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 homomorphic 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. 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. [17], and much later, Feigenbaum and Merritt's question [12] 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 had been very little progress made in determining whether such efficient and secure encryption schemes exist until 2009, when Gentry constructed such an encryption scheme [13].

While the use of Gentry's scheme and other HE schemes (e.g., [22,7,6]) allows us to securely evaluate any function in a theoretical sense, the evaluation cost is still far from being practical for many functions. 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. Currently, perhaps the simplest way to search for records satisfying a particular condition over encrypted databases is via searchable encryption (*e.g.*, [21,3,2,10]). However, privately processing `sum` and `avg` aggregation queries in the same condition is performed using homomorphic encryption (*e.g.*, [11,16] and [5]). 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.

Recently, 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. 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*? However, 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. In general, SIMD technology allows for basic operations to be performed on several data elements in parallel. On the contrary, our proposal works on packed ciphertexts of several data elements and thus enables one to improve the efficiency of the basic operations 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 $\mathcal{O}(\log \mu)$ depth.
- **Further improvements in the performance of query processing**: HE schemes usually use $\mathbb{Z}_2$ as a message space so that their encryption algorithm encrypts each bit of message. 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 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.

1.2 A High-level Overview of Our Approach

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 a HE scheme and publishes the public key to a DB server. The 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.*, us-

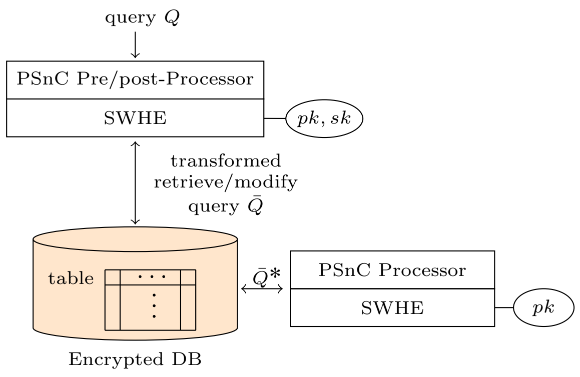

Fig. 1: Our PSnC Framework

ing the `insert-into` statement). 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.

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

1.3 Closely Related Work

A few results closely related to our work can be found in the literature. First, Lauter et al. in [15] showed how to privately compute `avg` and `var` functions using a variant of Brakerski et al.'s SWHE scheme [8]. 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. [4] proposed a way to privately process the `where` clause in a `select` statement and produce a set of matching indices. Their technique uses private set intersection together with homomorphic encryption. It also has the following drawbacks: (1) their scheme only allows conjunctive and disjunctive conditions; (2) the equality test is restricted to comparisons with a constant value; and (3) the users must revisit the server to obtain a list of real tuples because they only know the indices of those tuples.

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 $<>$.

Oraganization. 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.

[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.

2 Preliminaries

In this section, we focus on describing the efficient variant of the Brakerski-Gentry-Vaikuntanathan(BGV)-type cryptosystem [6,14], which is our underlying encryption scheme. In what follows, we give a description of the security model that our constructions assume.

2.1 The BGV-type SWHE Scheme

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. We choose a chain of moduli $q_0 < q_1 < \cdots < q_L = q$ whereby the SWHE scheme can evaluate a depth-L arithmetic circuit. Here is the RLWE-based SWHE scheme:

- $(a, b; \mathsf{s}) \leftarrow \mathsf{Kg}(1^\kappa, h, \sigma, q_L)$: The algorithm Kg chooses a weight h secret key s and generates a RLWE instance (a, b) relative to that secret key. We set the secret key $sk = \mathsf{s}$ and the public key $pk = (a, b)$.
- $\mathbf{c} \leftarrow \mathsf{E}_{pk}(x)$: To encrypt a message $x \in \mathbb{A}_t$, the algorithm chooses a small polynomial v and two Gaussian polynomials e_0, e_1 (with variance σ^2). It outputs the ciphertext $\mathbf{c} = (c_0, c_1)$ by computing

$$(c_0, c_1) = (x, 0) + (bv + te_0, av + te_1) \bmod q_L.$$

- $x \leftarrow \mathsf{D}_{sk}(\mathbf{c})$: Given a ciphertext $\mathbf{c} = (c_0, c_1)$ at level l, the algorithm outputs $x = [c_0 - \mathsf{s} \cdot c_1]_{q_l} \bmod t$.
- $\mathbf{c}_f \leftarrow \mathsf{Ev}_{ek}(f; \mathbf{c}, \mathbf{c}')$: If the function f is an addition over ciphertexts, the algorithm outputs the ciphertext performed by simple component-wise addition of the two ciphertexts. If f is a multiplication over ciphertexts, it outputs the one performed using a tensor product.

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 [4]. 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 (for the `where` clauses) and an integer addition circuit (for the `select` clauses). 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 along with automorphism operation.

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 additions. Similarly, for homomorphic multiplication, we use "$\cdot$" and M.

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. As shown by Smart and Vercauteren [20], we can pack each bit x_i into a single ciphertext. 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 $\log \mu$. We provide a better description of the complexity in Table 1.

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`	$\nu\mathsf{A} + (3\nu - 5)\,\mathsf{M}$

[†]Comp.: Computational complexity during homomorphic evaluations

3.2 Greater-than Comparison Circuit

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

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

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 $i \in [1, \mu - 2]$. 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, $\texttt{comp}(\bar{x}, \bar{y}) = \bar{c}_{\mu-1} + \bar{x}_{\mu-1} + \bar{y}_{\mu-1}$. 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 $\texttt{fadd}_\nu$ is recursively defined as follows: $\texttt{fadd}_\nu(\bar{x}, \bar{y}) = (\bar{s}_0, \bar{s}_1, \cdots, \bar{s}_{\nu-1})$ 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 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 $(\nu^3 - 3\nu^2 + 8\nu)/6$ homomorphic multiplications.

4 Search-and-compute on Encrypted Data

In this section, we show how to efficiently perform queries on encrypted data using the circuit primitives. We first describe our techniques in a general setting and then show how our ideas are applied to database applications.

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 $R_1 \parallel \cdots \parallel R_N$ and that the data have been encrypted and stored in the form of $\bar{R}_1 \parallel \cdots \parallel \bar{R}_N$.

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

We then extend this operation to a more general operation on encrypted data, *i.e.*, search-and-compute on encrypted data, as follows. Let $F : \mathcal{C}^N \to \mathcal{C}$ be an arithmetic function on encryptions. Then, for restricted search $S_\varphi : \mathcal{C}^N \to \mathcal{C}^N$, we say that $(F \circ S_\varphi)(\bar{R}_1, \ldots, \bar{R}_N)$ 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. If a predicate φ allows one to specify all the comparison operators in the set $\{<, \leqslant, >, \geqslant, \neq\}$, then we can perform $S_\varphi(\bar{R}_1, \ldots, \bar{R}_N)$ with $\mathcal{O}(\mu N)$ homomorphic multiplications.

Theorem 1 *Let $\mathsf{M}(\varphi)$ and $\mathsf{M}(F)$ be the total number of homomorphic multiplications for φ and F, respectively. Then, we can perform $(F \circ S_\varphi)(\bar{R}_1, \ldots, \bar{R}_N)$ 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. $\square$

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 of them has a length of μ bits.

4.2.1 Search Queries

Simple Selection Queries. Consider a simple retrieval query as follows:

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

where α is a constant value. An efficient construction of (Q.1) using our `equal` circuit is as follows:

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

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

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

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

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

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

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

A disjunctive query whose logical connectives are all `or`s is also evaluated by changing the predicate into $\left(1 + \prod_{k=1}^{\tau}\left(\texttt{equal}\left(\bar{A}^{(i)}_{j'_k}, \bar{\alpha}_k\right) + 1\right)\right)$. 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 $(\bar{\mathsf{Q}}^*.1)$. Table 2 reports the complexity analysis.

4.2.2 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 $F = \texttt{sum}$.

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

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

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:

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

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. □

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 `count`$(*)$ in place of `sum`(A_{j_1}) in (Q.3). The query can also be efficiently processed by $\texttt{fadd}_{\log N}\left(\texttt{equal}\left(\bar{A}_{j_0}^{(i)}, \bar{\alpha}\right)\right)$.

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.

4.2.3 Join Queries

Now, we design the join queries within the search-and-compute paradigm. Suppose that we have the other relation $\mathsf{S}(B_1, \ldots, B_e)$ consisting of M tuples for $M \leqslant N$. First, we consider a simple join query as follows:

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

Then this type of query is expressed as the following: For each $i \in [1, N], i' \in [1, M]$,

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

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 $(2NM)\,\mathsf{A} + NM\,(2 + \log \mu)\,\mathsf{M}$.

Next, we consider an advanced join query ($\mathsf{Q}.5$) with two aggregate functions $\texttt{sum}(\mathsf{r}.A_j), \texttt{count}(*)$ and the same simple condition as ($\mathsf{Q}.4$). 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 ($\mathsf{Q}.5$) is as follows:

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

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 $1 + \log \mu + \log(NM)\,(1 + \log(\nu - 2))$ and the computation complexity is $(2NM + \nu(NM - 1))\mathsf{A} + (NM\,(1 + \log \mu) + (NM - 1)\,(3\nu - 5))\,\mathsf{M}$.

5 Performance Improvements

There is still room to further improve the performance of the circuit primitives in Section 3. Our strategies are composed of three interrelated parts: Switch the message space $\mathbb{Z}_2$ into $\mathbb{Z}_t$, adapt the circuit primitives to $\mathbb{Z}_t$, and fine-tune the circuit primitives using SIMD operations again.

5.1 Larger Message Spaces with Lazy Carry Processing

If we encrypt messages in a bit-by-bit manner, the primary advantage is that two comparison operations are very cheap, but running an integer addition circuit on encrypted data is expensive (see Table 3). On the contrary, it would be of substantial benefit to take the message domain as a large integer ring if one can quite efficiently evaluate the addition circuit with much lesser depth. 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.

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

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. We suspect that integer encoding yields more benefits in performing search-and-compute queries because aggregate functions extensively rely on addition.

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 polynomial ring $\mathbb{Z}_t[X]/\langle \Phi_m(X)\rangle$ 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_k's. Let us denote 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. Subsequently, we obtain a tensor product of the ciphertexts, and the result has the noise magnitude as $2B([q_i^{-1}]_t)^2$. 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 satisfies the inequality $2B([q_i^{-1}]_t)^2 < p^{2\cdot\tilde{\mathtt{d}}}$.

We now consider the case in which $\mathbf{c}$ is multiplied by a constant. As above, 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 $\phi(m) \cdot (t/2)^2 < p^{2\cdot\tilde{\mathtt{d}}_c}$. $\square$

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 $h = 64$ 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 $\mathbb{Z}_t$. Next, consider the `comp` circuit on $\mathbb{Z}_t$. Recall that the closed form of $\bar{c}_{\mu-1}$ is

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

Rather than $d_j = (1 + \bar{x}_j + \bar{y}_j)$, we set $d_j = (1 + 2 \cdot \bar{x}_j \cdot \bar{y}_j - \bar{y}_j - \bar{x}_j) \cdot (1 + 2 \cdot \bar{x}_j \cdot \bar{y}_j - 2\bar{y}_j)$. As a result, Table 4 shows the complexity results of the search-and-compute queries on encrypted databases of N tuples with μ-bit attributes from using the new message space $\mathbb{Z}_t$.

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.

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

Table 4: Complexity of Search-and-sum Queries

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

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 [14] with Shoup's NTL library [18] and Shoup-Halevi's HE library [19]. 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 $r=\Theta(\omega+\log(\varepsilon N))$.*

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. $\square$

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 5a and that of $(\bar{\mathsf{Q}}^*.2)$ is in the bottom three rows in Table 5a, where s is the number of attributes and L is the number of ciphertext moduli.

6.3 Experiments for Search-and-sum

We conducted a series of additional experiments to measure performance of search-and-sum 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.

We divided our experiment into four cases by types of predicates: (1) Single equality, (2) Single comparison, (3) Multiple equality, and (4) Multiple comparison. In this paper, we only report the experiment results of Case I in Table 5b. We recommend that the readers review the original reference [9] for other experiments in more details.

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. We used $m = 13981$ so that each of the ciphertexts holds 600 plaintext slots.

Acknowledgements. The authors would like to thank the anonymous reviewers of WAHC 2015 for their valuable comments. Jung Hee Cheon and Miran Kim were supported by Samsung Electronics, Co., Ltd. (No. 0421-20140013). Myungsun Kim was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2014R1A1A2058377).

Table 5: Experiment Results

(a) ($\bar{\mathsf{Q}}^*.1$) and ($\bar{\mathsf{Q}}^*.2$)

Message Space	τ	L	s	Timing
$\mathbb{Z}_2$	1	6	5	0.38s
			10	0.76s
			20	1.51s
$\mathbb{Z}_2$	4	7	5	2.04s
			10	4.09s
			20	8.17s

(b) Case I

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

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. 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.
3. 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.
4. 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.
5. 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.
6. Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In S. Goldwasser, editor, *ITCS*, pages 309–325, 2012.

7. Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In R. Ostrovsky, editor, *FOCS*, pages 97–106, 2011.
8. 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.
9. J. H. Cheon, M. Kim, and M. Kim. Search-and-compute on encrypted data. *IACR Cryptology ePrint Archive*, 2014(812), 2014.
10. 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.
11. 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.
12. 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.
13. C. Gentry. Fully homomorphic encryption using ideal lattices. In M. Mitzenmacher, editor, *STOC*, pages 169–178, 2009.
14. 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.
15. K. Lauter, M. Naehrig, and V. Vaikuntanathan. Can homomorphic encryption be practical? In C. Cachin and T. Ristenpart, editors, *CCSW*, pages 113–124, 2011.
16. 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.
17. R. Rivest, L. Adleman, and M. Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation*, pages 165–179, 1978.
18. V. Shoup. NTL: A library for doing number theory. In http://www.shoup.net/ntl/, 2009.
19. V. Shoup and S. Halevi. Design and implementation of a homomorphic-encryption library. Technical report, IBM Technical Report, 2013.
20. N. Smart and F. Vercauteren. Fully homomorphic SIMD operations. *IACR Cryptology ePrint Archive*, 2011(133), 2011.
21. 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.
22. 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.