Search-and-compute on Encrypted Data

Jung Hee Cheon[1], Miran Kim[1], and Myungsun Kim[2]

[1] Department of Mathematical Sciences, Seoul National University
{jhcheon,alfks500}@snu.ac.kr
[2] Department of Information Security, The University of Suwon
msunkim@suwon.ac.kr

Abstract. Private query processing on encrypted databases allows users to obtain data from encrypted databases in such a way that the user's sensitive data will be protected from exposure. Given an encrypted database, the users typically submit queries similar to the following examples:

- How many employees in an organization make over $100,000?
- What is the average age of factory workers suffering from leukemia?

Answering the above questions requires one to **search** and then **compute** over the encrypted databases *in sequence*. In the case of privately processing queries with only one of these operations, many efficient solutions have been developed using a special-purpose encryption scheme (e.g., searchable encryption). In this paper, we are interested in efficiently processing queries that need to perform both operations on *fully* encrypted databases. One immediate solution is to use several special-purpose encryption schemes at the same time, but this approach is associated with a high computational cost for maintaining multiple encryption contexts. The other solution is to use a privacy homomorphism (or fully homomorphic encryption) scheme. However, no secure solutions have been developed that meet the efficiency requirements.

In this work, we construct a unified framework so as to efficiently and privately process queries with "search" and "compute" operations. To this end, the first part of our work involves devising some underlying circuits as primitives for queries on encrypted data. Second, we apply two optimization techniques to improve the efficiency of the circuit primitives. One technique is to exploit SIMD techniques to accelerate their basic operations. In contrast to general SIMD approaches, our SIMD implementation can be applied even when one basic operation is executed. The other technique is to take a large integer ring (*e.g.*, $\mathbb{Z}_{2^t}$) as a message space instead of a binary field. Even for an integer of k bits with $k > t$, addition can be performed with degree 1 circuits with lazy carry operations. As a result, search queries including a conjunctive or disjunctive query on encrypted databases of N tuples with μ-bit attributes require $\mathcal{O}(N \log \mu)$ homomorphic operations with depth $\mathcal{O}(\log \mu)$ circuits. Search-and-compute queries, such as a conjunctive query with aggregate functions in the same conditions, are processed using $\mathcal{O}(\mu N)$ homomorphic operations with at most depth $\mathcal{O}(\log \mu \log N)$ circuits. Further, we can process search-and-compute queries using only $\mathcal{O}(N \log \mu)$ homomorphic operations with depth $\mathcal{O}(\log \mu)$ circuits, even in the large domain. Finally, we present various experiments by varying the parameters, such as the query type and the number of tuples.

Keywords: encrypted databases, private query processing, homomorphic encryption.

1 Introduction

Privacy homomorphism is an important notion for encrypting clear data while allowing one to carry out operations on encrypted data without decryption. The concept was first introduced by Rivest et al. [29], and much later, Feigenbaum and Merritt's question [16] affirmed the concept: Is there an encryption function $\mathsf{E}(\cdot)$ such that both $\mathsf{E}(x+y)$ and $\mathsf{E}(x \cdot y)$ are easy to compute from $\mathsf{E}(x)$ and $\mathsf{E}(y)$? Since then, there have been a substantial number of studies concerned with solving this problem. However, there had been very little progress made in determining whether such efficient and secure encryption schemes exist until 2009, when Craig Gentry demonstrated the possibility of constructing such an encryption scheme [18].

Roughly speaking, Gentry's scheme allows anyone to compute $\mathsf{E}(f(x_1, \ldots, x_N))$ from a collection of encrypted data $\mathsf{E}(x_1), \ldots, \mathsf{E}(x_N)$ for any computable function f without knowing the actual data. He called this technique a fully homomorphic encryption (FHE) scheme.

While the use of Gentry's scheme and other FHE schemes (e.g., [35,8,7]) allows us to securely evaluate any function in a theoretical sense, the evaluation cost is still far from being practical for many functions. Moreover, the complexity for several important functions has not been determined. Among the important functions, we restrict our interest to a set of functions for databases, which raises the following question: *Given a set of fully encrypted databases, can we construct a set of efficient functions to process queries over the encrypted databases*? If so, *what is the computational cost of the functions*?

Although this question is the starting point of this work, to facilitate a better understanding of the approach, we describe the motivation for our work from a different perspective. Ultimately, the two perspectives share a common outcome, *i.e.*, fully homomorphic encryption. Currently, perhaps the simplest way to search for records satisfying a particular condition over encrypted databases is via searchable encryption (*e.g.*, [34,4,3]). However, privately processing `sum` and `avg` aggregation queries in the same condition is performed using homomorphic encryption (*e.g.*, [15,27] and [6]). Thus, the private processing of a query that includes both matching conditions and aggregate operations requires the use of two distinct encryption techniques in parallel, *i.e.*, searchable encryption and homomorphic encryption. We should note that simultaneously maintaining multiple cryptosystems is fairly expensive.

This observation leads to the natural question: *Can we construct a solution to efficiently address such a database query without maintaining multiple contexts of encryption*?

At first glance, FHE schemes seem to perfectly fit the problem of processing queries on encrypted databases with a single encryption context. However, further research on this topic reveals that there exists no solutions for *expressing* and *processing* various queries on *fully* encrypted databases in an *efficient* way.

1.1 Our Results

Our main results are as follows:

- **A unified framework for private query processing**: We provide a common platform so that database users may work on a *single underlying cryptosystem,* represent their query as a function in a conceptually simpler manner, and efficiently carry out the function on fully encrypted databases.
- **Optimizing circuits and their applications to compact expressions of queries**: The foundation of our simple framework is a set of optimized circuits: equality, greater-than comparison and integer addition. We call these *circuit primitives*. Our optimizations of circuit primitives have been taken in such a way as to minimize the circuit depth and the number of homomorphic operations. To do this, we make extensive use of single-instruction-multiple-data (SIMD) techniques to move data across plaintext slots. An automorphism operation on ciphertexts allows such a movement in the plaintext slots without additional cost. In particular, our SIMD implementations have an important difference compared with a general SIMD strategy. In general, SIMD technology allows for basic operations to be performed on several data elements in parallel. However, this does not make the basic operations themselves run faster. On the contrary, our proposal works on packed ciphertexts of several data elements and thus enables one to improve the efficiency of the basic op-

erations of circuit primitives. Furthermore, we find that all circuit primitives have $\mathcal{O}(\log \mu)$ depth for μ-bit data.
We then express more complicated queries by a composition of the optimized circuit primitives. The resulting query functions are conceptually simpler than other representations of database queries and are compact in the sense that retrieval queries require at most $\mathcal{O}(\log \mu \log N)$ depth. Here, N means the number of tuples of μ-bit attributes in a table.

– **Further improvements in the performance of query processing**: FHE schemes usually use $\mathbb{Z}_2$ as a message space so that their encryption algorithm encodes an input message into a bit string and then encrypts each bit into a ciphertext. While our circuit primitives efficiently work on bit encryptions, we can achieve further improvements by adopting a large integer ring (*e.g.*, $\mathbb{Z}_{2^t}$), especially in the case of *computing* on encrypted numeric-data. Even for an integer of k bits with $k > t$, addition can be performed with degree 1 circuits by processing lazy carry operations. Although this rectification requires us to amend our circuit primitives, we can again preserve their optimality by SIMD operations. In other words, search-and-compute queries can be processed with only $\mathcal{O}(\log \mu)$-depth circuits.

– **Comprehensive experiments**: We conduct comprehensive experiments for evaluating the performance of various queries expressed by our techniques from a theoretical as well as practical perspective. We first analyze the computational complexity of each type of query. For example, a function representing a conjunctive search query with τ matching conditions over N-block encrypted databases has $\mathcal{O}(\tau N \log \mu)$ computation complexity. Then, we implement these queries using Halevi-Shoup's library and Shoup's NTL library to verify the correctness and performance of the query.

1.2 A High-level Overview of Our Approach

Figure 1 graphically illustrates the high-level architecture of our approach.

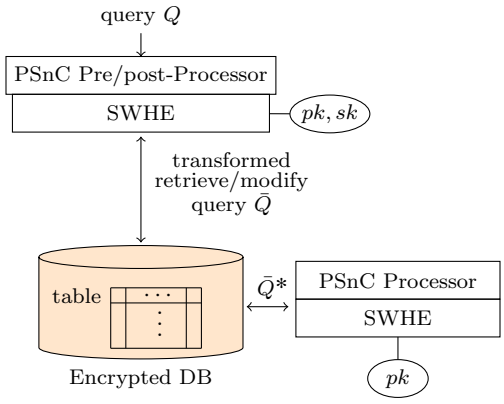

Fig. 1: Our PSnC Framework

Assuming a database consisting of N blocks, *i.e.*, $R_1 \parallel R_2 \parallel \cdots \parallel R_N$, to encrypt the record R_i, a DB user prepares a pair of public/private keys (pk, sk) for an FHE scheme and publishes the public

key to a DB server. The DB users store their encrypted records $\bar{R}_i = \mathsf{E}_{pk}(R_i)$ for $1 \leqslant i \leqslant N$ in the same way as normal write queries (*e.g.*, using the `insert-into` statement). We use an efficient variant of an FHE scheme: a somewhat homomorphic encryption (SWHE) scheme (we will discuss the SWHE scheme later).

Suppose that the user wants to submit a retrieval query Q to the DB server. Before being submitted, the query Q needs to be properly pre-processed so that all clear messages, such as constant values, are encrypted under the public key pk. We denote this transformed query by $\bar{Q}$.

Upon receiving $\bar{Q}$, the DB server compiles it into $\bar{Q}^*$ by applying our techniques. The readers can consider a dedicated module for performing this task.[3] Hereafter, we call the module a *Private Search-and-compute* (PSnC) processor. Next, the DB server homomorphically evaluates $\bar{Q}^*$ over the fully encrypted databases and returns the resulting ciphertexts to the user.

The DB user can decrypt the output using his private key sk while learning no additional data except for the records satisfying the `where` conditions.

For example, consider the following typical retrieval query Q_u:

$$\begin{array}{l} \texttt{select}\ A_{j_1}, A_{j_2}, A_{j_3} \\ \quad \texttt{from}\ \mathsf{R} \\ \texttt{where}\ A_{j_0} = \alpha; \end{array} \qquad (Q_u)$$

The PSnC preprocessor outputs $\bar{Q}_u$ with $\bar{\alpha} = \mathsf{E}_{pk}(\alpha)$ in place of α. The DB server transforms $\bar{Q}_u$ into $\bar{Q}_u^*$ by invoking its PSnC processor, where $\bar{Q}_u^*$ is written in the following form:

$$\texttt{equal}\left(\bar{A}_{j_0}^{(i)}, \bar{\alpha}\right) \cdot \left(\bar{A}_{j_1}^{(i)}, \bar{A}_{j_2}^{(i)}, \bar{A}_{j_3}^{(i)}\right) \qquad (\bar{Q}_u^*)$$

where $\texttt{equal}(\bar{x}, \bar{y})$ is one of our circuit primitives that outputs an encryption of 1 if $x = y$; otherwise, an encryption of 0; "$\cdot$" means homomorphic multiplication of ciphertexts, and $\bar{A}_j^{(i)}$ denotes an encryption of the j-th attribute value of the i-th tuple in the table. In this way, the DB server can process other involved queries without knowing the data.

1.3 Closely Related Work

A few results closely related to our work can be found in the literature. First, Lauter et al. in [24] showed how to privately compute `avg` and `var` functions using a variant of Brakerski et al.'s SWHE scheme [9]. They encode an integer as a polynomial with binary coefficients over $\mathbb{Z}_t[X]/\langle X^n + 1\rangle$ for sufficiently large t. Rather than directly computing the variables of the encrypted data, they compute the sum and the sum of squares of the ciphertexts and obtain the result with one division operation after decryption. However, their work only focused on applying homomorphic encryption to compute aggregate functions in query statements. Thus, it is not clear how to address their `where` clauses in a private manner.

Recently, Boneh et al. [5] proposed a way to privately process the `where` clause in a `select` statement and produce a set of matching indices. Their solution allows the `where` clause to involve conjunctive and disjunctive conditions. Their technique uses private set intersection together with

[3] Alternatively, one may imagine that $\bar{Q}^*$ transformed by the DB user directly is sent to the DB server. However, considering optimization and performance, we believe that the better choice involves the module becoming part of the DBMS.

homomorphic encryption. Their basic idea is as follows. Suppose that a DB server holds a database consisting of N tuples with d attributes, *i.e.*, $\left\{A_1^{(1)}, \ldots, A_d^{(N)}\right\}$. Let us denote v_{ij} as the attribute value of $A_j^{(i)}$ in the i-th tuple. The DB server considers the DB as a multivariate polynomial $D(x, y)$ satisfying $D(i, j) = v_{ij}$ for each i, j. For a `where` clause having $A_j = \alpha_j$, a DB user prepares a query polynomial $Q(y)$ such that $Q(j) = \alpha_j$ for all $1 \leqslant j \leqslant s$, where α_j is a constant value. When the DB user submits its encrypted polynomial $\bar{Q}(y)$, the DB server computes $R(x, y) = D(x, y) - \bar{Q}(y)$, obtains $R_j(x) = R(x, j)$ for each $1 \leqslant j \leqslant s$ by evaluating $R(x, y)$ at j, and sends all $R_j(x)$ to the DB user after randomizing the polynomial. Upon receiving $\{R_1(x), \ldots, R_s(x)\}$, the DB user decrypts and evaluates each polynomial $R_j(x)$ at each tuple. If $R_j(i) = 0$ for all j, the tuple index i implies that the i-th tuple is what the user wants.

However, their scheme has the following drawbacks: (1) their scheme only allows conjunctive and disjunctive conditions. Namely, the DB user cannot make a query that contains general comparison operators, such as $<, \leqslant, >$, and $\geqslant$; (2) the equality test is restricted to comparisons with a constant value; and (3) the DB users must revisit the DB server to obtain a list of real tuples because they only know the indices of those tuples. In conclusion, the query type supposed by their scheme is fairly constrained.

Our work differs in several ways from prior efforts. First, our solution can privately process the `select` clause and the `where` clause all at once. Second, our solution supports a wide range of query types–from simple search queries to join queries. In particular, our solution allows the DB users to express rich conditions, including $<, \leqslant, >, \geqslant$, and $<>$. More detailed survey results are provided in Section 7.

The remainder of the paper is structured as follows. In Section 2, we briefly review the BGV-type homomorphic encryption scheme. In Section 3, we construct the optimized circuits for expressing queries. Then, in Section 4, we show how to construct database queries having search and/or compute operations using our circuit primitives. Section 5 presents our optimization techniques for further improvements in performance, and Section 6 shows the experimental evaluations of our constructions.

2 Preliminaries

In this section, we recall the concept of homomorphic encryption and focus on describing the BGV-type cryptosystem [7,20], which is our underlying encryption scheme. In what follows, we give a description of the security model that our constructions assume.

2.1 Homomorphic Encryption (HE)

A homomorphic encryption (HE) scheme $\mathsf{HE} = (\mathsf{Kg}, \mathsf{E}, \mathsf{D}, \mathsf{Ev})$ is a quadruple of probabilistic polynomial-time algorithms that proceeds as follows:

- $(pk, ek, sk) \leftarrow \mathsf{Kg}(1^\kappa)$: The algorithm takes as input the security parameter κ and outputs a public encryption key pk, a public evaluation key ek and a secret decryption key sk.
- $c \leftarrow \mathsf{E}_{pk}(x; \gamma)$: The algorithm takes the public key pk, a single-bit message $x \in \{0, 1\}$ and a randomizer γ and outputs a ciphertext c. When no confusion may arise, we occasionally omit the randomizer γ.
- $x \leftarrow \mathsf{D}_{sk}(c)$: The algorithm takes the secret key sk and a ciphertext c and outputs a message $x \in \{0, 1\}$.

– $c_f \leftarrow \mathsf{Ev}_{ek}(f; c_1, \ldots, c_k)$: The algorithm takes the evaluation key ek, a function $f : \{0,1\}^k \to \{0,1\}$ and a set of k ciphertexts $c_1, \ldots, c_k$ and outputs a ciphertext c_f.

We assume that f will be represented by an arithmetic circuit over $\mathbb{Z}_2$, with the addition and multiplication gates.

We say that an HE scheme $\mathsf{HE} = (\mathsf{Kg}, \mathsf{E}, \mathsf{D}, \mathsf{Ev})$ is *homomorphic* if for any set of inputs $(x_1, \ldots, x_k)$, for every positive polynomial p and for all sufficiently large κ, it holds that

$$\Pr\left[\mathsf{D}_{sk}\left(\mathsf{Ev}_{ek}(f; c_1, \ldots, c_k)\right) \neq f(x_1, \ldots, x_k)\right] = \frac{1}{p(\kappa)}$$

where $c_i = \mathsf{E}_{pk}(x_i)$ for $1 \leqslant i \leqslant k$.

Although homomorphic cryptosystems (*e.g.*, [18,35,9,7,25]) allow us to homomorphically evaluate any arithmetic circuit without decryption, the noise of the resulting ciphertext grows during homomorphic evaluation, slightly with addition but substantially with multiplication. Two techniques are used for noise management: One is bootstrapping, which makes the ciphertext noise free by evaluating the decryption circuit homomorphically by the decryption key. The other is modulus switching, which scales down a ciphertext during every multiplication operation and reduces the noise by its scaling factor. The bootstrapping technique allows us to utilize an FHE scheme [19,10] at the cost of a significant degradation in performance. By contrast, Although a modulus-switching technique supports limited operations, this is more efficient for supporting low-degree homomorphic computations on encrypted data. This is called somewhat homomorphic encryption (SWHE).

The BGV SWHE Scheme Our solutions are implemented with an efficient variant of the Brakerski-Gentry-Vaikuntanathan (BGV) cryptosystem using a modulus-switching technique. We first review an SWHE version of a BGV FHE scheme [7] based on ring learning with errors (RLWE) problems [20].

Let us denote the reduction of the integer x modulo q into the interval $(-q/2, q/2] \cap \mathbb{Z}$ by $[x]_q$. For a security parameter κ, we choose an $m \in \mathbb{Z}$ that defines the m-th cyclotomic polynomial $\Phi_m(X)$. For a polynomial ring $\mathbb{A} = \mathbb{Z}[X]/\langle \Phi_m(X) \rangle$, we set the message space to $\mathbb{A}_t := \mathbb{A}/t\mathbb{A}$ for some fixed $t \geqslant 2$ and the ciphertext space to $\mathbb{A}_q := \mathbb{A}/q\mathbb{A}$ for an integer q. Then, all the ciphertexts are treated as vectors of elements in $\mathbb{A}_q$.

In this scheme, homomorphic addition is performed by simple component-wise addition of the ciphertexts, and homomorphic multiplication is performed using a tensor product over $\mathbb{A}_q$. One could use a key-switching technique to convert the product ciphertext into the refreshed ciphertext. Moreover, one chooses a chain of moduli $q_0 < q_1 < \cdots < q_L = q$ whereby the SWHE scheme can evaluate a depth-L arithmetic circuit. Modulus switching down during homomorphic computation implies that when we reach the smallest modulus q_0, we can no longer compute on ciphertexts. Here, the depth of an arithmetic circuit under an SWHE scheme means the number of reduced moduli in the circuit being evaluated homomorphically.

To formally describe their scheme, we need to introduce extra notation for denoting distributions as follows:

– $\mathcal{U}_q$: This is the uniform distribution over $(\mathbb{Z}/q\mathbb{Z})^{\phi(m)}$.
– $\mathsf{d}\mathcal{G}_q(\sigma^2)$: This is the discrete Gaussian that draws a real $\phi(m)$-vector according to the normal distribution $\mathcal{N}(0, \sigma^2)^{\phi(m)}$, rounds the real vector to the nearest integer vector, and produces the integer-vector-reduced modulo q.

- $\mathcal{T}(\rho)$: For a real value $\rho \in [0,1]$, this distribution outputs a vector in $\{0,\pm 1\}^{\phi(m)}$ with probability $\rho/2$ for each $+1$ and -1 and probability $1-\rho$ for each 0.
- $\mathcal{H}(h)$: For an integer $h \leqslant \phi(m)$, this distribution uniformly outputs a vector at random from $\{0,\pm 1\}^{\phi(m)}$ such that the number of non-zero entries is h.

Finally, we denote by $a \twoheadleftarrow \mathcal{D}$, choosing $a \in \mathbb{A}$ according to the distribution $\mathcal{D}$. Considering these notations, we describe an RLWE-based SWHE scheme of the BGV cryptosystem.

- $(a, b; \mathsf{s}) \leftarrow \mathsf{Kg}(1^\kappa, h, \sigma, q_L)$: The algorithm Kg chooses $\mathsf{s} \twoheadleftarrow \mathcal{H}(h), a \twoheadleftarrow \mathcal{U}_{q_L}$ and $e \twoheadleftarrow \mathsf{d}\mathcal{G}_{q_L}(\sigma^2)$. It then calculates $b = [a \cdot \mathsf{s} + 2e]_{q_L}$ and sets the secret key $sk = \mathsf{s}$ and the public key $pk = (a, b)$.
- $\mathbf{c} \leftarrow \mathsf{E}_{pk}(x)$: To encrypt a message $x \in \{0,1\}$, the algorithm chooses $v \twoheadleftarrow \mathcal{T}(1/2)$ and $(e_0, e_1) \twoheadleftarrow \mathsf{d}\mathcal{G}_{q_L}(\sigma^2)$ and outputs the ciphertext $\mathbf{c} = (c_0, c_1)$ by computing

$$(c_0, c_1) = (x, 0) + (bv + 2e_0, av + 2e_1) \bmod q_L.$$

  In practice, we use $\mathbf{c} = ((c_0, c_1), t, \eta)$ to represent a normal ciphertext with its level t and the noise magnitude η. Note that the initial ciphertext for a message is in level L.
- $x \leftarrow \mathsf{D}_{sk}(\mathbf{c})$: Given a ciphertext $\mathbf{c} = ((c_0, c_1), t, \eta)$, the algorithm outputs $x = [c_0 - \mathsf{s} \cdot c_1]_{q_t}$ as a corresponding plaintext.
- $\mathbf{c}_f \leftarrow \mathsf{Ev}_{ek}(f; \mathbf{c}, \mathbf{c}')$. The algorithm first determines whether $\mathbf{c}$ and $\mathbf{c}'$ have the same level. If not, it brings both ciphertexts to the same level by reducing the larger one modulo the smaller of the two moduli, e.g., $\tilde{t}$.
  Specifically, assume $\mathbf{c} = ((c_0, c_1), t, \eta)$ and $\mathbf{c}' = ((c'_0, c'_1), t', \eta')$ are the two ciphertexts of the messages x and x', respectively. Then, if the function f is an addition over ciphertexts, the algorithm outputs

$$\mathbf{c}_+ = (([c_0 + c'_0]_{q_{\tilde{t}}}, [c_1 + c'_1]_{q_{\tilde{t}}}), \tilde{t}, \eta + \eta'),$$

  which is considered to be an encryption of $x + x'$. If f is a multiplication over ciphertexts, it outputs

$$\mathbf{c}_\times = ([c_0 c'_0]_{q_{\tilde{t}}}, [c_0 c'_1 + c_1 c'_0,]_{q_{\tilde{t}}}, [c_1 c'_1]_{q_{\tilde{t}}}), \tilde{t}, \eta \cdot \eta').$$

Another very useful homomorphic operation allowed in the BGV-typed cryptosystem is the automorphism $X \mapsto X^g$ for some $g \in \mathbb{Z}_m^*$ because we can perform a cyclic rotation on the plaintext slots, in particular, without increasing the noise of the ciphertexts.

SIMD Technique In general, FHE (and SWHE) schemes encrypt small plaintexts (*e.g.*, $\mathbb{Z}_2$) into large ciphertexts (*e.g.*, $\mathbb{Z}_q$ for $q \gg 2$). Thus, provided that a ciphertext is able to contain a number of independent plaintexts, we can use memory space far more efficiently. Smart and Vercauteren [32] first mentioned that choosing appropriate parameters in some FHE schemes enables the FHE schemes to support SIMD operations on finite fields of characteristic two. Their key observation was that the plaintext space $\mathbb{A}_2$ can be considered as a vector of plaintext slots by the polynomial CRT (Chinese remainder theorem). Then, addition and multiplication in $\mathbb{A}_2$ are performed in the same way as component-wise addition and multiplication of the vector of slots. In particular, because there is no need for the values in the slots to be only bits, we can use them to represent elements in $\mathbb{Z}_{2^r}$. We recommend that the readers review the original reference [33] for more details.

2.2 Security Model

We will consider the following threat model. First, we assume that an SQL server is semi-honest. Thus, it should follow all specifications of our scheme. However, an adversary is allowed to access all databases maintained by a corrupted SQL server. Moreover, a corrupted DBA may become such an attacker. It is fairly plausible for an attacker to legally login to the SQL server, to make an illegal copy of interesting data, and to hand it over to any malicious buyer. Therefore, the DB server should learn nothing about a query beyond what is explicitly revealed (*e.g.*, the number of tuples).

Second, we assume that a DB user is also semi-honest but is not allowed to collude with an SQL server. Some corrupted DB users can create an illegal copy of sensitive data; however, the volume of illegally copied data leaked at any given time is assumed to be negligible. The DB user should not be given access to data that are not part of the query result.

To formulate our security model, we follow Boneh et al.'s security definition [5]. Specifically, the dishonest DB server should not be able to distinguish between $\bar{Q}_0$ and $\bar{Q}_1$, where two transformed queries $\bar{Q}_0$ and $\bar{Q}_1$ have the same syntactical form. Moreover, the adversarial DB user should not be able to distinguish two encrypted DBs $\overline{\mathsf{DB}}_0$ and $\overline{\mathsf{DB}}_1$ for every fixed query Q and for all pairs of DBs $(\mathsf{DB}_0, \mathsf{DB}_1)$ such that $Q(\overline{\mathsf{DB}}_0) = Q(\overline{\mathsf{DB}}_1)$.

3 Circuit Primitives

We devise three primitives, equality, comparison, and integer addition circuits, by which queries are represented compactly. We focus on a method of optimizing these circuits with respect to the depth and required homomorphic operations. To do this, we make use of SIMD operations along with an automorphism operation.

As mentioned before, computing on two encrypted data may appear both at the `where` clauses and at the `select` clauses. Hence, we should prepare an equality-test circuit and a greater-than comparison circuit for the former and an addition circuit for the latter. When input messages are decomposed and encrypted in a bitwise manner, the encryption $\bar{x}$ of a message $x = x_{\mu-1} \cdots x_1 x_0$ means $\{\bar{x}_0, \bar{x}_1, \ldots, \bar{x}_{\mu-1}\}$, where $x_i \in \{0, 1\}$.

We use "+" to denote homomorphic addition and A to denote the number of homomorphic additions. Similarly, for homomorphic multiplication, we use " $\cdot$ " and M. For two integers $a \leqslant b$, we use $[a, b]$ to denote the set $\{a, a+1, \ldots, b\}$.

3.1 Equality Circuit

For two μ-bit integers x and y, we define an arithmetic circuit for the equality test as follows:

$$\texttt{equal}(\bar{x}, \bar{y}) = \prod_{i=0}^{\mu-1} (1 + \bar{x}_i + \bar{y}_i)\,. \tag{1}$$

The output of $\texttt{equal}(\cdot, \cdot)$ is $\bar{1}$ in the case of equality and $\bar{0}$ otherwise.

In the bit-sliced implementation, we assume that one ciphertext is used per bit; therefore, we have 2μ ciphertexts in total for evaluating the equality test. Instead of regular multiplication, if we multiply each term after forming a binary-tree structure, the depth of the `equal` circuit becomes $\log \mu$. Specifically, the algorithm requires two homomorphic additions for computing $1 + \bar{x}_i + \bar{y}_i$ and that μ ciphertexts be multiplied by each other while consuming $\log \mu$ depth.

Optimizations Our optimizations are focused on minimizing the number of homomorphic operations, especially for homomorphic multiplication. We apply the optimization technique below to the other two circuits.

As mentioned above, as shown by Smart and Vercauteren [32], we can pack each bit x_i into a single ciphertext. Rather than $\bar{x} = \{\bar{x}_0, \ldots, \bar{x}_{\mu-1}\}$, we consider the ciphertext $\bar{x}$ as follows:

$$\bar{x} = \boxed{\bar{x}_0} \boxed{\bar{x}_1} \boxed{\cdots} \boxed{\bar{x}_{\mu-1}}$$

Next, we expand the right-hand side of Equation (1) and rearrange each term so as to fit in well with the SIMD executions. Then, we repeatedly apply SIMD operations to a vector of SIMD words. This is the key to reducing the number of homomorphic multiplications from $\mu - 1$ to $\log \mu$. For example, one SIMD homomorphic multiplication can be depicted as in Fig. 2, denoting by $\bar{x}^{(i)}$ the vector obtained by applying the rotation-by-i to each element in $\bar{x}$. Namely, we implement a single automorphism $X \mapsto X^{g^i}$ for some element $g \in \mathbb{Z}_m^*$ of order μ in the original group $\mathbb{Z}_m^*$ and the quotient group $\mathbb{Z}_m^*/\langle 2 \rangle$.

$$\begin{array}{rcccc}
\bar{x} = & \boxed{\bar{x}_0} & \boxed{\bar{x}_1} & \boxed{\cdots} & \boxed{\bar{x}_{\mu-1}} \\
 & & \cdot & & \\
\bar{x}^{(2)} = & \boxed{\bar{x}_{\mu-2}} & \boxed{\bar{x}_{\mu-1}} & \boxed{\cdots} & \boxed{\bar{x}_{\mu-3}} \\
 & & || & & \\
 & \boxed{\bar{x}_0 \cdot \bar{x}_{\mu-2}} & \boxed{\bar{x}_1 \cdot \bar{x}_{\mu-1}} & \boxed{\cdots} & \boxed{\bar{x}_{\mu-1} \cdot \bar{x}_{\mu-3}}
\end{array}$$

Fig. 2: A Sample Step Illustrating SIMD Execution

Due to space limitations, we have omitted a detailed description of an application of the SIMD operations. We provide a better description of the complexity in Table 1.

3.2 Greater-than Comparison Circuit

For two unsigned μ-bit integers, the circuit $\mathtt{comp}(\bar{x}, \bar{y})$ outputs $\bar{0}$ if $x \geqslant y$ and $\bar{1}$ otherwise. This operation can be recursively defined as follows:

$$\mathtt{comp}(\bar{x}, \bar{y}) = \bar{c}_{\mu-1}, \tag{2}$$

where $\bar{c}_i = (1 + \bar{x}_i) \cdot \bar{y}_i + (1 + \bar{x}_i + \bar{y}_i) \cdot \bar{c}_{i-1}$ for $i \geqslant 1$ with an initial value $\bar{c}_0 = (1 + \bar{x}_0) \cdot \bar{y}_0$.

Optimizations As the first step of optimization, we express Equation (2) in the following closed form

$$\bar{c}_{\mu-1} = (1 + \bar{x}_{\mu-1}) \cdot \bar{y}_{\mu-1} + \sum_{i=0}^{\mu-2} (1 + \bar{x}_i) \cdot \bar{y}_i \cdot d_{i+1} d_{i+2} \cdots d_{\mu-1},$$

where $d_j = (1 + \bar{x}_j + \bar{y}_j)$. Because it has degree $\mu + 1$, we can deduce that the depth of the circuit is $\log(\mu + 1)$. Next, it is easy to see that a naive construction of the circuit incurs $\mathcal{O}(\mu^2)$ homomorphic multiplications.

The key observation is that the closed form is expressed by a sum of products of $(1 + \bar{x}_i) \cdot \bar{y}_i$ and $(1 + \bar{x}_i + \bar{y}_i)$ terms for $i \in [0, \mu - 1]$. We are able to compute $(1 + \bar{x}_i) \cdot \bar{y}_i$ for all i using only 1 homomorphic multiplication due to the use of the SIMD technique. Now, we have to compute $\prod_{k=i}^{\mu-1} d_k$ for each $i \in [1, \mu - 1]$. As mentioned above, a naive method incurs $\mathcal{O}(\mu^2)$, but using SIMD operations requires one to perform only $2\mu - 4$ homomorphic multiplications, consuming $\log \mu$ depth. Finally, we need to multiply $(1 + \bar{x}_i) \cdot \bar{y}_i$ by the result of the above computation, which also incurs only 1 homomorphic multiplication. Thus, the total number of homomorphic multiplications equals $2\mu - 2$.

Remark 1 *We can address the* signed *numbers by slightly modifying the circuit. Assume that we place a sign bit in the leftmost position of a value (e.g., 0 for a positive number and 1 for a negative number) and use the two's complement system. Then, for two μ-bit values x and y,* $\mathtt{comp}(\bar{x}, \bar{y}) = \bar{c}_{\mu-1} + \bar{x}_{\mu-1} + \bar{y}_{\mu-1}$. *It is clear that the case of two positive numbers corresponds to $\bar{x}_{\mu-1} = \bar{y}_{\mu-1} = \bar{0}$.*

3.3 Integer Addition Circuit

Suppose that for two μ-bit integers x and y and for an integer $\nu > \mu$, we construct two ν-bit integers by padding zeros on the left. Then, a size-ν full-adder $\mathtt{fadd}_\nu$ is recursively defined as follows:

$$\mathtt{fadd}_\nu\,(\bar{x}, \bar{y}) = (\bar{s}_0, \bar{s}_1, \cdots, \bar{s}_{\nu-1})$$

where a sum $\bar{s}_i = \bar{x}_i + \bar{y}_i + \bar{c}_{i-1}$ and a carry-out $\bar{c}_i = (\bar{x}_i \cdot \bar{y}_i) + ((\bar{x}_i + \bar{y}_i) \cdot \bar{c}_{i-1})$ for $i \in [1, \nu - 1]$ with initial values $\bar{s}_0 = \bar{x}_0 + \bar{y}_0$ and $\bar{c}_0 = \bar{x}_0 \cdot \bar{y}_0$. The main reason for considering such a large full-adder is to cover SQL aggregate functions with many additions.

Optimizations Our strategy for optimization is the same as above. Namely, we express each sum and carry in the closed form and find a way to minimize the number of homomorphic operations using SIMD and automorphism operations. As a result, $\bar{s}_i$'s are written as follows:

$$\bar{s}_i = \bar{x}_i + \bar{y}_i + \sum_{j=0}^{i-1} t_{ij}$$

where $t_{ij} = (\bar{x}_j \cdot \bar{y}_j) \prod_{j+1 \leqslant k \leqslant i-1} (\bar{x}_k + \bar{y}_k)$ for $j < i - 1$ and $t_{i,i-1} = \bar{x}_{i-1} \cdot \bar{y}_{i-1}$. When $i = \nu - 1$ and $j = 0$, because $\nu - 2$ homomorphic multiplications are required, we see that the circuit has $\log(\nu - 2)$ depth. However, we need to perform an additional multiplication by $\bar{x}_j \cdot \bar{y}_j$. Thus, the total depth amounts to $\log(\nu - 2) + 1$. As before, the use of SIMD and parallelism by automorphism allows us to evaluate the integer addition circuit with only $3\nu - 5$ homomorphic multiplications, while a naive method requires $\dfrac{(\nu^3 - 3\nu^2 + 8\nu)}{6}$ homomorphic multiplications. Due to space limitations, we omit the details of the calculation of the counts.

4 Search-and-compute on Encrypted Data

In this section, we show how to efficiently perform queries with both basic operations (*i.e.*, search and compute operations) on encrypted data using the circuit primitives. To this end, we first describe our techniques in a general setting and then show how our ideas are applied to database applications.

Table 1: Complexity of Circuit Primitives

	Circuits	Complexity
Depth†	`equal`	$\log \mu$
	`comp`	$1 + \log \mu$
	`fadd`	$1 + \log(\nu - 2)$
Comp.‡	`equal`	$2\mathsf{A} + (\log \mu)\,\mathsf{M}$
	`comp`	$(\mu + 1 + \log \mu)\,\mathsf{A} + (2\mu - 2)\,\mathsf{M}$
	`fadd`	$\nu\mathsf{A} + (3\nu - 5)\,\mathsf{M}$

†Depth: The number of levels for homomorphic evaluations

‡Comp.: Computational complexity during homomorphic evaluations

4.1 General-Purpose Search-and-Compute

We begin by describing our basic idea for performing a *search* operation over encrypted data. We assume that a collection of data is partitioned into N μ-bit items denoted by $x_1 \parallel \cdots \parallel x_N$ and that the data have been encrypted and stored in the form of $\bar{x}_1 \parallel \cdots \parallel \bar{x}_N$. In addition, for an SWHE scheme $(\mathsf{Kg}, \mathsf{E}, \mathsf{D}, \mathsf{Ev})$, the key pair $(pk, sk) \leftarrow \mathsf{Kg}(1^\kappa)$ determines its ciphertext space denoted by $\mathcal{C}_{pk}$.

For a predicate φ on $\mathcal{C}_{pk}$, a search on encrypted data outputs $\bar{x}_i$ if $\varphi(\bar{x}_i) = \bar{1}$ and $\bar{0}$ otherwise. More formally, let $\varphi : \mathcal{C}_{pk} \to \{\bar{0}, \bar{1}\}$ be a predicate on encrypted data. Then, we say that $S_\varphi : \mathcal{C}_{pk}^N \to \mathcal{C}_{pk}^N$ is a search on the encrypted data and define $S_\varphi(\bar{x}_1, \ldots, \bar{x}_N) := (\varphi(\bar{x}_1) \cdot \bar{x}_1, \ldots, \varphi(\bar{x}_N) \cdot \bar{x}_N)$.

We then extend this operation to a more general operation on encrypted data, *i.e.*, search-and-compute on encrypted data, as follows. Let $F : \mathcal{C}_{pk}^N \to \mathcal{C}_{pk}$ be an arithmetic function on encryptions. Then, for restricted search $S_\varphi : \mathcal{C}_{pk}^N \to \mathcal{C}_{pk}^N$, we say that $(F \circ S_\varphi)(\bar{x}_1, \ldots, \bar{x}_N)$ is search-and-compute on encryptions.

Further, we measure the efficiency of the search-and-compute operations on encrypted data in Theorem 1. The theorem states that if we can perform a search on encrypted data restricted by φ, which specifies only the equality operator, then the search queries on encrypted data require $N(2\mathsf{A} + \log \mu \mathsf{M})$ homomorphic operations in total. In general, it is not difficult to construct an involved search on encrypted data by composing two circuit primitives–`equal` and `comp`. If a predicate φ allows one to specify all the comparison operators in the set $\{<, \leqslant, >, \geqslant, \neq\}$, then we can perform $S_\varphi(\bar{x}_1, \ldots, \bar{x}_N)$ with $\mathcal{O}(\mu N)$ homomorphic multiplications.

Theorem 1 *Let $\varphi, \mathcal{C}_{pk}$, and $\bar{x}_1, \ldots, \bar{x}_N$ be defined as above. Let $\mathsf{M}(\varphi)$ and $\mathsf{M}(F)$ be the total number of homomorphic multiplications for φ and F, respectively. Then, we can perform $(F \circ S_\varphi)(\bar{x}_1, \ldots, \bar{x}_N)$ with $\mathcal{O}(N(\mathsf{M}(\varphi)) + \mathsf{M}(F))$ homomorphic operations. Specifically, we can perform a search on encrypted data restricted by φ using at most $\mathcal{O}(N(\mathsf{M}(\varphi)))$ homomorphic operations.*

Proof. Because homomorphic multiplication dominates the performance of the operation, we might only count it. Because a predicate φ requires $\mathcal{O}(\mathsf{M}(\varphi))$ homomorphic operations, we see that S_φ requires $\mathcal{O}(N(\mathsf{M}(\varphi)))$ homomorphic operations to compute the predicate N times. Then, the operation uses $\mathcal{O}(\mathsf{M}(F))$ homomorphic operations to evaluate an arithmetic function F on encrypted data. Therefore, we can conclude that the total computation complexity of search-and-compute on encryptions is $\mathcal{O}(N(\mathsf{M}(\varphi)) + \mathsf{M}(F))$. In particular, if we consider the search on encrypted data, F can be considered

to be the identity map. Therefore, we can perform a search on encrypted data restricted by φ using at most $\mathcal{O}(N(\mathsf{M}(\varphi)))$ homomorphic operations.

Security Secrecy against a semi-honest DB server is ensured because encrypted data cannot be leaked due to the semantic security of our underlying SWHE scheme. Secrecy against a semi-honest DB user follows because the result of queries expressed by our circuit primitives is equivalent to $\bar{0}$ if specified conditions do not hold; therefore, the resulting ciphertext is equal to $\bar{0}$. This implies that the evaluated ciphertexts do not leak anything else except for the number of unsatisfied tuples.

4.2 Applications to Encrypted Databases

We denote $\mathsf{R}(A_1, \ldots, A_d)$ as a relation schema R of degree d consisting of attributes $A_1, \ldots, A_d$, and we denote by $\bar{A}_j$ the corresponding encrypted attribute. As mentioned above, we use $A_j^{(i)}$ to denote the j-th attribute value of the i-th tuple, and for convenience, we assume that each attribute value has a length of μ bits.

Search Queries

Simple Selection Queries Consider a simple retrieval query as follows:

$$\begin{aligned} &\texttt{select}\ A_{j_1}, \ldots, A_{j_s} \\ &\quad \texttt{from}\ \mathsf{R} \\ &\texttt{where}\ A_{j_0} = \alpha; \end{aligned} \tag{Q.1}$$

where α is a constant value and $s \leqslant d, j_0 \in [1, d]$.

An efficient construction of (Q.1) using our `equal` circuit is as follows:

$$\texttt{equal}\left(\bar{A}_{j_0}^{(i)}, \bar{\alpha}\right) \cdot \left(\bar{A}_{j_1}^{(i)}, \ldots, \bar{A}_{j_s}^{(i)}\right) \tag{$\bar{\mathsf{Q}}$*.1}$$

for each $i \in [1, N]$. It follows from Theorem 1 that the construction ($\bar{\mathsf{Q}}$*.1) has the complexity evaluation given in Table 2.

Conjunctive & Disjunctive Queries The query (Q.1) is extended by adding one or more conjunctive or disjunctive conditions to the `where` clause. Consider a conjunctive query as follows:

$$\begin{aligned} &\texttt{select}\ A_{j_1}, \ldots, A_{j_s} \\ &\quad \texttt{from}\ \mathsf{R} \\ &\texttt{where}\ A_{j'_1} = \alpha_1\ \texttt{and}\ \cdots\ \texttt{and}\ A_{j'_\tau} = \alpha_\tau; \end{aligned} \tag{Q.2}$$

The query (Q.2) is expressed as the following: For each $i \in [1, N]$,

$$\prod_{k=1}^{\tau} \texttt{equal}\left(\bar{A}_{j'_k}^{(i)}, \bar{\alpha}_k\right) \cdot \left(\bar{A}_{j_1}^{(i)}, \ldots, \bar{A}_{j_s}^{(i)}\right). \tag{$\bar{\mathsf{Q}}$*.2}$$

A disjunctive query whose logical connectives are all ors is also efficiently evaluated by the following expression: For each $i \in [1, N]$,

$$\left(1 + \prod_{k=1}^{\tau} \left(\texttt{equal}\left(\bar{A}_{j'_k}^{(i)}, \bar{\alpha}_k\right) + 1\right)\right) \cdot \left(\bar{A}_{j_1}^{(i)}, \ldots, \bar{A}_{j_s}^{(i)}\right).$$

Denoting by τ the number of connectives, ($\bar{\mathsf{Q}}^*.2$) additionally requires $\log \tau$ in depth to compute the multiplications among the τ equality tests in comparison with ($\bar{\mathsf{Q}}^*.1$). Note that in general, $\tau \ll N$. Table 2 reports the complexity analysis.

Table 2: Complexity of Search Queries

	Queries	Complexity
Depth	($\bar{\mathsf{Q}}^*.1$)	$1 + \log \mu$
	($\bar{\mathsf{Q}}^*.2$)	$1 + \log \mu + \log \tau$
Comp.	($\bar{\mathsf{Q}}^*.1$)	$2N\mathsf{A} + N\,(1 + \log \mu)\,\mathsf{M}$
	($\bar{\mathsf{Q}}^*.2$)	$2\tau N\mathsf{A} + \tau N\,(1 + \log \mu)\,\mathsf{M}$

Search-and-compute Queries We continue presenting important real constructions as an extension of Theorem 1, in which F is one of the built-in SQL aggregate functions–`sum`, `avg`, `count` and `max`. We begin with the case $F =$ `sum`.

Note that in contrast to Lauter et al.'s approach [24], because our plaintext space is $\mathbb{Z}_2$, we should be careful when addressing search-and-compute queries.

Search-and-sum Query Consider the following `sum` query:

$$\begin{aligned} &\texttt{select sum}(A_{j_1}) \\ &\quad \texttt{from R} \\ &\quad \texttt{where } A_{j_0} = \alpha; \end{aligned} \tag{Q.3}$$

As mentioned above, due to our plaintext space being $\mathbb{Z}_2$, repeatedly applying simple homomorphic additions does not ensure correctness. This is the motivation for our integer addition circuit (See Section 3.3). Now, we can efficiently perform (Q.3), expressed as follows:

$$\texttt{fadd}_{\mu+\log N}\left(\texttt{equal}\left(\bar{A}^{(i)}_{j_0}, \bar{\alpha}\right) \cdot \bar{A}^{(i)}_{j_1}\right). \tag{$\bar{\mathsf{Q}}^*.3$}$$

Because the result of the search-and-sum query is less than $2^{\mu}N$, it suffices to use a full adder of size $\nu = \mu + \log N$ for adding all the values. Using our optimized equality circuit, ($\bar{\mathsf{Q}}^*.3$) requires N equality tests in total and N homomorphic multiplications for each result of the test. Thus, the total computation cost is $(2N + \nu(N-1))\mathsf{A} + (N\,(1 + \log \mu) + (N-1)\,(3\nu - 5)\,\mathsf{M}$ with the depth $1 + \log \mu + \log N\,(1 + \log(\nu - 2))$ based on Theorem 2 below.

Theorem 2 *Let* $|\mathsf{R}|$ *denote the cardinality of a set of tuples from a relation schema* R. *Suppose that all the keyword attributes in the* `where` *clause and the numeric attributes in the* `select` *clause have* $\|kwd\|$ *bits and* $\|num\|$ *bits, respectively. Then, a search-and-sum query can be processed with the depth*

$$1 + \lceil \log(\|kwd\|) \rceil + \lceil \log |\mathsf{R}| \rceil \cdot (1 + \lceil \log (\|num\| + \lceil \log |\mathsf{R}| \rceil - 2) \rceil).$$

Proof. The query ($\bar{\mathsf{Q}}^*.3$) consumes $1 + \lceil \log(\|kwd\|) \rceil$ levels to compute all the equality tests. Then, it performs $(|\mathsf{R}| - 1)$ full-adder operations on the results, each of which is of size $(\|num\| + \lceil \log |\mathsf{R}| \rceil)$ and which consumes $(1 + \lceil \log (\|num\| + \lceil \log |\mathsf{R}| \rceil - 2) \rceil)$ levels.

Thus, if $|\mathsf{R}| = 10^3$, $\|kwd\| = 10$, and $\|num\| = 30$, the number of levels consumed by this query is approximately 75.

Search-and-Count Query We observe that search-and-count queries can be processed in a similar manner.

For example, assume a search-and-count query with $\texttt{count}(*)$ in place of $\texttt{sum}(A_{j_1})$ in (Q.3). The query can also be efficiently processed by

$$\texttt{fadd}_{\log N}\left(\texttt{equal}\left(\bar{A}_{j_0}^{(i)}, \bar{\alpha}\right)\right).$$

Note that the output of search-and-count queries is less than N.

Search-and-Avg Query To process a search-and-compute query with the `avg` aggregate function, it suffices to compute search-and-sum queries because an average can be obtained using one division after decryption.

Search-and-Max(Min) Query It is clear that one can obtain the `max` (or `min`) aggregate function by repeatedly applying the `comp` circuit primitive.

5 Performance Improvements

Due to the benefits of combining SIMD and automorphism operations, we can achieve the three optimized circuit primitives in Section 3. However, there is still room to further improve the performance of the circuit primitives. Our strategies are composed of three interrelated parts: Switch the message space $\mathbb{Z}_2$ into $\mathbb{Z}_t$, adapt the circuit primitives (in Section 3) to $\mathbb{Z}_t$, and fine-tune the circuit primitives using SIMD operations again.

5.1 Larger Message Spaces with Lazy Carry Processing

Lauter et al. [24] presented a comparison of two message-encoding techniques: *bit-wise* encoding and *integer* encoding. The former method (*i.e.*, the message space is $\mathbb{Z}_2$) encrypts messages in a bit-by-bit manner, whereas the latter encrypts them as elements of $\mathbb{Z}_t$ for sufficiently large t. The primary advantage of using $\mathbb{Z}_2$ is that two comparison operations are very cheap. In contrast, running an integer addition circuit on encrypted data is expensive (see Table 3) with respect to the running time.

One of the important motivations of using such a large message space is that the bit length of keyword attributes (*e.g.*, $\leqslant 20$ bits) in the `where` clause is generally smaller than that of numeric-type attributes (*e.g.*, $\geqslant 30$ bits) in the `select` clause. Therefore, it would be of substantial benefit to take the message domain as an integer ring if one can quite efficiently evaluate the addition circuit with much lesser depth.

Specifically, if we represent a numeric-type attribute A in the radix 2^ω, then we have

$$\sum_i A^{(i)} = \sum_k \sum_i [A^{(i)}]_k \cdot (2^\omega)^k;$$

therefore, it suffices to compute $\sum_i [A^{(i)}]_k$ over the integers. Assuming that the plaintext modulus t is sufficiently large, we are able to perform addition without overflow in $\mathbb{Z}_t$. We should note that we only have to process carry operations after computing each of them over the large integer ring.

To verify the performance gained by integer encoding, we report the running time of each circuit primitive in Table 3: equality, great-than comparison, and integer addition. We conducted experiments over 10^2 ciphertexts, and we measured the average running times of equality circuits and comparison circuits for 10-bit keywords. In addition, we performed homomorphic addition over 30-bit integers. Consequently, we observed the drastic performance improvement in the operation. We suspect that integer encoding yields more benefits in performing search-and-compute queries because aggregate functions extensively rely on addition.

Table 3: Running-time Comparisons in $\mathbb{Z}_2$ and $\mathbb{Z}_{2^{14}}$

Message Space	`equal` (10-bits)	`comp` (10-bits)	`add` (30-bits)
$\mathbb{Z}_2$	2.2621 ms	8.5906 ms	228.5180 ms
$\mathbb{Z}_{2^{14}}$	208.6543 ms	307.5200 ms	0.0004 ms

5.2 Calibrating Circuit Primitives

It is clear that the use of a different message space results in modifications of our circuit primitives. Before discussing our modifications in detail, we need to determine some lower bounds of depth for homomorphic multiplication as a function of t. We have two types of homomorphic multiplications: multiplying a ciphertext either by another ciphertext or by a known constant. We formally state this in Theorem 3.

Theorem 3 *Suppose that the native message space of the BGV cryptosystem is a large integer ring $\mathbb{Z}_t$ and that a chain of moduli is defined by a set of primes of roughly the same size, $p_0, \cdots, p_L$, that is, the i-th modulus q_i is defined as $q_i = \prod_{k=0}^{i} p_k$. For simplicity, assume that p is the size of the p_ks. Let us denote by σ the standard deviation in our RLWE instance and by H the Hamming weight of the secret key. For $i \leqslant j$, let $\mathbf{c}$ and $\mathbf{c}'$ be normal ciphertexts at level i and j, respectively. Then, the depth, denoted by $\tilde{\mathtt{d}}$, for multiplying $\mathbf{c}$ and $\mathbf{c}'$ is the smallest nonnegative integer that satisfies the following inequality:*

$$t^2 \cdot \phi(m) \cdot (1 + H) \cdot ([q_i^{-1}]_t)^2 < 6p^{2 \cdot \tilde{\mathtt{d}}}.$$

In addition, the depth, denoted by $\tilde{\mathtt{d}}_c$, for multiplying $\mathbf{c}$ by a constant is the smallest nonnegative integer for which the following inequality holds:

$$\phi(m) \cdot (t/2)^2 < p^{2 \cdot \tilde{\mathtt{d}}_c}.$$

Proof. Before multiplying two ciphertexts, we set their noise magnitude to be smaller than the pre-set constant $B = t^2\phi(m)(1 + H)/12$ by modulus switching. That is, while the noise is larger than B, it is scaled down by the next prime. Therefore, $\mathbf{c}$ becomes a ciphertext with a prime set $\{p_0, \ldots, p_i\}$ and has the dominant noise term B. Similarly, $\mathbf{c}'$ also becomes a ciphertext with a prime set $\{p_0, \ldots, p_j\}$, but if necessary, additional modulus-switching operations are required to bring $\mathbf{c}'$ to the same level

i as $\mathbf{c}$. Subsequently, we obtain a tensor product of the ciphertexts, and the result has the following noise magnitude:

$$2 \cdot B \cdot B \cdot ([{q_i}^{-1}]_t)^2$$

Next, the scale-down is performed by removing small primes p_k's from the current prime-set of the tensored ciphertext; we say that Δ is the product of the removed primes. We then have $2B^2([{q_i}^{-1}]_t)^2/\Delta^2 < B$. By assumption, it may be considered that $\Delta = p^{\tilde{\mathtt{d}}}$, which means that $\tilde{\mathtt{d}}$ is the smallest nonnegative integer that the following inequality satisfies

$$2B([q_i^{-1}]_t)^2 < p^{2\cdot\tilde{\mathtt{d}}}.$$

We now consider the case in which $\mathbf{c}$ is multiplied by a constant. As above, we may assume that it initially has small noise magnitude B by modulus-switching. Then the size of the constant is multiplied by its noise, *i.e.*, the result has approximately the same noise estimate as $B \cdot \phi(m) \cdot (t/2)^2$. Thus, we see that $\tilde{\mathtt{d}}_c$ is the smallest nonnegative integer that satisfies the inequality $\phi(m) \cdot (t/2)^2 < p^{2\cdot\tilde{\mathtt{d}}_c}$. This completes the proof.

As a concrete example, we have $\tilde{\mathtt{d}} = 2$ and $\tilde{\mathtt{d}}_c = 1$ in $\mathbb{Z}_{2^{14}}$ with the assumption that $\sigma = 3.2$, $H = 64$, and $m = 13981$.

We now describe a basic idea that underlies our modifications. It is well known that for $x, y \in \{0, 1\}$, the following properties hold:

$$x \oplus y = x + y - 2 \cdot x \cdot y \quad \text{and} \quad x \wedge y = x \cdot y,$$

where $+$, $-$, and $\cdot$ are arithmetic operations over integers. Based on this observation, our equality test can be rewritten as follows:

$$\mathtt{equal}(\bar{x}, \bar{y}) = \prod_{i=0}^{\mu-1} (1 - \bar{x}_i - \bar{y}_i + 2 \cdot \bar{x}_i \cdot \bar{y}_i)\,.$$

We then see that with only a small extra cost, we can construct a new arithmetic circuit for an equality test working on $\mathbb{Z}_t$. We report the additional cost in Table 4.

Next, consider the `comp` circuit on $\mathbb{Z}_t$. Recall that the closed form of $\bar{c}_{\mu-1}$ is (with slight modification)

$$\bar{c}_{\mu-1} = (1 - \bar{x}_{\mu-1}) \cdot \bar{y}_{\mu-1} + \sum_{i=0}^{\mu-2} (1 - \bar{x}_i) \cdot \bar{y}_i \cdot (d_{i+1} d_{i+2} \cdots d_{\mu-1}).$$

Rather than $d_j = (1 + \bar{x}_j + \bar{y}_j)$, we set $d_j = (1 + 2 \cdot \bar{x}_j \cdot \bar{y}_j - \bar{y}_j - \bar{x}_j) \cdot (1 + 2 \cdot \bar{x}_j \cdot \bar{y}_j - 2\bar{y}_j)$. Table 4 shows the new complexity introduced by this modification.

5.3 SIMD-based Fine Tuning

We do not need to describe the details of SIMD operations in FHE cryptosystems. However, our experiments in Section 6 widely make use of SIMD operations, and several notations related to them are newly presented for discussion. Thus, we add a short description to facilitate a better understanding of our experiments. The reader can refer to [33] for the details.

Table 4: Complexity of Circuit Primitives over $\mathbb{Z}_t$

	Circuits	Complexity
Depth	`equal`	$(1 + \log \mu)\,\tilde{\mathsf{d}} + \tilde{\mathsf{d}}_c$
	`comp`	$(3 + \log \mu)\,\tilde{\mathsf{d}} + \tilde{\mathsf{d}}_c$
Comp.	`equal`	$3\mathsf{A} + (2 + \log \mu)\,\mathsf{M}$
	`comp`	$(4 + \mu + \log \mu)\,\mathsf{A} + 2\mu\mathsf{M}$

In our implementation, plaintexts are elements of $\mathbb{A}_{2^r}$ obtained by setting $t = 2^r$ for a small positive integer r. The polynomial $\Phi_m(X)$ mod 2^r factors into ℓ irreducible factors, each of degree δ, *e.g.*, $\Phi_m(X) = \prod_{j=1}^{\ell} F_j(X)$. Then, each factor corresponds to a plaintext slot. In other words, we consider the ℓ-copies of the space $\mathbb{Z}_{2^r}$ as the plaintext slots. This enables us to directly embed our messages as elements of $\mathbb{Z}_{2^r}$ into each slot. As a result, we can process ℓ messages a single ciphertext at a time.

As in Section 3, we again apply this technique to the devised circuit primitives to reduce the required depth of each circuit and the number of homomorphic operations. Compared with the result of [24], a sum of encryptions of 100 numeric data (all 128-bit data) only requires approximately 0.227 milliseconds, which is 100 times faster.

5.4 Efficiency Results

The remainder of this section reports on the new complexity results from using the new message space $\mathbb{Z}_t$. We measured the complexities of only search-and-sum and -count queries. Table 5 shows the complexity of a search-and-sum query, and Table 6 shows that of a search-and-count query.

Table 5: Complexity of Search-and-sum Queries

	Search	Complexity
Depth	`equal`	$(2 + \log \mu)\,\tilde{\mathsf{d}} + \tilde{\mathsf{d}}_c$
	$\texttt{conj}_\tau$	$(2 + \log \mu + \log \tau)\,\tilde{\mathsf{d}} + \tilde{\mathsf{d}}_c$
	`comp`	$(4 + \log \mu)\,\tilde{\mathsf{d}} + \tilde{\mathsf{d}}_c$
Comp.	`equal`	$(4N - 1)\,\mathsf{A} + N\,(3 + \log \mu)\,\mathsf{M}$
	$\texttt{conj}_\tau$	$((3\tau + 1)\,N - 1)\,\mathsf{A} + \tau N\,(3 + \log \mu)\,\mathsf{M}$
	`comp`	$(N\,(\mu + 5 + \log \mu) - 1)\,\mathsf{A} + N\,(2\mu + 1)\,\mathsf{M}$

6 Experimental Results

This section demonstrates the performance of query processing expressed by our optimized circuit primitives. The essential goal of the experiments in this section is to verify the efficiency of our solution in terms of performance. Thus, we reported the experimental results for each query. We performed a

Table 6: Complexity of Search-and-count Queries

	Search	Complexity
Depth	`equal`	$(1+\log\mu)\,\tilde{\mathtt{d}}+\tilde{\mathtt{d}}_c$
	$\mathtt{conj}_\tau$	$(1+\log\mu+\log\tau)\,\tilde{\mathtt{d}}+\tilde{\mathtt{d}}_c$
	`comp`	$(3+\log\mu)\,\tilde{\mathtt{d}}+\tilde{\mathtt{d}}_c$
Comp.	`equal`	$(4N-1)\,\mathsf{A}+N\,(2+\log\mu)\,\mathsf{M}$
	$\mathtt{conj}_\tau$	$((3\tau+1)\,N-1)\,\mathsf{A}+\tau N\,(2+\log\mu)\,\mathsf{M}$
	`comp`	$(N\,(\mu+5+\log\mu)-1)\,\mathsf{A}+2\mu N\mathsf{M}$

somewhat fair comparison with the prior related works in [24,5], although each work is fairly different from its underlying SWHE scheme and experimental settings.

All experiments reported in our paper were performed on a machine with an Intel Xeon 2.3 GHz processor with 192 GB of main memory running a Linux 3.2.0 operating system. All methods were implemented using the GCC compiler version 4.2.1. In our experiments, we used a variant of a BGV-type SWHE scheme [20] with Shoup's NTL library [30] and Shoup-Halevi's HE library [31]. Throughout this section, when we measured the average running times, we excluded computing times used in data encryption and decryption.

6.1 Adjusting the Parameters

Without a loss of generality, we assume that the bit length of keyword attributes in the `where` clause is 10-bit and that of numeric-type attributes in the `select` clause is 30-bit. The keyword attributes are expressed in a bit-by-bit manner, and each bit is an element of $\mathbb{Z}_{2^r}$. In addition, numeric-type attributes are expressed by the radix 2^ω but are still in the same space $\mathbb{Z}_{2^r}$.

We begin by observing the following relation among the parameters. At this point, we consider the *selectivity* of a selection condition, which means the fraction of tuples that satisfies the condition, and we denote it by ε.

Theorem 4 *Let A be a numeric-type attribute. For a positive integer $\omega \geqslant 1$, suppose that each attribute is written as $A = \sum_k [A]_k \cdot (2^\omega)^k$ with $0 \leqslant [A]_k < 2^\omega$. Then, to process a search-and-sum query, one can take a plaintext modulus with*

$$r = \Theta(\omega + \log(\varepsilon \cdot N)).$$

Similarly, for a search-and-count query, it suffices to choose the parameter r so that $r = \Theta(\log(\varepsilon \cdot N))$.

Proof. The goal of the theorem is to provide a bound for the size of a plaintext modulus; therefore, we simply omit an overhead bar for all variables. Let us denote by φ a predicate on encrypted data and by A^* a keyword attribute. Then, a search-and-sum query can be written as

$$\sum_i S_\varphi(A^*, \alpha) \cdot A^{(i)} = \sum_k \left(\sum_i S_\varphi(A^*, \alpha) \cdot [A^{(i)}]_k \right) \cdot (2^\omega)^k.$$

We then have that

$$\sum_i S_\varphi(A^*, \alpha) \cdot [A^{(i)}]_k < 2^\omega \sum_i S_\varphi(A^*, \alpha) = 2^\omega \cdot (\varepsilon N).$$

Thus, for a database with N records, it is sufficient to choose r such that

$$2^\omega \cdot (\varepsilon N) \leqslant 2^r.$$

Note, the larger we make the plaintext modulus 2^r, the more noise there is in the ciphertexts and thus the faster we consume the ciphertext level. Therefore, it appears that $\omega + \log(\varepsilon N)$ is the tight bound for the parameter r.

Because a search-and-count query does not need to consider a specific attribute, we immediately know that $\sum_i S_\varphi(A^*, \alpha) = \varepsilon N < 2^r$.

One may wonder why $S_\varphi(\cdot, \ldots)$ does not take multiple keyword attributes in the proof. Because we consider the selectivity ratio, it does not need to do so. In our experiments, we varied the selectivity ratio from 5 to 40% and plotted the average running time of queries over a database with $N = 10^2, 10^3$, and 10^4 tuples.

6.2 Experiments for Search

We measured the running time per query while varying the number of numeric-type attributes. We take the ring modulus $m = 8191$, and each of the ciphertexts has 630 plaintext slots. For $N = 1$, the experiment of ($\bar{\mathsf{Q}}^*.1$) query is given in the top three rows of Table 7 and that of ($\bar{\mathsf{Q}}^*.2$) is in the bottom three rows in Table 7, where s is the number of attributes, L is the number of ciphertext moduli, and Comm. means the communication cost.

Table 7: Performance of ($\bar{\mathsf{Q}}^*.1$) and ($\bar{\mathsf{Q}}^*.2$)

Message Space	τ	L	s	Timing	Comm.
$\mathbb{Z}_2$	1	6	5	0.38 s	53.99 KB
			10	0.76 s	107.97 KB
			20	1.51 s	215.95 KB
$\mathbb{Z}_2$	4	7	5	2.04 s	73.48 KB
			10	4.09 s	146.96 KB
			20	8.17 s	293.93 KB

6.3 Experiments for Search-and-compute

We conducted a series of additional experiments to measure performance of search-and-compute queries. Because each of the ciphertexts can hold ℓ plaintext slots of elements in $\mathbb{Z}_{2^r}$ and because a numeric-type attribute with a length of 30 bits is encoded into $\tilde{\omega}$ ($= \lceil 30/\log(2^\omega) \rceil = \lceil 30/\omega \rceil$) slots, we can process $\tilde{\ell}$ ($= \lfloor \ell/\tilde{\omega} \rfloor$) attributes per ciphertext.

At first glance, a larger ω seems to be better. However, if ω is too large, by Theorem 4, a plaintext modulus 2^r becomes large. This results in an increased depth of circuits. Therefore, we need to choose a sufficiently large ω whereby the resulting plaintext space is not too large.

Experiments for Search-and-sum We divided our experiment into four cases: (1) Single equality, (2) Multiple equality, (3) Single comparison, and (4) Multiple comparison.

Case I: Single equality This case contains one equality test in the `where` clause. We chose a plaintext space so that the number of plaintext slots is divisible by 10. Then, the entire keyword attribute is packed in only one ciphertext. Further, we take the ring modulus m whereby there exists $g \in \mathbb{Z}_m^*$ that has order 10 in the original group $\mathbb{Z}_m^*$ and in the quotient group $\mathbb{Z}_m^*/\langle 2 \rangle$. Then, there is a Frobenius automorphism of cyclic right shift over those 10 plaintext bits. We used $m = 13981$ so that each of the ciphertexts holds 600 plaintext slots. We report this experimental result in Table 8.

Table 8: Experiments for Case I ($\bar{\mathsf{Q}}^*.3$)

N	ε	Message Space	Radix	L	Timing	Comm.
10^2	$< 16\%$	$\mathbb{Z}_{2^{14}}$	2^{10}	14	3.69s	3.47KB
	$< 32\%$	$\mathbb{Z}_{2^{15}}$		15	3.89s	3.75KB
10^3	$\leqslant 6\%$	$\mathbb{Z}_{2^{16}}$	2^{10}	15	38.78s	3.75KB
	$\leqslant 25\%$		2^8		51.64s	5.01KB
10^4	$\leqslant 10\%$	$\mathbb{Z}_{2^{16}}$	2^6	15	681.05s	6.25KB
	$\leqslant 20\%$		2^5		817.26s	7.50KB
	$\leqslant 40\%$		2^4		1089.68s	10.03KB

Case II: Multiple equality This case contains two or more equality tests in the `where` clause (*i.e.*, $\tau \geqslant 2$). We performed experiments for $\tau = 2$ and $\tau = 4$. When $\tau = 2$, we used $m = 13981$ as before. For the $\tau = 4$ case, we chose $m = 20485$ to support more multiplications than before. Similarly, each ciphertext holds 640 plaintext slots. Compared with queries in the conjunctive form, disjunctive-formed queries require more addition operations. However, both of them require the same depth; therefore, their running times are not significantly different from each other.

Each result is presented in Table 9 and Table 10 (the 6th column of each table consists of two parts: The left part is for conjunctive-formed queries, and the right part is for disjunctive-formed ones.)

Table 9: Experiments for Case II ($\tau = 2$)

N	ε	Message Space	Radix	L	Timing		Comm.
10^2	$< 16\%$	$\mathbb{Z}_{2^{14}}$	2^{10}	16	4.81s	4.84s	3.68KB
	$< 32\%$	$\mathbb{Z}_{2^{15}}$		17	5.12s	5.26s	3.98KB
10^3	$\leqslant 6\%$	$\mathbb{Z}_{2^{16}}$	2^{10}	17	51.63s	52.14s	3.98KB
	$\leqslant 25\%$		2^8		68.83s	69.52s	5.31KB
10^4	$\leqslant 10\%$	$\mathbb{Z}_{2^{16}}$	2^6	17	913.18s	926.11s	6.64KB
	$\leqslant 20\%$		2^5		1095.81s	1111.33s	7.97KB
	$\leqslant 40\%$		2^4		1261.08s	1481.77s	10.63KB

Table 10: Experiments for Case II ($\tau = 4$)

N	ε	Message Space	Radix	L	Timing		Comm.
10^2	$< 16\%$	$\mathbb{Z}_{2^{14}}$	2^{10}	18	9.79s	9.86s	5.09KB
	$< 32\%$	$\mathbb{Z}_{2^{15}}$		19	10.24s	10.28s	5.44KB
10^3	$\leqslant 6\%$	$\mathbb{Z}_{2^{16}}$	2^{10}	19	101.86s	105.15s	5.44KB
	$\leqslant 25\%$		2^8		135.59s	139.97s	7.24KB
10^4	$\leqslant 10\%$	$\mathbb{Z}_{2^{16}}$	2^6	19	1788.19s	1800.84s	9.05KB
	$\leqslant 20\%$	$\mathbb{Z}_{2^{17}}$	2^6	20	1850.70s	1864.36s	9.05KB
	$\leqslant 40\%$	$\mathbb{Z}_{2^{17}}$	2^5	20	2234.81s	2251.30s	10.93KB

Case III: Single comparison This case contains one greater-than comparison in the `where` clause. For the experiments, we used $m = 20485$ in the case of $L = 20$, but in all other experiments, we used $m = 13981$. We report the experimental results in Table 11.

Table 11: Experiments for Case III

N	ε	Message Space	Radix	L	Timing	Comm.
10^2	$< 16\%$	$\mathbb{Z}_{2^{14}}$	2^{10}	17	9.98s	3.71KB
	$< 32\%$		2^9		13.31s	4.94KB
10^3	$\leqslant 6\%$	$\mathbb{Z}_{2^{14}}$	2^8	17	133.12s	4.94KB
	$\leqslant 25\%$		2^6		166.40s	6.18KB
10^4	$\leqslant 10\%$	$\mathbb{Z}_{2^{14}}$	2^4	17	2805.97s	9.88KB
	$\leqslant 20\%$	$\mathbb{Z}_{2^{17}}$	2^6	20	3116.66s	10.66KB
	$\leqslant 40\%$	$\mathbb{Z}_{2^{17}}$	2^5	20	3763.51s	12.88KB

We observed that the results for Case IV are very similar to those for Case II. Thus, due to space limitations, we omitted the Case IV experimental results.

For a better comparison, in Figure 3, we graphically depict the experimental results described above, while the selectivity ratio ε is fixed at 10%.

Experiments for Search-and-count The experiments for search-and-count can also be divided into four cases as performed above. In these experiment, the plaintext modulus $m = 13981$ was used; therefore, each of the ciphertexts holds 600 plaintext slots. Table 12 shows the case with a single equality condition, Table 13 shows that with $\tau = 4$, and Table 14 shows that with a single comparison condition.

Finally, we summarize the above experiments using the graph presented in Figure 4, where we have also fixed the selectivity ratio at 10%.

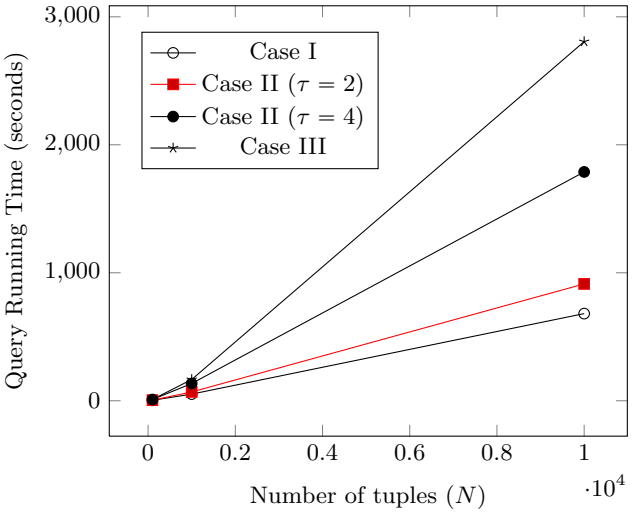

Fig. 3: Experimental Results for Search-and-sum

Table 12: Experiments using Single Equality

N	ε	Message Space	L	Timing	Comm.
10^2	$< 8\%$	$\mathbb{Z}_{2^3}$	7	5.66s	0.73KB
	$< 32\%$	$\mathbb{Z}_{2^5}$	8	7.34s	1.00KB
10^3	$\leqslant 6\%$	$\mathbb{Z}_{2^6}$	10	84.59s	0.93KB
	$\leqslant 25\%$	$\mathbb{Z}_{2^8}$	11	90.89s	1.03KB
10^4	$\leqslant 40\%$	$\mathbb{Z}_{2^{12}}$	12	961.84s	1.12KB

Table 13: Experiments for Multiple Equality ($\tau = 4$)

N	ε	Message Space	L	Timing		Comm.
10^2	$< 8\%$	$\mathbb{Z}_{2^3}$	9	131.35s	132.14s	0.91KB
	$< 32\%$	$\mathbb{Z}_{2^5}$	10	142.28s	144.13s	1.03KB
10^3	$\leqslant 6\%$	$\mathbb{Z}_{2^6}$	12	1718.08s	1741.13s	1.22KB
	$\leqslant 25\%$	$\mathbb{Z}_{2^8}$	15	2184.16s	2178.22s	1.23KB
10^4	$\leqslant 40\%$	$\mathbb{Z}_{2^{12}}$	16	21870.80s	22195.40s	1.25KB

6.4 Handling Join Query

In this section,we design the join queries within the search-and-compute paradigm. For this purpose, suppose that we have the other relation $\mathsf{S}(B_1, \ldots, B_e)$ consisting of M tuples. For simplicity, we assume that $N \geqslant M$.

Table 14: Experiments using Single Comparison

N	ε	Message Space	L	Timing	Comm.
10^2	$< 8\%$	$\mathbb{Z}_{2^3}$	8	17.10s	0.82KB
	$< 32\%$	$\mathbb{Z}_{2^5}$	9	19.24s	0.91KB
10^3	$\leqslant 6\%$	$\mathbb{Z}_{2^6}$	11	224.04s	0.93KB
	$\leqslant 25\%$	$\mathbb{Z}_{2^8}$	15	311.84s	1.25KB
10^4	$\leqslant 40\%$	$\mathbb{Z}_{2^{12}}$	15	3029.05s	1.25KB

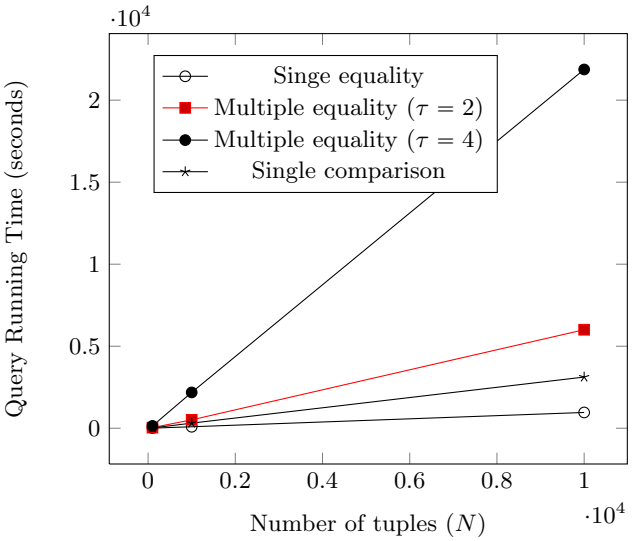

Fig. 4: Experimental Results for Search-and-count

First, we consider a simple join query as follows:

$$\begin{aligned} &\texttt{select } \mathsf{r}.A_{j_1}, \ldots, \mathsf{r}.A_{j_s}, \mathsf{s}.B_{j'_1}, \ldots, \mathsf{s}.B_{j'_{s'}} \\ &\quad \texttt{from } \mathsf{R} \texttt{ as } \mathsf{r}, \mathsf{S} \texttt{ as } \mathsf{s} \\ &\quad \texttt{where } \mathsf{r}.A_{j_k} = \mathsf{s}.B_{j'_{k'}}; \end{aligned} \tag{Q.4}$$

This type of query is expressed and efficiently processed using only the equality circuit. Specifically, for each $i \in [1, N], i' \in [1, M]$, the query (Q.4) is expressed as

$$\texttt{equal}\left(\mathsf{r}.\bar{A}^{(i)}_{j_k}, \mathsf{s}.\bar{B}^{(i')}_{j'_{k'}}\right) \cdot \left(\mathsf{r}.\bar{A}^{(i)}_{j_1}, \mathsf{s}.\bar{B}^{(i')}_{j'_1}, \ldots\right) \tag{$\bar{\mathsf{Q}}^*$.4}$$

For fixed i and i', we suppose that each numeric-type attribute is packed in only one ciphertext. Then, the only difference from ($\bar{\mathsf{Q}}^*$.1) is that ($\bar{\mathsf{Q}}^*$.4) requires two homomorphic multiplications by the result of search operations; thus, we need to perform NM equality tests in total. Hence, the depth of circuit needed to process ($\bar{\mathsf{Q}}^*$.4) is $1 + \log \mu$, and the computation complexity is $(2NM)\,\mathsf{A} + NM\,(2 + \log \mu)\,\mathsf{M}$. In addition, a join query with τ conjunctive conditions needs to perform τNM equality tests; therefore, the required depth is $(1 + \log \mu + \log \tau)$.

Next, we consider an advanced join query demanding search-and-compute operations. For the sake of readability, we consider a join query with two aggregate functions and a simple condition as follows.

$$\begin{array}{c} \texttt{select}\ \texttt{sum}(\mathsf{r}.A_j), \texttt{count}(*) \\ \texttt{from}\ \mathsf{R}\ \texttt{as}\ \mathsf{r}, \mathsf{S}\ \texttt{as}\ \mathsf{s} \\ \texttt{where}\ \mathsf{r}.A_{j_k} = \mathsf{s}.B_{j'_{k'}}; \end{array} \tag{Q.5}$$

We can express the query (Q.5) in a manner similar to that used in Section 4.2. Assuming $\texttt{sum}(\mathsf{r}.A_j) < 2^{\mu}NM$, we use a full adder of size $\nu = \mu + \log(NM)$. By contrast, the result of $\texttt{count}(*) < NM$, and it suffices to use a full adder of size $\log(NM)$. Thus, one candidate of circuit construction for (Q.5) is as follows:

$$\begin{array}{c} \texttt{fadd}_{\mu+\log NM}\left(\texttt{equal}\left(\mathsf{r}.\bar{A}^{(i)}_{j_k}, \mathsf{s}.\bar{B}^{(i')}_{j'_{k'}}\right) \cdot \mathsf{r}.\bar{A}^{(i)}_{j}\right), \\ \texttt{fadd}_{\log NM}\left(\texttt{equal}\left(\mathsf{r}.\bar{A}^{(i)}_{j_k}, \mathsf{s}.\bar{B}^{(i')}_{j'_{k'}}\right)\right). \end{array} \tag{$\bar{\mathsf{Q}}$*.5}$$

With respect to $\texttt{sum}(\mathsf{r}.A_j)$, this is the same as ($\bar{\mathsf{Q}}$*.3), except for the number of operands for additions. Therefore, the depth for evaluation amounts to

$$1 + \log\mu + \log(NM)\left(1 + \log(\nu - 2)\right),$$

and the computation complexity is

$$(2NM + \nu(NM-1))\mathsf{A} + \left(NM\left(1+\log\mu\right) + (NM-1)\left(3\nu-5\right)\right)\mathsf{M}.$$

We remark that it is straightforward to extend this approach to a join query with two or more aggregate functions in the `select` clause.

Finally, we performed some experiments for join queries. We measured the average running time for processing a single equality test while varying N, M from 10 to 10^2 and fixing $s = s' = 5$. Table 15 reports the experimental results for ($\bar{\mathsf{Q}}$*.4). As N and M increase, the running time of the algorithm grows linearly. Table 16 shows the experimental results for ($\bar{\mathsf{Q}}$*.5), assuming the selectivity ratio is fixed at 10%. Because the experiments of ($\bar{\mathsf{Q}}$*.4) are implemented for more numeric-type attributes than those for ($\bar{\mathsf{Q}}$*.5), the query takes longer to perform.

Table 15: Experiments for ($\bar{\mathsf{Q}}$*.4)

$\tau = 1$ and $s = s' = 5$		
$N = M = 10$	$N = 10^2, M = 10$	$N = M = 10^2$
42.75 s	423.86 s	4210.53 s

Table 16: Experiments for ($\bar{\mathsf{Q}}$*.5)

$\tau = s = 1$		
$N = M = 10$	$N = 10^2, M = 10$	$N = M = 10^2$
3.79 s	50.84 s	680.27 s

7 Literature Review

There have been a number of studies with similar goals, as mentioned in Section 1.3. In this section, we present a brief overview of these studies.

We begin with a study on private information retrieval (PIR) primitives. PIR enables a DB user to retrieve a tuple from a DB [11] without revealing which tuple the DB user is retrieving, and with the communication complexity lower than $\mathcal{O}(N)$. However, because the DB user may learn additional bits of information in addition to the originally requested tuples, PIR does not ensure the privacy of the DB server. This issue has been resolved in [21], but in turn, a DB user is required to provide an index of tuples that they would like to obtain. Sometimes, the DB user may not have any information on the index.

The next important research topic is searchable encryption (SE) [34,13,14]. These techniques allow a DB user to encrypt and store their data on a DB server in combination with block ciphers and stream ciphers. Later, the DB user can search for a specific keyword by submitting a trapdoor without revealing keywords and original data. Boneh et al. in [4] generalized this into the public-key setting. Using SE as a primitive, Yang et al. [36] proposed a scheme to privately process a conjunctive query.

There are different research areas focused on realizing private query processing. Hacigümüs *et al.* [22] tried to support general DB queries in a private manner. Hore *et al.* [23] claimed their schemes can support range queries that maintain privacy. However, they later were found to reveal the underlying data distributions. Olumofin and Goldbeg [26] extended PIR into SQL-enabled PIR to privately process general DB queries. They focused on ensuring query privacy but did not consider the privacy of databases. Other works, such as [28,12], assumed that there is a set of mutually trusted and host participants. Ge and Zdonik considered the same security model [17]. Their scheme is, however, restricted to aggregate queries. Ada Popa *et al.*'s CryptDB [1] processed general types of database queries using layers of different encryption schemes: deterministic encryption for equality condition queries, order-preserving encryption for range queries, and homomorphic encryption for aggregate queries. The disadvantage of their work is that in the long run, it downgrades to the lowest level of data privacy provided by the weakest encryption scheme. For example, it may enable one to determine the data order. Recently, TrustedDB [2] achieved the goal by placing the DB engine and all sensitive data processing inside a secure co-processor.

References

1. R. Ada Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In T. Wobber and P. Druschel, editors, *SOSP*, pages 85–100, 2011.
2. S. Bajaj and R. Sion. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In T. Sellis, R. Miller, A. Kementsietsidis, and Y. Velegrakis, editors, *SIGMOD*, pages 205–216, 2011.
3. M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and efficiently searchable encryption. In A. Menezes, editor, *Advances in Cryptology-Crypto*, LNCS 4622, pages 535–552, 2007.
4. D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In C. Cachin and J. Camenisch, editors, *Advances in Cryptology-Eurocrypt*, LNCS 3027, pages 506–522, 2004.
5. D. Boneh, C. Gentry, S. Halevi, F. Wang, and D. Wu. Private database queries using somewhat homomorphic encryption. In M. Jacobson Jr., M. Locasto, P. Mohassel, and eihaneh Safavi-Naini, editors, *ACNS*, LNCS 7954, pages 102–118, 2013.
6. D. Boneh, E.-J. Goh, and K. Nissim. Evaluating 2-DNF formulas on ciphertexts. In J. Kilian, editor, *TCC*, LNCS 3378, pages 325–341, 2005.
7. Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In S. Goldwasser, editor, *ITCS*, pages 309–325, 2012.

8. Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In R. Ostrovsky, editor, *FOCS*, pages 97–106, 2011.
9. Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In P. Rogaway, editor, *Advances in Cryptology-Crypto*, LNCS 6841, pages 505–524, 2011.
10. J. H. Cheon, J.-S. Coron, J. Kim, M. S. Lee, T. Lepoint, M. Tibouchi, and A. Yun. Batch fully homomorphic encryption over the integers. In T. Johansson and P. Q. Nguyen, editors, *Advances in Cryptology-Eurocrypt*, LNCS 7881, pages 315–335, 2013.
11. B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *Journal of the ACM*, 45(6):965–981, 1998.
12. S. Chow, J. Lee, and L. Subramanian. Two-party computation model for privacy-preserving queries over distributed databases. In *NDSS*, 2009.
13. R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In A. Juels, R. Wright, and S. De Capitani di Vimercati, editors, *ACM Conference on Computer and Communications Security*, pages 79–88, 2006.
14. R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. *Journal of Computer Security*, 19(5):895–934, 2011.
15. T. El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. R. Blakley and D. Chaum, editors, *Advances in Cryptology-Crypto*, LNCS 196, pages 10–18, 1984.
16. J. Feigenbaum and M. Merritt. Open questions, talk abstracts, and summary of discussions. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 2:1–45, 1991.
17. T. Ge and S. Zdonik. Answering aggregation queries in a secure system model. In C. Koch, J. Gehrke, M. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C. Y. Chan, V. Ganti, C.-C. Kanne, W. Klas, and E. Neuhold, editors, *VLDB*, pages 519–530, 2007.
18. C. Gentry. Fully homomorphic encryption using ideal lattices. In M. Mitzenmacher, editor, *STOC*, pages 169–178, 2009.
19. C. Gentry and S. Halevi. Implementing Gentry's fully-homomorphic encryption scheme. In K. Paterson, editor, *Advances in Cryptology-Eurocrypt*, LNCS 6632, pages 129–148, 2011.
20. C. Gentry, S. Halevi, and N. Smart. Homomorphic evaluation of the AES circuit. In R. Safavi-Naini and R. Canetti, editors, *Advances in Cryptology-Crypto*, LNCS 7417, pages 850–867, 2012.
21. Y. Gertner, Y. Ishai, E. Kushilevitz, and T. Malkin. Protecting data privacy in private information retrieval schemes. In J. Vitter, editor, *STOC*, pages 151–160, 1998.
22. B. I. Hakan Hacigümüs, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In M. Franklin, B. Moon, and A. Ailamaki, editors, *SIGMOD*, pages 216–227, 2002.
23. B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *VLDB*, pages 720–731, 2004.
24. K. Lauter, M. Naehrig, and V. Vaikuntanathan. Can homomorphic encryption be practical? In C. Cachin and T. Ristenpart, editors, *CCSW*, pages 113–124, 2011.
25. A. Lopez-Alt, E. Tromer, and V. Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In H. Karloff and T. Pitassi, editors, *STOC*, pages 1219–1234, 2012.
26. F. Olumofin and I. Goldberg. Privacy-preserving queries over relational databases. In M. Atallah and N. Hopper, editors, *Privacy Enhancing Technologies*, LNCS 6205, pages 75–92, 2010.
27. P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In J. Stern, editor, *Advances in Cryptology-Eurocrypt*, LNCS 1592, pages 223–238, 1999.
28. M. Raykova, B. Vo, S. Bellovin, and T. Malkin. Secure anonymous database search. In R. Sion and D. Song, editors, *CCSW*, pages 115–126, 2009.
29. R. Rivest, L. Adleman, and M. Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation*, pages 165–179, 1978.
30. V. Shoup. NTL: A library for doing number theory. In http://www.shoup.net/ntl/, 2009.
31. V. Shoup and S. Halevi. Design and implementation of a homomorphic-encryption library. Technical report, IBM Technical Report, 2013.
32. N. Smart and F. Vercauteren. Fully homomorphic SIMD operations. *IACR Cryptology ePrint Archive*, 2011(133), 2011.
33. N. Smart and F. Vercauteren. Fully homomorphic SIMD operations. *Des. Codes Cryptography*, 71(1):57–81, 2014.
34. D. Song, D. Wagner, and A. Perrig. Practical techniques for searching on encrypted data. In *IEEE Symposium on Security and Privacy*, pages 44–55, 2000.
35. M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In H. Gilbert, editor, *Advances in Cryptology-Eurocrypt*, LNCS 6110, pages 24–43, 2010.

36. Z. Yang, S. Zhong, and R. Wright. Privacy-preserving queries on encrypted data. In D. Gollmann, J. Meier, and A. Sabelfeld, editors, *ESORICS*, LNCS 4189, pages 479–495, 2006.