build-in function SUM-SQUARES with a mapping function of depth 2 • Evaluate our solution’s build-in functions VAR and STDEV with a mapping function of depth 1 AND • Evaluate equality filtering/grouping on four 2-bit columns, or one 8-bit column • Or, evaluate non-equality filtering on two two-bit columns, or one 2-bit column • Or, evaluate grouping on two 2-bit columns *and* non-equality filtering on one 2-bit column
7	• Evaluate our solution’s build-in functions SUM, COUNT, and AVG with a mapping function of depth 5 • Evaluate our solution’s build-in function SUM-SQUARES with a mapping function of depth 4 • Evaluate our solution’s build-in functions VAR and STDEV with a mapping function of depth 3 AND • Evaluate equality filtering/grouping on four 8-bit columns, or one 32-bit column • Or, evaluate non-equality filtering on eight two-bit columns, or one 4-bit column • Or, evaluate grouping on four 4-bit columns *and* non-equality filtering on four 2-bit columns
9	• Evaluate our solution’s build-in functions SUM, COUNT, and AVG with a mapping function of depth 7 • Evaluate our solution’s build-in function SUM-SQUARES with a mapping function of depth 6 • Evaluate our solution’s build-in functions VAR and STDEV with a mapping function of depth 5 AND • Evaluate equality filtering/grouping on four 32-bit columns, or one 128-bit column • Or, evaluate non-equality filtering on eight 4-bit columns, or one 8-bit column • Or, evaluate grouping on eight 8-bit columns *and* non-equality filtering on four 4-bit columns

TABLE VII
QUERY FUNCTIONALITY ENABLED BY MULTIPLICATIVE DEPTH (PART 2)

Multiplicative Depth	Possibilities
11	• Evaluate our solution's build-in functions SUM, COUNT, and AVG with a mapping function of depth 9 • Evaluate our solution's build-in function SUM-SQUARES with a mapping function of depth 8 • Evaluate our solution's build-in functions VAR and STDEV with a mapping function of depth AND • Evaluate equality filtering/grouping on eight 64-bit columns, or one 512-bit column • Or, evaluate non-equality filtering on eight 8-bit columns, or one 16 bit column • Or, evaluate grouping on eight 32-bit columns *and* non-equality filtering on four 8-bit columns
13	• Evaluate Q1 and Q6 of TPC-H benchmark • Evaluate our solution's build-in functions SUM, COUNT, and AVG with a mapping function of depth 11 • Evaluate our solution's build-in function SUM-SQUARES with a mapping function of depth 10 • Evaluate our solution's build-in functions VAR and STDEV with a mapping function of depth 9 AND • Evaluate equality filtering/grouping on 32 64-bit columns, or one 2048-bit column • Or, evaluate non-equality filtering on eight 16-bit columns, or one 32 bit column • Or, evaluate grouping on 16 64-bit columns *and* non-equality filtering on four 16-bit columns
15	• Evaluate our solution's build-in functions SUM, COUNT, and AVG with a mapping function of depth 13 • Evaluate our solution's build-in function SUM-SQUARES with a mapping function of depth 12 • Evaluate our solution's build-in functions VAR and STDEV with a mapping function of depth 11 AND • Evaluate equality filtering/grouping on 128 64-bit columns, or one 8192-bit column • Or, evaluate non-equality filtering on eight 32-bit columns, or one 64 bit column • Or, evaluate grouping on 64 64-bit columns *and* non-equality filtering on four 32-bit columns
17	• Evaluate our solution's build-in functions SUM, COUNT, and AVG with a mapping function of depth 15 • Evaluate our solution's build-in function SUM-SQUARES with a mapping function of depth 14 • Evaluate our solution's build-in functions VAR and STDEV with a mapping function of depth 13 AND • Evaluate equality filtering/grouping on 512 64-bit columns, or one 32768-bit column • Or, evaluate non-equality filtering on 32 32-bit columns, or one 128-bit column • Or, evaluate grouping on 256 64-bit columns *and* non-equality filtering on 16 32-bit columns

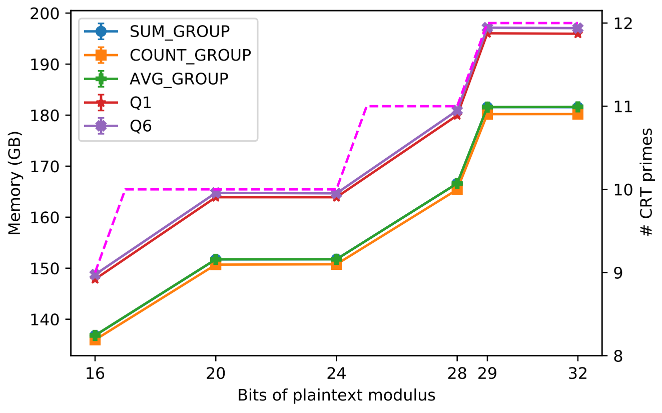


Fig. 14. Query evaluation memory usage for different plaintext space sizes, with an dotted overlay of the amount of CRT primes used per plaintext space size