Updates on Sorting of Fully Homomorphic Encrypted Data

Nitesh Emmadi*, Praveen Gauravaram$†, Harika Narumanchi*, Habeeb Syed*
*TCS Innovation Labs, India
Email: {nitesh.emmadi1,h.narumanchi,habeeb.syed}@tcs.com
†Queensland University of Technology, Australia
Email: praveen.gauravaram@qut.edu.au

***Abstract*—In this paper, we show implementation results of various algorithms that sort data encrypted with Fully Homomorphic Encryption scheme based on Integers. We analyze the complexities of sorting algorithms over encrypted data by considering Bubble Sort, Insertion Sort, Bitonic Sort and Odd-Even Merge sort. Our complexity analysis together with implementation results show that Odd-Even Merge Sort has better performance than the other sorting techniques. We observe that complexity of sorting in homomorphic domain will always have worst case complexity independent of the nature of input. In addition, we show that combining different sorting algorithms to sort encrypted data does not give any performance gain when compared to the application of sorting algorithms individually.**

I. Introduction

Since the invention of Fully Homomorphic Encryption (FHE) by Gentry in 2009 [10], cryptographic applications to secure data stored and managed on the cloud environment have been gaining significant importance. With FHE algorithms it is possible to perform arbitrary computations on the encrypted data and still preserve the confidentiality of the data. Also, there has been consistent research on designing efficient FHE algorithms [9], [8], [21], [15] and their analysis. However, the practical viability of FHE, in general, has not seen a similar level of progress, which is the prime motivation for our research work.

In this paper we focus on applying sorting techniques to sort ciphertexts encrypted using the FHE scheme proposed by Smart and Vercauteren [17]. Sorting is an important cloud computing operation and sorting of ciphertexts stored on cloud managed by a cloud service provider is required whenever sorting has to be computed on the plaintexts and these plaintexts need to be encrypted. For example, on an encrypted Relational Database Management System (RDBMS), the ability to sort on encrypted records is possible, for example, by supporting ORDER BY SQL query on the encrypted records. Sorting over encrypted data through FHE could be useful for real-world cloud service offerings such as Protonmail [2] and Ciphercloud [1] once such techniques are feasible to deploy in practice. This paper focuses on how far we can reach on sorting data in the encrypted domain with the sorting algorithms used to sort plaintexts.

A. Our Contributions

We observe that sorting over data encrypted with a FHE algorithm has worst case complexity irrespective of the nature of input data with respect to the sorting order. We note that even some efficient sorting algorithms such as Merge sort that have $O(n\ log(n))$ complexity in the worst case to sort n plaintext integers, would have $O(n^2)$ complexity in the FHE domain to sort n ciphertext integers. As a result, algorithms that are efficient in sorting plaintext integers would become less efficient when applied to sort encrypted integers. Therefore, it is interesting to investigate on the complexity of the sorting algorithms on the encrypted data and analyze which ones have lower worst case complexity. In this direction, we have analyzed some popular sorting algorithms on the FHE encrypted data and found that Odd-Even Merge Sort(OEMS) outperforms Bitonic, Insertion and Bubble sorting algorithms. We implemented these algorithms to sort arrays of ciphertext integers whose corresponding plaintext integers are 32 bits each. We ran our implementations on a 3.4GHz Intel QuadCore Processor with 16GB RAM running a Ubuntu 14.04 operating system. We compared running times of sorting techniques together with some optimizations. Our experiments show that Odd-Even Merge sort takes significantly less time when compared to the other sorting techniques.

Chatterjee *et al.* [6] described a sorting technique over encrypted data called `LazySort` that combines Bubble sort and Insertion sort to sort homomorphic data. In this technique initially input data is fed to the Bubble sort algorithm to produce a partially sorted array. This partially sorted array is given as input to the Insertion sort to produce a sorted array in linear time [6, Section 4]. We observe that applying Insertion sort on a partially sorted array doesn't take linear time for partially sorted array, thus, contradicting the claims of Chatterjee et al. [6]. Finally, we discuss our ongoing work in this research direction.

B. Related Work

Study of applications of FHE has been an area of pursuit in the last few years mainly because of its impact on cloud computing security [13], [7], [12], [19]. To our knowledge, the idea of sorting an array of encrypted integers was first discussed by Carlos *et al.* [3]. Here authors explicitly presented control flow of Bubble sort algorithm that can sort encrypted data. Later Chatterjee *et al.* [6] used Insertion and Bubble sorting techniques to sort an array of encrypted integers whose plaintext array size is 5-40 by using hcrypt [14] library. This

$Praveen Gauravaram is supported by Australian Research Council Discovery Project grant number DP130104304.

library was programmed in C and is based on FHE scheme proposed by Smart and Verteceuran [17], which has relatively smaller key and ciphertext sizes and yet, following Gentry's bootstrapping blueprint[10] of developing a FHE scheme from a Somewhat Homomorphic Encryption scheme. Chatterjee *et al.* [6] also proposed a sorting technique called `LazySort` which claims a substantial performance improvement compared to the classical Bubble and Insertion sorting techniques. Most recently, Çetin *et al.* [5] proposed techniques to sort encrypted data focused more on reducing the multiplicative depth that effects the performance of such sorting algorithms. Their new algorithms called the Direct sort and Greedy sort have $O(log(N) + log(l))$ multiplicative depth but still they require $O(n^2)$ pre-computed comparisons. We note that observations were similar to ours' on the equivalence between worst case and average case complexity of FHE sorting algorithms was independently made by Çetin *et al.*.

II. Sorting of Encrypted Data

A. Brief Overview of a FHE scheme

A public key FHE scheme $\xi : \mathcal{M} \longrightarrow C$ is described by a tuple of four polynomial time algorithms i.e.

$$\xi = (\texttt{KeyGen}, \texttt{Enc}, \texttt{Dec}, \texttt{Eval})$$

where KeyGen, Enc, Dec denote the key generation, encryption and decryption functions of ξ respectively. Eval is the evaluation algorithm used for computation on encrypted data. This algorithm takes as input a polynomial expression $\mathcal{P}$ and a set of ciphertexts $\texttt{C} = \{C_0, C_1, \ldots, C_n\}$ which are needed to compute $\mathcal{P}$. The input output of Eval satisfies following equation:

$$\texttt{Dec}(\texttt{Eval}(\mathcal{P}, \texttt{C}, \texttt{pk}), \texttt{sk}) = \mathcal{P}(\texttt{Dec}(\texttt{C}, \texttt{sk})) \qquad (1)$$

In the above expression pk denotes keys that are public, like encryption keys or ReCrypt keys and sk denotes private or decryption key which is secret and known only to the generator of the keys. For the sake of brevity, we omit mentioning these keys in our work unless it is essential to use them. To illustrate Eval, consider polynomial expression $\mathcal{P}(X, Y) = X + Y$ which adds two ciphertexts X and Y and results in addition of corresponding plaintexts. According to Equation (1), for ciphertext inputs (A, B), we have

$$\begin{aligned}\texttt{Dec}(\texttt{Eval}(\mathcal{P}, A, B)) &= \texttt{Dec}(\mathcal{P}(A, B))\\ &= \mathcal{P}(\texttt{Dec}(A) + \texttt{Dec}(B))\end{aligned}$$

B. Smart-Vercauteren FHE scheme

The Smart-Vercauteren FHE scheme [17] consists of four algorithms: {KeyGen, Enc, Dec, ReCrypt} parametrized by three values $\{N, \eta, \mu\}$ which are typically taken as $\{N, 2^{\sqrt{N}}, \sqrt{N}\}$. This scheme supports two operations: {Add, Mul}.

KeyGen():

– Set the plaintext space to be $\mathcal{P} = \{0, 1\}$.

– Choose a monic irreducible polynomial $F(x) \in \mathbb{Z}[x]$ of degree N.

– Repeat until p is prime.

- $S(x) \longleftarrow_R \mathcal{B}_{\infty,N}(\eta/2)$.
- $G(x) \longleftarrow 1 + 2.S(x)$.
- $p \longleftarrow \texttt{resultant}(G(x), F(x))$.

– $D(x) \longleftarrow \gcd(G(x), F(x))$ over $\mathbb{F}_p[x]$.

– Let $\alpha \in \mathbb{F}_p$ denote the unique root of $D(x)$.

– Apply the XGCD-algorithm over $\mathbb{Q}[x]$ to obtain $Z(x) = \Sigma_{i=0}^{N-1} z_i x_i \in \mathbb{Z}[x]$ such that $Z(x).G(x) = p \ mod \ F(x)$.

– $B \longleftarrow z_0 \ (mod \ 2p)$.

– The public key is $\texttt{pk} = (p, \alpha)$ and the private key is $\texttt{sk} = (p, B)$.

For fully homomorphic encryption scheme a new algorithm called ReCrypt is defined that takes a ciphertext c and re-encrypts to a new ciphertext c_{new}. This is done by extending the above steps of KeyGen with the following additional operations and two integer parameters s_1 and s_2.

– Generate s_1 uniformly random integers B_i in $[-p, \ldots, p]$ such that there exists a subset S of s_2 elements with $\sum_{j \in S} B_j = B$ over the integers.

– Define $\texttt{sk}_i = 1$ if $i \in S$ and 0 otherwise.Only s_2 of the bits $\{\texttt{sk}_i\}$ are set to 1.

– Encrypt the bits $\texttt{sk}_i = 1$ under the encryption operation to obtain $c_i = \texttt{Enc}(\texttt{sk}_i, \texttt{pk})$.

– The public key is $(p, \alpha, s_1, s_2, \{c_i, B_i\}_{i=1}^{s_1})$.

ReCrypt(c,pk):

– Write down the first t bits of the s_1 floating point numbers $(c.B_i) \ (mod \ 2p)/p$ as an $s_1 \times t$ matrix $(b_{i,j})$.

– Encrypt each of the bits $b_{i,j}$ under the public key pk to obtain an $s_1 \times t$ matrix of clean ciphertexts $(c_{i,j})$.

– Multiply each row of the matrix by the corresponding encryption c_i of $\texttt{sk}_i$ to obtain $(c_i.c_{i,j}) \ mod \ p$. As such we obtain the encryption of a matrix with only s_2 non-zero rows.

– Compute the sum of each column as the Hamming weight using symmetric polynomials and hence reduce the sum of s_1 floating point values to the sum of t floating point values of t bits of precision. More precisely, denote by $h_{i,j}$ the j-th bit of the Hamming weight of the i-th column for $i = 1, \ldots, t$ and $j = 1, \ldots, s$ and form the $t \times t$ matrix $(H_{i,j})$ with $H_{i,j} = h_{i,i-j+s}$ whenever the right hand side is defined and zero otherwise.

– Merge rows of the matrix H, so as to obtain an $s \times t$ matrix H such that the sum of the rows of H equals the sum of the rows of H.

– Apply carry-save-adders to progressively reduce the matrix to one with two rows. Each set of three rows is reduced to two, and then this procedure is repeated.

– Perform the final addition, and output the encryption of a single bit.

```
Enc(M,pk):
– Parse pk as $(p, \alpha)$.
– if $M \notin \{0,1\}$ abort.
– $R(x) \longleftarrow_R \mathcal{B}_{\infty,N}(\mu/2)$.
– $C(x) \longleftarrow M + 2.R(x)$.
– $c \longleftarrow C(\alpha)$ $(mod\ p)$.
– Output $c$.
```

```
Dec(c,sk):
– Parse pk as $(p, B)$.
– $M \longleftarrow (c - \lfloor c.B/p \rceil)$.
– Output $M$.
```

```
Add($c_1, c_2$, pk):
– Parse pk as $(p, \alpha)$.
– $c_3 \longleftarrow (c_1 + c_2)$ $(mod\ p)$.
– Output $c_3$.
```

```
Mul($c_1, c_2$, pk):
– Parse pk as $(p, \alpha)$.
– $c_3 \longleftarrow (c_1.c_2)$ $(mod\ p)$.
– Output $c_3$.
```

C. Sorting of Encrypted Integers

In this section we describe the mathematical model of sorting algorithm over encrypted data developed by Chatterjee *et al.* [6]. Let a, b be any two integers. Using arithmetic based on 2's compliment we have

$$a - b = \underbrace{a + (2's\,\text{compliment of }\; b)}_{(\star)} \tag{2}$$

We denote Most Significant Bit (MSB) of $(\star)$ by β. According to bit-wise arithmetic, the value of β is 1 if the subtraction result is negative and 0 otherwise i.e.

$$a \geq b \qquad \Longleftrightarrow \qquad \beta == 0$$

To swap a and b using the MSB β, we perform the following operations:

$$temp = \beta * a + (1 - \beta) * b;$$

$$b = (1 - \beta) * a + \beta * b;$$

$$a = temp;$$

Based on above observations FHE_SWAP *circuit* [1] [6] is evaluated as follows: Let A, B be two ciphertext integers which are encryptions of a, b respectively. FHE_SWAP circuit takes A, B as input and return another pair of ciphertexts (X, Y) such that $\mathtt{Dec}(X) \leq \mathtt{Dec}(Y)$ (assuming that we are sorting in ascending order).

In other words,

$$(X, Y) \longleftarrow \text{FHE\_SWAP}(A, B) \Longleftrightarrow \mathtt{Dec}(X) \leq \mathtt{Dec}(Y) \tag{3}$$

Using FHE_SWAP as the building block, Bubble sort and Insertion sort algorithms for sorting encrypted data were applied wherein classical swapping of plaintext integers is replaced by FHE_SWAP.

D. FHE_SWAP *and Security of Encryption Scheme*

It is well known that any public-key encryption scheme which supports comparison based operations is not secure against ciphertext only attacks. For any such scheme one can recover plaintext of a target ciphertext by following *binary search* method [16]. In this section, we explain the relevance of this result to sorting algorithms applied on the encrypted data and more specifically to the FHE_SWAP circuit and discuss security implications on the underlying encryption scheme ξ. Recall that the semantic security definition of a public key FHE scheme is the same as that of any public key encryption scheme [11]. In particular, *semantic security* or *Indistinguishability against Choosen Plaintext Attack* (IND-CPA) is independent of `Eval` algorithm [18, Definition 3.1] and every FHE scheme must satisfy IND-CPA security under the assumption that the adversary has access to `Eval` algorithm. For an FHE scheme $\xi = (\mathtt{KeyGen}, \mathtt{Enc}, \mathtt{Dec}, \mathtt{Eval})$, the notion of IND-CPA security can be described as follows: Suppose that $\mathcal{A}$ is a polynomial time adversary and C is a ciphertext such that $C \in \{\mathtt{Enc}(m_0), \mathtt{Enc}(m_1)\}$ for a pair of given plaintexts (m_0, m_1) if

$$\Pr[\mathcal{A}(\mathtt{Enc}(m_b)) \rightarrow m_b] = \frac{1}{2} + \epsilon \qquad b \in \{0, 1\} \tag{4}$$

then ϵ is the *advantage* of the adversary $\mathcal{A}$. The scheme ξ is said to be semantically secure if the advantage ϵ is *negligible* for any polynomial adversary $\mathcal{A}$.

Now we focus on FHE_SWAP and analyze how semantic security of FHE scheme ξ is affected. Note that the binary search attack [16] is not applicable to FHE schemes because although FHE_SWAP supports comparison based operations it does not reveal output of comparison operation. Consider the FHE_SWAP circuit which takes as input a pair of ciphertext integers (A, B) and outputs another distinct pair of ciphertexts (X, Y). The connection between input and output is that $\mathtt{Dec}(X), \mathtt{Dec}(Y) \in \{\mathtt{Dec}(A), \mathtt{Dec}(B)\}$ but there is no way of knowing if $\mathtt{Dec}(X) = \mathtt{Dec}(A)$ or $\mathtt{Dec}(Y) = \mathtt{Dec}(A)$. In other words for any polynomial time adversary $\mathcal{A}$ if

$$\Pr[\mathcal{A}(A, X) \rightarrow \mathtt{Dec}(X) = \mathtt{Dec}(A)]) = \frac{1}{2} + \epsilon' \tag{5}$$

then ϵ' is advantage of $\mathcal{A}$. Note that this advantage ϵ' can be defined for every possible pair $(X, B), (Y, A), (Y, B)$ and not just (X, A). For any pair (A, B) if ϵ' is non-negligible then by considering $m_0 = A, m_1 = B$ in (4) we get ϵ non-negligible, thus breaking semantic security of ξ. The output (X, Y) of FHE_SWAP(A, B) is the result of addition, subtraction and multiplication operations over the input (A, B). Since ξ is assumed to be FHE scheme (X, Y) are indistinguishable from (A, B). Thus, we conclude that although FHE_SWAP operation supports comparison based operations it does not disturb semantic security of ξ.

III. Complexity of Sorting Encrypted Integers

Let $A = \{A_1, \ldots A_n\}$ be a set where $A_i = \mathtt{Enc}(a_i)$ for some plaintext integer a_i that we need to sort (in ascending order). Fundamental difference between sorting of plaintext data and encrypted data is described in Table I.

While sorting plaintext integers, (a_i, a_j) are swapped based on the output of the comparison operation. However, in the

[1] Circuit is the term used in FHE parlance to denote a function which is computable on encrypted data.

TABLE I. Algorithms for sorting encrypted and unencrypted data

Swap on Unencrypted Data	Swap on Encrypted Data
1. For every pair with indices i, j with $i < j$ 2. Compare (a_i, a_j) 3. Swap (a_i, a_j) if necessary	1. For every pair with indices i, j with $i < j$ 2.$(X_i, Y_j) \longleftarrow$ FHE_SWAP(A_i, A_j)

case of encrypted inputs, both comparison and swap are combined together inside the FHE_SWAP circuit. Note that the output of FHE_SWAP(A_i, A_j) is encrypted and without decryption there is no way of knowing if inputs are swapped. Thus, to sort the set A, it is necessary to call FHE_SWAP(A_i, A_j) for every pair with indices i, j $(i < j)$. Unlike sorting plaintext integers, we cannot skip unnecessary comparisons and swaps while sorting encrypted integers. This shows that for any data dependent sorting algorithm like Bubble sort, Insertion sort, Quick sort and Merge sort that depend on FHE_SWAP we have

$$\text{Average Case Complexity} = \text{Worst Case Complexity} \quad (6)$$

The complexities of various sorting algorithms in the encrypted domain and the plaintext domain are given in Table II.

TABLE II. Sorting algorithms and their complexities in plain and encrypted domains.

Algorithm	Plain Domain (Best case)	Encrypted Domain (Any case)
Bubble Sort	$O(n)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$
Quick Sort	$O(nlog(n))$	$O(n^2)$
Merge Sort	$O(nlog(n))$	$O(n^2)$
Bitonic Sort	$O(log^2(n))$	$O(nlog^2(n))$
Odd-Even Merge Sort	$O(log^2(n))$	$O(nlog^2(n))$

IV. Boosting Performance of Sorting

We considered several sorting techniques for FHE sorting including Bubble sort, Insertion sort, Merge sort, Quick sort, BIS and OEMS and analyzed them in terms of number of comparisons. BIS and OEMS algorithms are defined in Algorithms 1,2,3,4. For any sorting algorithm, time taken for sorting depends on the number of comparisons. For FHE sorting, comparison of ciphertext integers in an array is the most expensive operation. Therefore, complexity of FHE sorting directly depends on the number of comparisons. For sorting an array of n elements using Merge sort, the array is subdivided into sub-arrays of size 1. These sub-arrays are repeatedly merged to generate sorted sub-arrays until there are no sub-arrays to merge. In FHE Merge sort, as we do not have the knowledge of the input elements, we cannot avoid comparisons of ciphertexts and FHE_SWAP calls as in the case of un-encrypted integers until we have a single merged list. Therefore, FHE Merge sort requires $O(n^2)$ comparisons.

For sorting an array of n elements using Quick sort, we pick an element called pivot from the array and divide the array such that the elements smaller than the pivot are in the left partition and the elements greater than the pivot are in the right partition. We recursively pick the pivot element for the sub-arrays with smaller values and greater values to produce a sorted array. Therefore, as in the case of Merge sort, FHE Quick sort also requires $O(n^2)$ comparisons. In contrast to Merge sort, the Odd-Even Merge sort is data independent i.e. the number of comparisons are constant independent of the

Algorithm 1: Bitonic Sort

```
Input: An array of encrypted sequences $encarr$, Public
       Key $pk$, Lower Index $lo$, Array Size $n$
Output: Sorted sequences in ascending order or
        descending order
if $n > 1$ then
    $k \leftarrow n/2$
    BITONIC_SORT(encarr, lo, k, 1);
    BITONIC_SORT(encarr, lo + k, k, 0);
    BITONIC_MERGE(encarr, pk, lo, n, b);
```

Algorithm 2: Bitonic Merge

```
Input: An array of encrypted sequences $encarr$, Public
       Key $pk$, Lower Index $lo$, Array Size $n$, bit value
       $b$ 1 for ascending 0 for descending
for $i \leftarrow lo$ to $lo + k$ do
    FHE_SUB($encarr_i$, $encarr_{i+1}$, MSB, pk);
    FHE_SWAP($encarr_i$, $encarr_{i+1}$, MSB, pk);
BITONIC_MERGE(encarr, pk, lo, k, b);
BITONIC_MERGE(encarr, pk, lo + k, k, b);
```

Algorithm 3: Oem_Sort

```
Input: An array of encrypted sequences $encarr$, Public
       Key $pk$, Lower Index $lo$, Array Size $n$,bit value
       $b$ 1 for ascending 0 for descending
Output: Sorted sequences in ascending order or
        descending order
if $n > 1$ then
    $k \leftarrow n/2$
    OEM_SORT(encarr, pk, lo, k, b);
    OEM_SORT(encarr, pk, lo + k, k, b);
    OEM_MERGE(encarr, pk, lo, n, b);
```

Algorithm 4: Oem_Merge

```
Input: An array of encrypted sequences $encarr$, Public
       Key $pk$, Lower Index $lo$, Array Size $n$, bit value
       $b$ 1 for ascending 0 for descending
$m \leftarrow b * 2$ if $m < n$ then
    for $i \leftarrow (lo + b)$ to $(lo + n - b)$ do
        FHE_SUB($encarr_i$, $encarr_{i+b}$, MSB, pk);
        FHE_SWAP($encarr_i$, $encarr_{i+b}$, MSB, pk);
else
    FHE_SUB($encarr_i$, $encarr_{i+b}$, MSB, pk);
    FHE_SWAP($encarr_i$, $encarr_{i+b}$, MSB, pk);
OEM_MERGE(encarr, pk, lo, k, b);
OEM_MERGE(encarr, pk, lo + k, k, b);
```

input. Odd-Even Merge sort divides the input array into two halves containing even index positions (even sub-sequence) and odd index positions (odd sub-sequence). The even and odd sub-sequences are recursively compared and merged to produce a sorted array. Let $\mathcal{T}(n)$ be the number of comparisons performed by Odd-Even Merge sort. For $n > 2$, we have $\mathcal{T}(n) = 2\mathcal{T}(n/2) + n/2 - 1$. Since $\mathcal{T}(2) = 1$, we have $\mathcal{T}(n) = (n/2)(log(n) - 1) + 1 \in \mathcal{O}(n\ log(n))$. Therefore, the number of comparisons in Odd-Even Merge sort defined in Algorithm 3 is $\mathcal{O}(n\ log^2(n))$. Similarly, Bitonic sort is another data independent algorithm that sorts two sub-sequences where first sub-sequence is smaller or equal than the second sub-sequence. Each sub-sequence is sorted recursively by applying Bitonic sort Algorithm 1. To produce a sorted array of length n from two sorted sub-arrays of length $n/2$, $log(n)$ comparisons are required. The number of comparisons for sorting entire array is given by $\mathcal{T}(n) = log(n) + \mathcal{T}(n/2) = log(n)(log(n)+1)/2$. Therefore, Bitonic sort algorithm requires $\mathcal{O}(n\ log^2(n))$ comparisons.

Based on this analysis we implemented Bubble sort, Insertion sort, Bitonic sort and Odd-Even Merge sort in C on encrypted data for l-bit integers with $l = 32$ by using the hcrypt library [14] and evaluated algorithm performance for various array sizes. The execution times for FHE sorting based on these algorithms are shown in Tables III,IV and V where NT refers to the time taken for sorting without optimizations, OTA refers to the time taken for sorting after removal of `ReCrypt` in homomorphic addition function of subtraction and swap modules, OTM refers to sorting time after removal of `ReCrypt` from addition operations in the subtract module and from multiplication operation in the swap module. Even though the size of the ciphertext increases during the multiplication in the swap module, the successive `ReCrypt` operations present in the addition operations would control the size of ciphertexts. This gives us an advantage of suppressing $4 * l$ `ReCrypt` operations per swap in OTM instead of suppressing $2 * l$ `ReCrypt` operations per swap in OTA for l-bit inputs. Our experiments show that OEMS takes about 73% less comparisons and hence 73% less execution time when compared to Bubble sort or Insertion sort. In addition, OEMS takes 19% less comparisons and execution time when compared to BIS. Time taken for sorting n elements is given as the product of number of comparisons and the time taken for each comparison. Time taken for each comparison in the case of NT is 12.5 seconds, for OTA is 6 seconds and in the case of OTM it is 4.7 seconds. We ran all our experiments on a 3.4GHz Intel Quad-core Processor with 16GB RAM running a Ubuntu 14.04 operating system.

Remark 1: Chatterjee *et al.* [6] provide Bubble sort and Insertion sort statistics of 5-40 elements without any details of the system on which the experiments were conducted. As shown in this paper, it is possible to do practical FHE sorting for an array of more than 40 elements. Nevertheless, our experiments show that while sorting an array of 128 integers, sorting algorithms hit a maximum RAM limit of 16GB and consequently sorting aborts. This demonstrates the impracticality of FHE sorting techniques for reasonable array sizes, thus leaving prospects for further research to design more efficient FHE schemes as well as developing techniques to improve sorting efficiency.

Remark 2: As shown in Tables III, IV and V, the input sizes for the Bubble sort, Insertion sort, Bitonic sort and Odd-Even Merge sort need not be a power of 2. These implementations can be adapted for arbitrary input sizes.

TABLE III. Bubble Sort/Insertion Sort Timings

Bubble/Insertion Sort				
#Inputs	#Comparisons	NT(in mins)	OTA(in mins)	OTM(in mins)
4	6	1.25	0.61	0.46
8	28	5.8	2.75	2.2
16	120	25	12	9.43
32	496	103	49	38.9
64	2016	418	199	157.91

TABLE IV. Bitonic Sort Timings

Bitonic Sort Timings				
#Inputs	#Comparisons	NT(in mins)	OTA(in mins)	OTM(in mins)
4	6	1.23	0.6	0.46
8	24	5	2.4	1.86
16	80	16	8	6.31
32	240	49	23	18.98
64	672	139	67	52.63

TABLE V. Odd-Even Merge Sort Timings

Odd-Even Merge Sort Timings				
#Inputs	#Comparisons	NT(in mins)	OTA(in mins)	OTM(in mins)
4	5	1	0.51	0.4
8	19	3.93	1.9	1.5
16	63	13	6.3	4.95
32	191	39.5	19	15
64	543	112	53	42.65

V. On the performance claims of combining sorting techniques

Chatterjee *et al.* [6, Section 4] proposed a new sorting technique called `LazySort`. In this technique, input array is first sorted using the Bubble sorting without `ReCrypt` to produce a partially sorted array. This partially sorted array is then fed to Insertion sort to yield a complete sorted array. Consider the initial input set $A = \{A_1, \ldots, A_n\}$ which is sorted by Bubble sort algorithm giving output $B = \{B_1, \ldots, B_n\}$ which is then sent to Insertion sort for further sorting. It was reported that applying Insertion sort on a partially sorted array works in linear time. However, this is not true. For Insertion sort there is no difference between sending A or B as input because the algorithm cannot identify the difference between partially sorted array and unsorted array leading to the same number of comparisons for both the arrays. Therefore, time taken for sorting by combining sorting algorithms is the sum of time taken by all algorithms for sorting the data.

Remark 3: The timings of LazySort [6] do not follow a consistent pattern of variation i.e. for the input size 10 the improvement is around 36% whereas for 40 elements the improvement is about 93%. However our experiments show a consistent improvement of about 51%.

VI. Discussion and Future work

Improving practicality of FHE schemes is an active research area. A very effective way of achieving this is to use batching techniques [8][20] which pack multiple plaintext bits into single ciphertext thereby reducing actual time and computations performed while evaluating any function on the encrypted data. Another way of reducing execution time of

FHE scheme is to minimize calls to ReCrypt which can be achieved by using modulus switching [9] or relinearization [4] methods. Our ongoing work is focused on using these optimizations to sorting algorithms to make FHE sorting more efficient.

Acknowledgment

Praveen Gauravaram thanks Xavier Boyen, Douglas Stebila and Qinyi Li from QUT and Georg Lippold from Telstra, Australia for discussions on this subject.

References

[1] Ciphercloud. The official website is accessible at http://www.ciphercloud.com/ (Accessed on 01/06/2015).

[2] Protonmail. The official website is accessible at https://protonmail.ch/ (Accessed on 01/06/2015).

[3] C. Aguilar-Melchor, S. Fau, C. Fontaine, G. Gogniat, and R. Sirdey. Recent advances in homomorphic encryption: A possible future for signal processing in the encrypted domain. *Signal Processing Magazine, IEEE*, 30(2):108–117, 2013.

[4] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. *IACR Cryptology ePrint Archive*, page 344, 2011.

[5] Gizem S. Çetin, Yarkin Doröz, Berk Sunar, and Erkay Savas. Depth optimized efficient homomorphic sorting. In *Progress in Cryptology - LATINCRYPT 2015*, volume 9230 of *Lecture Notes in Computer Science*, pages 61–80. Springer, 2015.

[6] Ayantika Chatterjee, Manish Kaushal, and Indranil Sengupta. Accelerating sorting of fully homomorphic encrypted data. In *INDOCRYPT*, volume 8250 of *Lecture Notes in Computer Science*, pages 262–273. Springer, 2013.

[7] Kai-Min Chung, Yael Kalai, and Salil Vadhan. Improved delegation of computation using fully homomorphic encryption. In *Advances in Cryptology–CRYPTO 2010*, pages 483–501. Springer, 2010.

[8] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. Batch fully homomorphic encryption over the integers. In *Advances in Cryptology EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, page 36, 2013.

[9] Jean-Sébastien Coron, David Naccache, and Mehdi Tibouchi. Public key compression and modulus switching for fully homomorphic encryption over the integers. In *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 446–464. Springer, 2012.

[10] Craig Gentry. *A fully homomorphic encryption scheme*. Ph.d. thesis, 2009.

[11] Craig Gentry. Computing arbitrary functions of encrypted data. *Commun. ACM*, 53(3):97–105, 2010.

[12] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 1219–1234. ACM, 2012.

[13] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 113–124. ACM, 2011.

[14] Henning Perl. hcrypt project. This library is accessible at https://github.com/hcrypt-project/libScarab (Accessed on 28/05/2015).

[15] Thomas Plantard, Willy Susilo, and Zhenfei Zhang. Lll for ideal lattices: re-evaluation of the security of gentryhalevis fhe scheme. volume 76, pages 325–344. Springer US, 2015.

[16] R. L. Rivest, L. Adleman, and M. L. Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation*, 1978.

[17] Nigel Smart and Fre Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *Public Key Cryptography - PKC 2010*, pages 420–443. Springer LNCS 6056, 2010.

[18] Vinod Vaikuntanathan. Computing blindfolded: New developments in fully homomorphic encryption. In *FOCS*, pages 5–16. IEEE Computer Society, 2011.

[19] Marten Van Dijk and Ari Juels. On the impossibility of cryptography alone for privacy-preserving cloud computing. *HotSec*, 10:1–8, 2010.

[20] Masaya Yasuda, Takeshi Shimoyama, Jun Kogure, Kazuhiro Yokoyama, and Takeshi Koshiba. Practical packing method in somewhat homomorphic encryption. In *DPM/SETOP*, volume 8247 of *Lecture Notes in Computer Science*, pages 34–50. Springer, 2013.

[21] Zhenfei Zhang, Thomas Plantard, and Willy Susilo. Reaction attack on outsourced computing with fully homomorphic encryption schemes. In Howon Kim, editor, *Information Security and Cryptology - ICISC 2011*, volume 7259 of *Lecture Notes in Computer Science*, pages 419–436. Springer Berlin Heidelberg, 2012.