Searching and Sorting of Fully Homomorphic Encrypted Data on Cloud

Ayantika Chatterjee and Indranil Sengupta

***Abstract*—The challenge of maintaining confidentiality of stored data in cloud is of utmost importance to realize the potential of cloud computing. Storing data in encrypted form may solve the problem, but increases the security issues and diminishes the essence of cloud while performing operations on cloud data by repeated decryption-encryption. Hence, Fully homomorphic encryption (FHE) is an effective scheme to support arbitrary operations directly on encrypted data. Further, cloud mostly acts as storage database, hence secured sorting and searching of FHE cloud data can be an effective field of research. We have investigated the feasibility of performing comparison as well as partition based sort on CPA resistant FHE data and highlight an important observation that time requirement of partition based sort on FHE data is no better than comparison based sort owing to the security of the cryptosystem. We identify the recrypt operation, which is the denoising step of FHE as the main reason of costly timing requirement of such operations. Finally, we propose a two stage sorting technique termed as Lazy sort with reduced recrypt operation, which proves to be better in terms of performance on FHE data in comparison to partition as well as comparison sort.**

***Index Terms*—Cloud, Searching, Sorting, Fully Homomorphic encryption.**

I. Introduction

CLoud services provide a low cost approach to use large shared resources in the domain of data storage and management. However, due to possibility of public access to the information in cloud, cloud security is an important area of research nowadays. Confidentiality of data can be maintained by storing the encrypted form of data in the cloud. This process assures the security but imposes extra challenges in case of performing any operation on such data. Each time it is required to bring the data back to process in unencrypted domain in the client side. This leads to several security issues as the ciphertext is continuously exposed to the adversary. Furthermore, if the computations are performed at the client side, the basic objective of cloud computing is defeated. Hence, it is required to perform direct processing on encrypted data and this is supported by homomorphic encryption scheme. With this scheme the notion of delegating the ability to process secured data without giving access to it was first introduced in [1]. However, Gentry in his work [2], introduced the concept of performing arbitrary manipulations like addition, multiplication etc on encrypted data without the knowledge of secret key and it is termed as fully homomorphic encryption. The basic idea of Fully Homomorphic Encryption (FHE) is as follows: Consider the messages $m_1, \ldots, m_t$, which are encrypted to the ciphertexts $c_1, \ldots, c_t$ with the FHE scheme under some key. For any polynomial time computable function f, the FHE scheme allows anyone to efficiently compute a ciphertext that decrypts to $f(m_1, \ldots, m_t)$ under the secret key.

In recent few years, works on different approaches on FHE schemes have been reported in literature. Fundamental encrypted additions and multiplications on single bits are defined in [2] and implemented using integers in [3] and [4]. A survey and further efficiency enhancement on fully homomorphic encryption has been reported on [5] and [6]. In [7] and [8] some advancements have been proposed to implement faster encryption schemes. However, all such research mainly aim to achieve an increased efficiency of the homomorphic encryption schemes. In [9], [10], Brenner et.al have proposed an encrypted processor to perform encrypted operations. However, implementing operations on encrypted data is not very straightforward as it is done on unencrypted data and works are very limited on investigating how to define practical operations over fully homomorphic data. However, such techniques can be fundamental to apply FHE on cloud data to allow computations over encrypted data in the cloud server without decryption. This motivates us to investigate how different operations can be defined on FHE cloud data. Since, cloud mainly acts as storage database, searching and sorting on stored encrypted data can prove to be very important operations. In this work, we investigate how to actually perform these two operations directly on FHE data. We propose a method for encrypted data search, in which an encrypted data (to be searched) should be send to cloud server where the database with encrypted data is residing. The entire search operation will take place in the server without any decryption operation and the encrypted search result is obtained. Our security analysis further confirms that the final search result should be encrypted otherwise the cryptosystem is prone to CPA attack [11].

Further, sorting has attracted a great deal of attention in computer science research. This theoretically interesting problem of information-shuffling has practical significance in cloud database. However, to the best of our knowledge not much work has been reported to sort data in encrypted domain. In [12], an initial effort has been made to tackle the problem of comparison based sort on FHE data. Present work targets to find an answer to another very important question: can any type of sorting be performed on FHE encrypted data? Since, efficiency is a major challenge in homomorphic domain, partition based sorting can be the first choice over comparison based sort due to the fact that partition based sort works on $O(nlogn)$ time to sort n numbers of unencrypted data. However, our analysis reveals an interesting result that

contrary to unencrypted domain, partition based sort provides the same time complexity (even poorer performance) with comparison based sort for encrypted data. In this context, we provide a formal proof to support that if an adversary is capable of performing traditional partition based sort on FHE data on an encrypted array, then the encryption scheme is prone to chosen plaintext attack (CPA). We show that to perform partition based sort (like Quicksort) one needs to not only encrypt the information, but also the indices of the array. This paper shows that an index encryption allows one to partition the array, but ensuring that the adversary is unable to perform CPA. Further this paper first points out the problem of handling recursion in encrypted domain and hence an iterative approach of quicksort is implemented with encrypted data using specialized stack-functions. In conclusion, implementation results are provided to support our analysis that the FHE Quicksort provides no additional advantage over comparison based sort, like FHE Bubble Sort and Insertion sort. Finally, we propose a two staged sorting technique, LazySort with reduced number of Recrypt operation, which is the de-noising step of FHE scheme. Since, recrypt is one of the costly FHE operations, our analysis shows how efficient reduction of recrypt operation increases the efficiency of sorting. Detailed experimental results show that LazySort is a better choice in terms of performance over comparison or partition based sort in case of encrypted FHE data. As a summary the contributions of the work are:

- We develop techniques for performing search and sort over encrypted data.
- We show that partition based sort over encrypted data does not perform better than comparison sort.
- We highlight the problems of handling recursion on encrypted data with underlying unencrypted processor and show that encrypted indices and encrypted stacks should be realized to support it.
- The security of the underlying FHE has been linked to the operations of searching and sorting.
- We present a technique called Lazysort to show that suitable reductions of costly recrypt operation is a way to improve the efficiency of computations over FHE data.

The overall paper is organized as follows: section II describes the basic concepts of homomorphic encryption and the Scarab library to design homomorphic modules. Section III describes how FHE is related to cloud computing and few related works with respect to encrypted searching and sorting. In section IV we explain the relation between security and search operation on a cryptosystem in the light of CPA and Section V discusses about the proposed search algorithm on FHE data and design of the related submodules. Section VI explains the proposed comparison sort technique with encrypted swap operation. In section VII, we explain the security analysis of performing comparison as well as partition based sort on FHE data and in section VIII we show that partition sort is actually feasible on encrypted data handling along with different design challenges of implementing encrypted operations on underlying unencrypted processor. However, in section IX our timing analysis proves that partition based sort does not provide any gain over comparison sort in case of encrypted data, hence finally in section X we propose a two stage sorting scheme with minimized recrypt operation which helps to make the encrypted sorting efficient. Section XI and section XII include experimental details and final conclusion.

II. Preliminaries

The main objective of this paper is to investigate the searching and sorting operations on FHE cloud data. Before discussing the operations, here we first discuss a few words about the FHE scheme. Fully Homomorphic encryptions provide a mechanism to perform arbitrary computations over encrypted data. The promise shown in the work of Gentry [13] had been followed by several improvements to develop more efficient realizations of this technique, which has potential applications for performing privacy preserving operations, so relevant to cloud computing. In this section, we first provide a brief outline of the FHE scheme and a popular library for performing the basic computations based on this encryption.

A. Fully Homomorphic Encryption

Homomorphism is a structure-preserving transformation between two sets, where an operation on two members in the first set is preserved in the second set on the corresponding members. Let P and C be sets with members $p_1, p_2 \in P$, T is a transformation with an operation $\oplus$ between the two sets with its reverse function T' and an operation $\ominus$. The system is homomorphic, if $\forall(p_1, p_2) \in P$, $(p_1 \oplus p_2) = T'(T(p_1) \ominus T(p_2))$.

Group Homomorphic Encryption (GHE) schemes are public key encryptions that allow to compute an operation on ciphertexts being equivalent to some binary operation on the corresponding plaintexts [11]. Somewhat Homomorphic Encryption (SHE) schemes allow a specific class of functions to be evaluated on ciphertexts. Usually this scheme supports an arbitrary number of one operation but only a limited number of second operation. Fully homomorphic encryption (FHE) scheme is an extended form of group homomorphic encryption (GHE). GHE only supports a single arbitrary operation on plaintext (as well as on ciphertext), whereas FHE supports two arbitrary operations $(+, *)$ on plaintexts (as well as $(\oplus, \odot)$ on ciphertexts).

Gentry defined a FHE scheme which is outlined next. The scheme has the security paramter λ, and sets $N = \lambda$, $P = \lambda^2$, $Q = \lambda^5$. The scheme also uses two integer parameters $0 < \alpha < \beta$ and the following algorithms:

(a) $KeyGen(\lambda)$: Generate a random P-bit odd integer, p. A set $\overrightarrow{y} = \{y_1, y_2, \ldots y_\beta\}$ is generated such that $y_i \in [0, 2)$. Out of these elements, there must exist a sparse subset $S \subset \overrightarrow{y}$ of α elements, such that $\sum_{y_j \in S}(y_j) = \frac{1}{p} \bmod 2$. Set sk to be a binary encoding s of the sparse subset S, where $s = (0, 1)^\beta$. Set $pk \leftarrow (p, \overrightarrow{y})$.

(b) $Encrypt(pk, m)$: Obtain the ciphertext $c = m' + pq$, where m' is a random N-bit integer st. $m = m' \bmod 2$. Generate $\overrightarrow{z} : z_i \leftarrow c.y_i \bmod 2$. Return $c^* = (c, \overrightarrow{z})$. In the rest of the paper, we shall mention $Encrypt(pk, m)$ as $Encrypt$.

(c) $Decrypt(sk, c^*)$: Output LSB(c) XOR LSB($\lfloor \sum_t S_t z_t \rceil$), where LSB() returns the least significant bit of the input, and $\lfloor . \rceil$ returns the nearest integer to the input. Decryption works since (up to small precision errors) $\sum_t S_t z_t = \sum_t c S_t y_t = \frac{c}{p} \bmod 2$.

It can be argued that the above encryption allows quite wonderfully arbitrary computations on encrypted data. Thus we can define operations like $Evaluate(f, c_1, \ldots, c_t)$, where f is an arbitrary operation on the ciphertexts, $c_1, \ldots, c_t$. The result of the computation is always a ciphertext, c whose decryption would be same as the function f applied on the plaintexts corresponding to, $c_1, \ldots, c_t$. However, the decryption can be erroneous if the noise (measured as $c \bmod p$ increases. In order to reduce the error during the computations, there is an additional operation, called *Recrypt* which takes the ciphertext, c and produces another ciphertext, say c' which corresponds to the same plaintext, but with a reduced noise level. The operation is done by allowing to compute the decryption function, as the function f in the *Evaluate* function.

However, direct application of Gentry's FHE scheme has performance issues, hence lots of improvements and approaches from alternate assumptions have been proposed in [3], [14]. In our work, while performing homomorphic operations, we have re-used the homomorphic modules proposed in Scarab library [15].

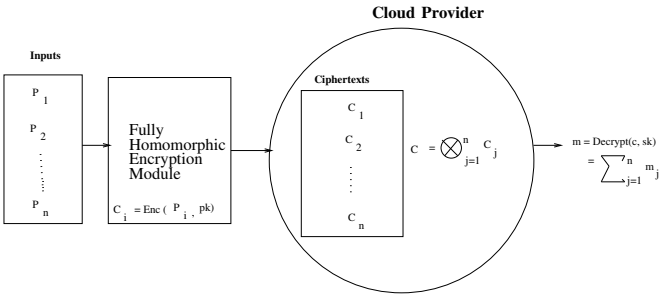

Fig. 1. FHE processing on cloud data

III. Why FHE is essential in Cloud Computing?

In recent years, privacy enhancing technologies in the context of cloud computing is an interesting area of research. Cryptographic solutions provide a possible solution to maintain confidentiality of stored data in cloud server [11], but it requires repeated decryption-encryption for any processing of cloud data. In this scenario, cloud can evaluate arbitrary functions directly on encrypted data without learning the result and having access to the secret key, if cloud data is encrypted with FHE scheme. Figure 1 shows how simple addition can directly be processed on FHE encrypted cloud data in the cloud server without the knowledge of secret key.

Till date FHE is considered to be impractical due to the slow performance and trusted server-side software is considered to be cheap and easy-to-implement alternatives to FHE [11]. However, in the field of cloud computing usage of secure hardware limits the power of outsourcing computations due to the limitation of required hardware. Further, the trusted hardware is usually very resource constrained. Some other approaches like Yao's Garbled circuits [16] suffer from a major drawback, that they need to be rebuilt for different inputs. Gennaro et al. [17] proposed how this limitation can be sorted with the use of FHE as black box. Other solutions like twin clouds [18] and token based cloud computing with additional hardware [19] also suffer due to lack of parallelization scope. In this scenario, FHE is a major solution to provide confidentiality to cloud computing.

As mentioned in [11], "it must be emphasized that homomorphy is a theoretical achievement that merely lets us arithmetically add and multiply plaintexts encapsulated inside a ciphertext. In theory, this allows the execution of any algorithm complex manipulations like text replacements or similar, but putting this to practice requires the design (compilation) of a specific circuit representation for the algorithm at hand. This may be a nontrivial task". To address this issue we choose two widely used operations sorting and searching which are very relevant to cloud computing databases and investigate how they can be realized by FHE operations.

A. Related works

The problem of searching on encrypted data was first considered explicitly by Song, Wagner and Perrig in 2001 [20]. However, this problem was addressed long back in [21] and [22]. The simplest way to perform encrypted data search by deterministic encryption scheme (or property-preserving encryption) suffers from security issues of leaking information [23]. Again the use of functional encryption (FE) performs slow search but better security [24]. Some other encrypted search operations are proposed in [25]–[27] and [28]. However, all these encryption schemes along with attribute based encryption do not support arbitrary operations on cloud data. Performing arbitrary operations on encrypted data is only possible if data is encrypted with Fully homomorphic encryption schemes. In [9], [10] confidential search with homomorphic cryptography is explored However this search scheme used the concept of an obfuscated Bloom Filter, which requires the knowledge of unencrypted database to reduce the search-space by constructing an initial data-structure. The secrecy of the data searched is also maintained by relying on the obfuscation used and not on the underlying encryption used, thus weakening the security of the scheme. In [29], authors propose the idea of order preserving indexing to compare encrypted ciphertext. The data owner (client) uses a trap-door to compute the index which is randomized and provided to the service provider (cloud server). The randomization prevents the leakage of the trap-door. However, as the authors mention that if FHE schemes like [5] are used, the order preserving indices will not remain randomized well. Furthermore, the fact that the service provider can determine the relative ordering of plaintexts from ciphertexts can have potential security implications [30]. In present literature, works on encrypted sorting are also very limited. In [31], few sorting techniques on somewhat homomorphic encrypted data have been discussed. In [32], homomorphic sorting is explored where data is encrypted with additive homomorphic encryption. However, for cloud servers where arbitrary data is stored encrypted with FHE

schemes, dedicated techniques are required for performing specific operations. To emphasize, although theoretically FHE can support arbitrary computations, for efficiency we need to develop suitable techniques of implementation.

In this paper, we consider two important operations on encrypted data, namely searching and sorting. To the best of our knowledge, there is not much reported work in the literature which discusses on the algorithmic issues and challenges faced in implementing common operations like searching and sorting on arbitrary FHE data (not structured data as proposed in [29]). In this pursuit, we face several challenges, like the inefficiency of divide and conquer strategies to reduce the run time of the algorithm. We link the limitations to the fact that the encryption is secured in the Chosen-Plaintext Attack (CPA) sense, and we present reductions to prove that such tricks are not viable on FHE data. On the other hand, we propose some other techniques to make the operations of searching and sorting more practical on FHE data. The main technique that we propose here is the Lazy Computation. In this method, we propose that suitably dropping the number of Recrypts, which are the most costly step in the FHE process, one can get an almost correct result. This can be backed up with some other algorithms which are more efficient on such input, and one can obtain an overall gain.

Throughout our discussion, we assume database, where data is encrypted using FHE. However, for divide and conquer strategies (we show with sorting as an example) the address (index of the array in this context) is also needed to be encrypted. In the next section, we first show the inter-relation between the capability to search the encrypted database with performing a Chosen Plaintext attack (CPA) on the underlying encryption. We subsequently proceed with the proposal of the search technique.

IV. Security Implication of the Capability to Perform Search on FHE Data

A. CPA security in cloud computing

CPA is an extremely practical attack model in the cloud computing context when considering the security of data encrypted with a public key algorithm. The reason is since the encryption key is public, an adversary can easily obtain ciphers for arbitrary chosen plaintexts, which makes CPA feasible. Hence, we evaluate the implementations of our operations on FHE (which is a public key encryption scheme) considering CPA as a threat model.

In our work, we intend to develop a search algorithm on encrypted cloud data. With the availability of such algorithms, it is expected that the client can encrypt data and check if the data is present in the encrypted database of cloud server. The data to be searched, and the database on which the search is performed are all encrypted. The steps of the search is thus performed ensuring that they do not leak information of any of the above, but the client should be certain whether the search result is a success or failure. However, one possible threat of performing the search is that any adversary can send an encrypted data to the cloud and check whether it exists in the cloud database or not (since both the encryption algorithm and cloud server are public) and this may leak information about the cloud database. For this reason, the security notion should confirm that every scheme used in this paper should be Chosen Plaintext Attack (CPA) secure, and we relate the capability of searching on FHE data to perform a CPA on the FHE. As mentioned in [30] a public-key encryption scheme $\Pi = (Gen, Enc, Dec)$ has indistinguishable encryptions under a chosen-plaintext attack (or is CPA secure) if for all probabilistic polynomial-time adversaries A there exists a negligible function $negl$ such that:

$$Pr[PubK^{cpa}_{A,\Pi}(n) = 1] \leq \frac{1}{2} + negl(n) \tag{1}$$

Here, $PubK^{cpa}_{A,\Pi}(n)$ is an experiment which returns 1, when an adversary A is able to determine whether a challenge c corresponds to the encryption of m_0 or m_1, two messages encrypted by Π. We follow the experimental set up for Indistinguishable CPA (IND-CPA) as discussed in [33] and perform the security analysis of searching and sorting operations on FHE data in the subsequent sections.

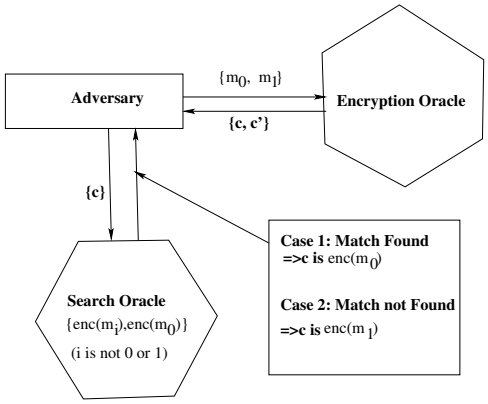

Fig. 2. Relation between Searching on FHE data and CPA

B. Security analysis of search operation on FHE data

In this section, we consider an adversary $A'(s, D)$, which is capable of performing a search of an encrypted data s, over an encrypted database, D. The adversary A' returns 1 if the search is successful (that implies the search decision is correct), else returns 0.

Now we propose a CPA adversary (as shown in figure 7) A, which uses the above adversary as a sub-routine. First, let us see the challenge of A according to the experiment $PubK^{cpa}_{Adv,\Pi}(n)$. Let $m_0, m_1, m_2, \ldots m_n$ be the messages and A choose two arbitrary messages m_0 and m_1 among them. The adversary is provided with a challenge c which is corresponding to the encryption of m_b, where $b \in \{0, 1\}$. The adversary is asked to guess the bit b, ie. to predict whether the c corresponds to the encryption of m_0 or m_1. Note that if the encryption is randomized, the adversary A has a negligible probability of success, as repeating the encryption with either m_0 or m_1 using the encryption oracle has a probability of $1/2^{n-1}$ of matching with the challenge cipher c. Considering E as encryption scheme, r as the random number used during

challenge c generation and r' as the random number used by adversary, mathematically this can be explained as:

$$\begin{aligned} Pr(b=b') &= Pr(b=b'|b=0).Pr(b=0) + \\ &\quad Pr(b=b'|b=1).Pr(b=1) \\ &= \frac{1}{2}.[Pr(b'=0|c=E(m_0,r)) \\ &\quad +Pr(b'=1|c=E(m_1,r))] \end{aligned} \tag{2}$$

$$\begin{aligned} Pr(b'=0|c=E(m_0,r)) &= Pr(b'=0|c=E(m_0,r), \\ &\quad r'=r).Pr(r'=r) \\ &\quad +Pr(b'=0|c=E(m_0,r), \\ &\quad r'\neq r).Pr(r'\neq r) \\ &= 1.\frac{1}{2^n}+\frac{1}{2^n}(1-\frac{1}{2^n}) \\ &= \frac{1}{2^n}(2-\frac{1}{2^n}) \end{aligned} \tag{3}$$

Further, $Pr(b'=1|c=E(m_1,r))$ is also $\frac{1}{2^n}(2-\frac{1}{2^n})$, hence $Pr(b=b') \approx 1/2^{n-1}$.

Instead, the adversary A calls the adversary A' to answer the above challenge in the experiment $PubK^{cpa}_{A,\Pi}(n)$. The adversary A passes the challenge c and the ciphertext corresponding to plaintext m_0, and all the other plaintexts, but not m_1. The adversary A' thus is given a database D consisting of the encryptions of m_0 and all other messages (note that the ciphertext corresponding to m_1 is missing). Then the adversary A' performs the search of the challenge cipher c in the database and returns a response 0 or 1 according to the success or failure of the match. It is easy to check, that a success indicates that c is corresponding to the ciphertext of m_0. However, if A' returns 0, then the challenge c can correspond to either m_0 or m_1. Hence, A returns a random bit b' Thus, if A' returns a 1, A guesses $b'=0$.

Formally, we assume $Pr[A'(c,D)=1]=\epsilon(n)$, where $\epsilon(n)$ is assumed to be non-negligible. Now, $Pr(b=b')$ can be realized with the following equations:

$$\begin{aligned} Pr(b=b') &= Pr(b=b'|A'(c,D)=1).Pr(A'(c,D)=1) + \\ &\quad Pr(b=b'|A'(c,D)=0).Pr(A'(c,D)=0) \\ &= 1.\epsilon(n)+\frac{1}{2}.(1-\epsilon(n)) \\ &= \frac{1}{2}+\frac{1}{2}.\epsilon(n) \end{aligned} \tag{4}$$

The value of $Pr(b=b')$ thus violates the CPA security requirement as mentioned in equation 1. From this discussion it is evident that if an adversary can successfully determine the result of the search operation, the underlying encryption scheme is vulnerable to CPA attack. Then the question arises how to perform search over encrypted data without compromising CPA security. The answer is though search operation is being performed directly on encrypted data (without any decryption), the search result is also encrypted. This encrypted search result is directly delegated as the input to next operation (if the next operation is dependent on the search result) or it is send back to the client. Now the client can perform a final decryption and get the result of the search. Hence, for performing the search in the encrypted domain on the FHE data, the final result of the search has to be encrypted to ensure CPA security. In the next section, we discuss how encrypted search techniques can actually be performed using modules for performing homomorphic computations on FHE data.

V. Linear encrypted search on FHE data

Performing search operation on a database requires two steps: first the search item should to be compared with each of the existing data of the database (comparison step), second the judgment to be made based on the comparison decision whether the item is actually present in the database (decision step). However, the encrypted comparison and decision making should be realized with some encrypted homomorphic operations. In next section, we first discuss how the modules required for encrypted search can be designed using homomorphic operations and then explain the actual search algorithm using those modules. All the underlying modules are designed based on some underlying Fully Homomorphic primitive submodules present in Scarab library [15].

A. Required submodules for implementing search algorithm on FHE data

Our proposed search is performed based on comparing the search item with other database elements. Comparison basically depends on the subtraction results of FHE data, and then the postprocessing of the results. Here are the main submodules required for the algorithm:

- FHE subtraction FHE_Sub module.
- FHE bit inversion FHE_Inv module.
- FHE equality check FHE_Equal module.

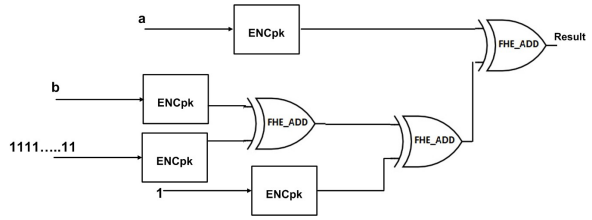

Fig. 3. Fully Homomorphic Subtraction

FHE subtraction module: FHE subtraction (FHE_Sub module as shown in figure 3) module outputs the subtraction result of two encrypted data. FHE_Sub module is built by performing FHE addition of one ciphertext with 2's complement of another ciphertext. The subtraction can be implemented by adding one number with the 2's complement of another. FHE addition module is designed based on bit-wise FHE addition module of Scarab library. Here, we show the basic operations of FHE subtraction circuit:

For two plaintext numbers a and b, subtraction can be computed as:

$$a-b = a \ + \ \text{2's complement of } b \tag{5}$$

Now, homomorphic subtraction between a' and b' (encryptions of a and b respectively) is computed using the homomorphic addition between a' and $Enc($ 2's complement of b).

The 2's complement of b in the encrypted domain is obtained as follows:
$Enc($ 2's complement of $b, pk)$
$= b' \oplus Enc(11\ldots1, pk) \oplus Enc(1, pk)$

Finally, the subtraction between a' and b' is computed as:

$$a' - b' = a' + Enc(\text{ 2's complement of } b) \quad (6)$$

FHE bit inversion: FHE_Inv module performs encrypted bit inversion by adding Enc(1) with the input bit (by FHE_add module). If the input bit is Enc(1), then addition of 1, results to encrypted 0. Again, if the input is encrypted 0, then the resultant is Enc(1) by addition.

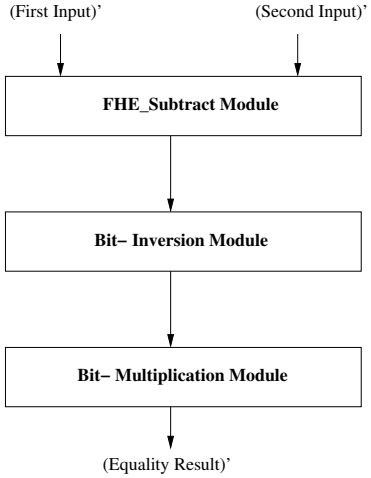

Fig. 4. Fully Homomorphic equality check

FHE equality check module: FHE_Equal module is designed to check if two FHE data are equal. If two operands are equal, then FHE subtraction (output of FHE_Sub module) of these two operands should result encrypted 0. To check whether the subtraction result is 0:

- Each of the bits of the subtraction result is inverted (using FHE_Inv module).
- Bitwise multiplication is performed on resultant inverted bits using FHE_Mul module present in Scarab library.
- If the subtraction result is nonzero (implies that two inputs are not equal), then atleast one bit of the subtraction result must be nonzero. Hence, the multiplication result becomes zero due to the presence of inversion of the nonzero bit (or bits).
- Thus, FHE_Equal module outputs Enc(1) if two FHE inputs are equal, else Enc(0) if two are unequal.

Now, in the next subsection we explain the search algorithm on FHE data using the above mentioned modules.

B. Linear search algorithm on FHE data

In this section, we outline how the search is actually performed on encrypted data using the homomorphic modules explained in section V-A:

- Let an encrypted data s is to be searched in a database D with n encrypted data items.
- Initially, FHE subtractions (using FHE_Sub module) are performed between s and each of the database items, let the encrypted subtraction results be $sub_0, sub_1, \ldots, sub_n$.
- Each subtraction result is checked if it is equal to 0 (using FHE_Equal module). Any subtraction result sub_i equals to 0 implies that s matches with the i-th item of the database. If no subtraction result equals to 0, it indicates s is not present in the database.
- However, it requires decryption of n outputs of FHE_Equal module to take decision about the search result at this stage. To avoid such large number of decryptions, we perform postprocessing on the outputs of FHE_Equal module in encrypted domain to generate a single bit encrypted search result. If the search result is Enc(1), it indicates the search item matches with one or more items present in the database.
- In the postprocessing phase, each of the outputs of the FHE_Equal module is inverted using FHE_Inv module and bit-wise FHE multiplication is performed among the inverted bits. If any of the outputs of FHE_Equal module equals to 1 (indicates match is found), then it inverts to bit 0. All other outputs of equality check becomes 0 and inverts to 1. Thus, final multiplication result bit becomes 0 (or 1), depending on any match is found or not.
- The inversion of this multiplication result is stored directly in the server as the encrypted search result. If the final multiplication result is 0, it indicates the search is unsuccessful and the intended data is not present in the database.

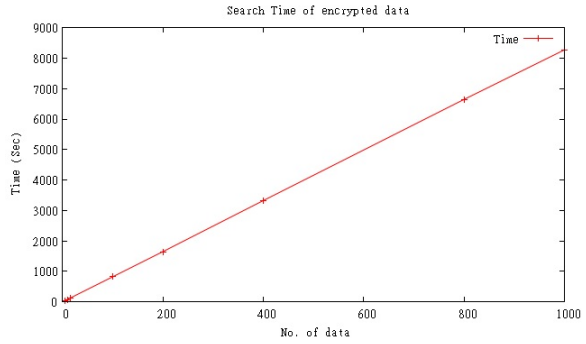

Fig. 5. Timing requirement for FHE Search

The above mentioned encrypted search algorithm has been evaluated for correctness on a Linux Ubuntu 64-bit machine with $i686$ architecture 1.6GHZ processor. Figure 5 shows the timing requirement for searching. The above linear search operation confirms the presence or absence of any data in the encrypted collection, but the final result of the search is also encrypted. In comparison to the method proposed in [29] of comparing encrypted ciphertexts using order preserving index, our method may be less efficient, but supports generalized FHE schemes present in [5].

Security Analysis

The proposed search algorithm on encrypted data meets the security requirement as explained in IV-B since all the

performed intermediate operations are encrypted and finally encrypted search result is produced. It may be noted here that one may be tempted to apply Divide and Conquer strategies like binary search to improve the timing of the above operation. However, as discussed in section VII, to ensure the CPA security of the scheme, the result of the search technique should not be known at any stage to the adversary. Thus the search algorithm cannot partition the entire database and perform selected search. Hence, classical algorithmic improvements are not directly applicable for operations on encrypted data. In the next section, we provide another example of operation on database, namely sorting and highlight the challenges of implementing partitioning on encrypted operations. Subsequently, we provide some techniques for accelerations which can be used in the FHE context.

VI. Sorting on Encrypted data

In this section, we shall describe the aspects of sorting on encrypted data. There are various types of sorting techniques on unencrypted data, and we classify them into two classes depending on their time complexities. It may be mentioned that all the following sorting algorithms are based on comparisons, but we classify them to emphasize that one type uses a divide and conquer strategy, while the other does not.

- **Simple Comparison based sorting** (like Bubble sort, Insertion sort etc) works in $O(n^2)$ time to sort n number of unencrypted data.
- **Divide and Conquer based sorting** (like Quick sort, Merge sort, Heap Sort etc) works in $O(nlogn)$ time to sort n number of unencrypted data. These class of sorting algorithms divide the problem into sub-problems and finally merges the solution in a recursive fashion.

Though sorting on data is an age old topic of research, much work has not been reported on sorting encrypted data. When the underlying data is encrypted by FHE to support arbitrary computations, new challenges and limitations are faced. A first attempt to realize simple comparison based sorting on FHE data was investigated in our prior work [12]. However, further security analysis and limitations to apply Divide and Conquer strategies to sort over FHE data were not mentioned. In this work, we provide first an overview on the techniques adopted for performing a simple comparison based sort and the timing results obtained thereof. Subsequently, we discuss the aspect of applying improved sorting algorithms like Quicksort to FHE data, and propose a method which we call as index encryption to achieve such operations. However we also present the limitations of applying such divide and conquer strategies to FHE data by showing explicitly that the ability to divide the problem (partition in case of Quicksort) is equivalent to define a successful CPA adversary against the FHE scheme. Finally, we propose a two stage sorting technique which proved to be effective to increase the performance of encrypted sorting.

A. Bubble and Insertion Sort on Encrypted Data

In this section, we consider two common sorting algorithms based on comparisons, namely Bubble and Insertion sorts on encrypted data. Comparison based sorting algorithms are based on conditional swap operations. When the data is encrypted this operation translates to a Fully Homomorphic Swap (FHS) operation. The FHS operation, denoted also as FHE_swap uses the following primitive circuits: FHE_add, FHE_mul, FHE_fulladd, and FHE_halfadd.

The FHS circuit depends on two main operations : subtraction operation (as explained in section V-A) and a multiplexer in the encrypted domain, denoted as FHE_Sub and FHE_mux respectively. These operations are built using the above mentioned primitive circuits. Figure 6 shows the overall swap operation in fully homomorphic domain.

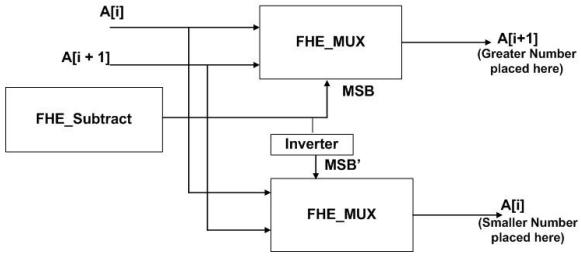

Fig. 6. Fully Homomorphic Swap

The FHE_swap operates on two encrypted arguments, denoted as A_0 and A_1. It compares them by performing a FHE_Sub and then uses the encrypted result of the comparison to multiplex out a ciphertext which is homomorphically greater than the other. Let, the greater one is stored in B_1, while the smaller one is stored in B_0. However, the logic is done in the encrypted domain in such a way that it is emphasized that the final results of the swap B_0 and B_1 may not be the same as the inputs A_0 and A_1. But they are homomorphically same, implying that the plaintext corresponding to them are same. This happens again due to the randomization of the encryption scheme. Finally, the following equations represent how the swap operation takes place between two elements A_0 and A_1 depending on the encrypted subtraction result of A_1 and A_0. Thus, $bt = MSB(FHE\_Sub(A_0, A_1))$, where MSB is the maximum significant bit of the subtraction result, which is also in the encrypted format. The equations are:

$$\begin{aligned} A_1 &= FHE\_MUX(A_1, A_0, bt) \\ A_0 &= FHE\_MUX(A_1, A_0, (1 - bt)) \end{aligned}$$

The detailed design of FHE_MUX is explained in [34], [35]. Having discussed the implementation of a comparison based sorting, we make an analysis on the aspect of implementing sorting through divide and conquer. We address this issue of applying recursive algorithms to FHE data, by considering partitioning based sorts like Quicksort, which provide a significant speed up in case of sorting unencrypted data. However, we show that in case of FHE data unique challenges are encountered. However, to the best of our knowledge there is no analysis present in the literature about performing partition based sort on encrypted data. Further, from the existing knowledge of partition based sort on unencrypted data, it is evident that partition based sort is expected to be more efficient over comparison based sort. Our research first analyzes how far the

encrypted partition based sort meets this expectation pertaining to the properties of FHE data. In other words, our work first explains whether it is really possible to achieve performance gain by partition based sort (over encrypted comparison sort) maintaining the security of the cryptosystem. Hence, in the next section we first analyze the feasibility of implementing partition based sort on FHE data in the light of security and then explain our proposed schemes of actually implementing partition sort on encrypted data.

VII. Sorting Based on Divide and Conquer: Partition based Sort

In this section we first analyze the feasibility of implementing partition based sort on FHE data in the light of security and then explain our proposed schemes of actually implementing partition sort on encrypted data.

A. Security analysis of partition based sorting in fully homomorphic domain

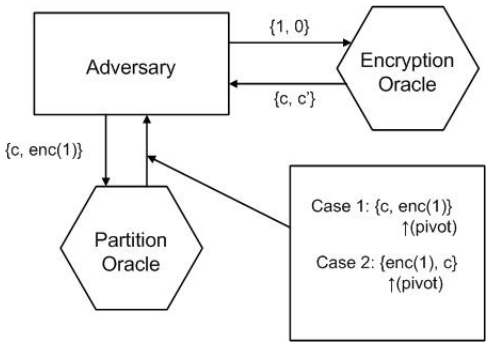

Fig. 7. CPA due to partition based FHE sorting

In this section, we shall investigate the security implications of performing partition based sort on FHE data. Basically, we examine whether the FHE scheme remains secured against Chosen-Plaintext Attacks (CPA) (discussed in section IV-A) if it is possible to perform partition based sort on encrypted data. Considering Quick sort as an example of partition based sort, encrypted quick sort will be discussed in the rest of the paper. Quick sort algorithm requires three steps:

- Pivot selection.
- Comparison of each element with the pivot and decision making to partition.
- Perform sorting recursively on each of the partition the array.

Now, we shall explain quick sort in the light of CPA indistinguishability experiment following the set-up explained in [33] and show how the possibility of performing encrypted partitioning may lead to the cryptosystem being vulnerable to CPA attacks. More formally the situation will be explained with figure 7 and the following steps:

- Let an adversary Adv be capable of performing partition based sort on an encrypted database D and returns the final sorted array.
- Figure 7 shows that that in such scenario adversary Adv has access to a **partition oracle** and an **encryption oracle**. **Encryption oracle** returns the encryption of any message, once the message is given as an input. The **Partition oracle** takes as input an array of encrypted data and a pivot element, and returns two partitions of the input array. One partition contains all the elements lesser than the pivot element and other partition holds all elements equal to or greater than the pivot element.
- Now, as before the adversary Adv receives a challenge c which is the encryption of either the message say m_0 or m_1. The adversary wins if he is capable of guessing correctly whether c is the encryption of m_0 or m_1, ie. he has to guess the bit b with a probability non-negligibly higher than $1/2$, where $Enc(pk, m_0) = c$ (where pk is the public key). For convenience we write $m_0 = 0$ and $m_1 = 1$, i.e $c = Enc(b)$ where $b \in \{0, 1\}$.

Now, we define such a successful adversary Adv, which uses a partition oracle. The partition oracle can obtain partition based sorted array on encrypted data (without any single decryption). However, that indicates that the decision of partitioning (whether one encrypted data is lesser or greater compared to encrypted pivot) is not hidden to the Adv. In this scenario, adversary can send $\{c, Enc(pk, 1)\}$ to the partition oracle and make it as pivot. Now, the partition oracle can return two results:

- Case 1: $\{c, \underbrace{Enc(1)}_{\text{pivot}}\}$ indicates c is lesser than the pivot (note that ordering is with respect to plaintext) and clearly it is the encryption of 0.
- Case 2: $\{\underbrace{Enc(1)}_{\text{pivot}}, c\}$ indicates c is equal to or greater than the pivot and clearly it is the encryption of 1.

Based on the response of the partition oracle, the adversary is thus capable to easily guess the bit b, ie. the plaintext 0 or 1 which resulted in the challenge ciphertext c. Hence, it is evident that the cryptosystem is prone to chosen plaintext attack if it is possible to perform partition based sorting on encrypted data in the traditional way of comparison with pivot (as done on unencrypted data). It is interesting to note here that the comparison based sort is not vulnerable to the CPA attack. We clarify and stress this point in the next section.

B. Why comparison based sorting is secured?

Comparison based sort on FHE data is performed using fully homomorphic conditional swap (FHS) operation. The FHS circuit depends on two main operations: subtraction operation and the homomorphic multiplexer. It is interesting to note that the capability to perform the above partition based sort can lead to a CPA adversary, while such a reduction is not possible for a comparison based sort. Close observation of the homomorphic swap operation explains why this sorting does not lead to the CPA adversary. The crux of this lies in the fact that it is never disclosed to the adversary which of the two inputs to the block FHE_swap (FHS) is greater. Since, all the elements that are fed as the input to the swap circuit are *modified* by the FHE operations in the FHE primitive circuits, changes also take place in the input ciphertext, ofcourse ensuring that they correspond to the same plaintexts. The

security of the FHE scheme is retained since an adversary is incapable of collecting the information whether the swap operation is really taking place or not. Note that the adversary cannot tally the outputs with the inputs, as they are changed and the equality is only in the plaintexts which is hidden to the attacker. This makes comparison based sort secured.

The basic difference between comparison based sort and partition based sort lies in their working techniques. Comparison sort works fine in the encrypted domain because it is not required to know if one given input is homomorphically greater (or smaller) than another input, but still one can place the greater elements and the smaller elements in their correct positions. However, this information is necessary in the partitioning phase of quicksort. In this phase of quicksort, one compares the elements of the array with the pivot and increases or decreases the running indices based on the results of the comparisons. However, in homomorphic domain this is not feasible as the FHS step never reveals whether the swap took place, and also changes the values of the inputs. Hence, the array of encrypted numbers cannot be partitioned in the classical sense. On the contrary, the ability to partition in the classical sense implies that the underlying FHE data can be subjected to a CPA attack.

In the next section, we take a fresh look at partitioning and make an attempt to realize quicksort on FHE data. To be more precise, the limitation in applying the partitioning to FHE can be alleviated if one also encrypts the index of the array. This layer of encryption helps one to hide the position of the pivot and thus although the partitioning happens the exact pivot index is hidden to the adversary. In the next section, we detail the idea of index encryption and explain how to use it to realize quicksort.

VIII. Partition based sorting with index encryption

In this section, we shall discuss the steps to perform partition based sort on an array of FHE data. The essential function for performing quick sort is as follows:

```
void quickSort(array, first, limit)
{
    if (first < last)
    {
        p = partition(A, first, last);
                 /* Partitioning index */
        quickSort(A, first, p - 1);
        quickSort(A, p + 1, last);
    }
}
```

The above code follows standard recursive manner by which quicksort on unencrypted data is implemented. The essential functions are partition and two recursive calls to the quicksort routine itself. Recursive codes are realized on the system stack, which performs two operations push and pop to maintain the activity chart of the program. Now consider the situation when the array is encrypted. As discussed, the position of the pivot is also computed in an encrypted fashion, and thus the limits of the arrays (which are the indices) are also stored in an encrypted fashion. Further note, that the system stack is unable to realize the above recursion. This is because the stack stores the start (left) and the end (right) indices in an encrypted format. But the decision to pop or push, ie. to decrement or increment the stack pointer depends on encrypted data: ie. the encrypted pivot position and the left or right index, both of which are also encrypted. Thus the address of the stack also needs to be encrypted, and hence one needs to develop a user-defined encrypted stack to realize a FHE-quicksort. In the following, we provide an overview starting with an organization of the encrypted array with an encrypted index.

A. Encrypted array with encrypted index

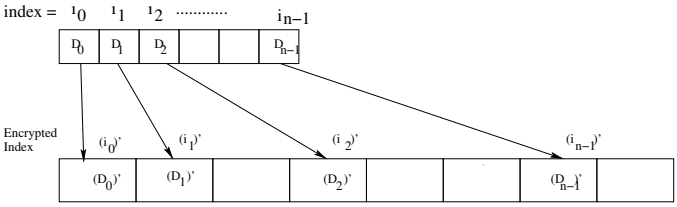

Fig. 8. Encrypted array with encrypted indices

Figure 8 shows how an array A is mapped to its encrypted form. The base address of A is encrypted (by FHE encryption) and that is considered as the base address of the encrypted array. The next encrypted locations are obtained by performing FHE additions to the previous locations. For example, enc(1) is added (by FHE addition) to the encrypted i-th location address $((i)\prime)$ and encrypted address of $i+1$-th location is obtained. However, due to the randomness property of FHE encryption, the consecutive unencrypted addresses do not remain consecutive after encryption. Finally, data of i-th location (say D_i) is placed to $(i)\prime$ location as $(D_i)\prime$ in the encrypted array.

This encrypted array is chosen as the data structure for implementing partition based quick sort on encrypted data. Now, we shall decide the implementation approach of the algorithm. Recursive and iterative are the two different practiced implementation approaches of performing quick sort. Here, we first discuss what are the problems of implementing recursive methods on encrypted data and then explain our proposed scheme handling such implementation challenges.

B. Problems of recursion on encrypted data

In general, recursive implementations are very popular for partition based sort on unencrypted data from design point of view. However, it is required to specify the initialization or termination condition of recursion. While working with encrypted data and encrypted indices, the initialization or termination conditions of recursion are the results of encrypted FHE operations (by homomorphic modules). However, present underlying processors are unencrypted and unable to process such encrypted recursion conditions. Hence, direct implementation of recursion methods are not possible in such existing processors.

In case of recursive implementation of partition sort, intermediate partition parameters are stored in recursion stack. In our approach of handling encrypted sort, an encrypted auxiliary stack is designed and encrypted partition indices values are stored in this encrypted stack. Now, we shall outline the design of an encrypted stack which is capable of handling encrypted push pop operations and extend our idea on how to use it implementing partition based sort. Here, we define the encrypted stack with following operations:

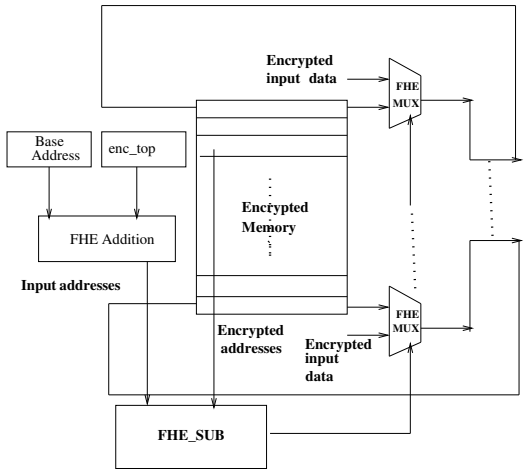

Fig. 9. Encrypted Push Operation

Encrypted Push operation:: Push is responsible for both initialization and data insertion to stack. At initialization of stack, stack size (mentioned by enc_top) is Enc(0).

During data insertion (Push operation), the encrypted base address of stack is increased by enc(1) each time and encrypted data is stored in the next address. enc_top is increased accordingly to hold the index of the top of the stack (which indicates the size of the stack too). All these increment operations are again homomorphic and take place using FHE addition operations. Figure 9 shows the push operation of encrypted data in top location of encrypted stack. Here enc_top is added with encrypted base address value and data is pushed when address match is found.

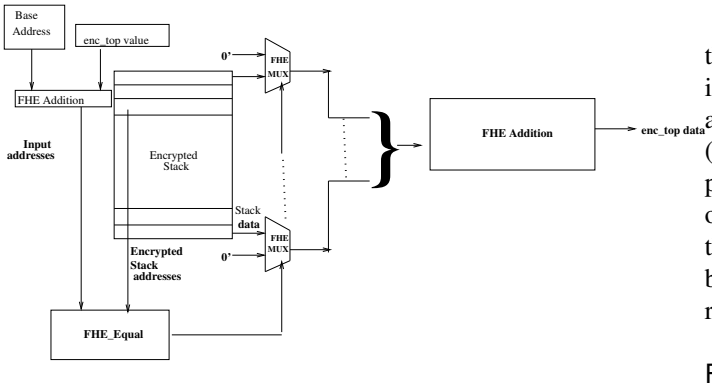

Fig. 10. Encrypted Pop Operation

Encrypted Pop operation:: Pop operation in such encrypted stack is even more interesting since all encrypted data are now residing in encrypted addresses. Now, during pop operation, the stack_top decreases and (when added to the base address) gives the modified address from where data should be popped. However, the challenge is this modified address value is not exactly bit-wise same with the existing stack locations. This creates the real ambiguity in case of fetching (or popping) data from an encrypted stack. Here we use the encrypted multiplexer to check which encrypted stack location is homomorphically equal to the modified address. If a match is found, data is popped from that particular location. However, encrypted multiplexer gives an encrypted result (address matched or not) and hence the location (from which data is actually fetched) remains hidden from any adversary. Figure 10 shows the pop operation of encrypted data from top location of encrypted stack. Here enc_top is added with encrypted base address value and data is popped when address match is found.

Error handling:: During handling of any stack, two error conditions are possible. Firstly, attempting to push data when the stack is full and the secondly trying to pop data when the stack is empty. Both these error conditions can be handled by comparing the existing stack size with maximum stack size (to check whether the stack is full) or with enc(0) (to check whether the stack is empty).

This encrypted stack is used to store the start and end indices of the partition and to continue quick sort in sub-arrays.

C. Quicksort using Encrypted Stack

The quicksort on the encrypted data (FHE-quicksort) where the array indices are also encrypted is next described. As discussed, the data is encrypted and stored in an array with encrypted index locations. The FHE-quicksort function has the following arguments:

```
fhe_qsort(encarray, encfirst, enclimit, pk)
```

The arguments are respectively the encrypted array, the encrypted starting address of the array, and the encrypted end address of the array and the public key pk. As discussed the implementation is done using an encrypted stack, which stores the intermediate partition parameters. Here, we first describe the function to realize the partition.

Encrypted partition: The partition function takes as input the encrypted array, along with the encrypted left and right indices. The initial pivot can be chosen as the first, middle or any random element of the input array. However, the last index (encrypted) has been chosen here. The final position of the pivot is determined by encrypted comparisons. A crucial step of the partitioning algorithm is to compare homomorphically the encrypted pivot pv data with the encrypted data pointed by an encrypted running index, $j\prime$. Based on the comparison result, another encrypted running index $i\prime$, is incremented.

This comparison is done using the comparison circuit FHE_isgreater (as shown in figure 11), which operates on two encrypted values. The two encrypted values are subtracted and MSB of the subtraction result is fed as the input of the comparison circuit. If the MSB is Enc(1), the first input is lesser than the second, else otherwise.

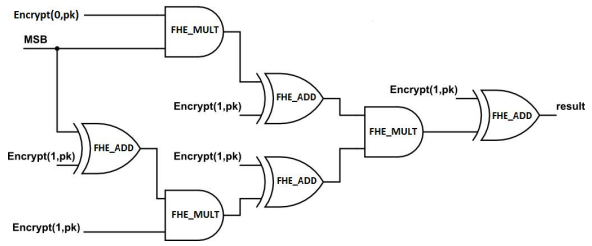

Fig. 11. Fully Homomorphic greater comparison

The comparison result between enclimit and the encrypted index jIndex homomorphically sets the value of a variable cond_loop depending on the result of the comparison. The encrypted bit *cond_loop* decides whether $i\prime$ will be incremented or hold the previous value. Likewise, the encrypted value stored in the encrypted address jIndex, denoted as data_indexj, is compared with the data of the encrypted pivot, data_pivot and another condition, cond2 is set homomorphically. Finally, both the conditions are homomorphically multiplied to generate the signal cond. The update of the index $i\prime$ is done through a homomorphic multiplexer, denoted as FHE_mux which takes the encrypted value of i, iIndex and the $(i+1)\prime$, iIndex_incr as inputs. The selection among the two inputs is done by the encrypted value cond. The code snippet is as follows:

```
fhe_isgreater(cond_loop, enclimit,
                                jIndex, pk);
fhe_isgreater(cond2, data_pivot,
                          data_indexj, pk);
fhe_mul(cond, cond_loop, cond2, pk);
fhe_mux(mod_i, iIndex,
                    iIndex_incr, cond, pk);
```

Thus this continues until the correct position of the pivot is determined in the encrypted form. However, since all the decisions are performed in the encrypted format, one cannot decide on the termination condition without decrypting. Hence, the pivot based partitioning, although performed homomorphically from the left index of the array (or sub-array in the recursive calls), to the right index, is always executed on the complete array length of maximum number of data to be sorted. This is a major bottle-neck in terms of performance, and overall time-complexity.

D. The Encrypted Quicksort through Encrypted Stack

As stated before, the overall sorting is performed using a user-defined stack, where both the content and the address are encrypted. The stack stores the encrypted end limits of the array portions currently being sorted (denoted as $l = encfirst$ and $h = enclimit$). *Encrypted Push operation* is responsible for both initialization and data insertion of the encrypted array limits to the stack. For initialization of the stack, an address is encrypted and that is considered as the starting base address of stack. Stack size (mentioned by encrypted stack_top) is Enc(0) at this point.

The sorting function at each iteration pops out the end limits of the encrypted array. The partition function (discussed in the previous section) provides the encrypted pivot position, p. Subsequently, the encrypted array indices $p-1$ and l are homomorphically compared to check $p-1 > l$ and if true, the stack is pushed with the data of l and $p-1$. Note the decision to push the stack depends on an encrypted comparison, and thus implying why the stack addresses also needs to be encrypted. Likewise, we perform the pushing of the encrypted index $p-1$. Furthermore, we perform the check whether $p+1 < h$, and in a similar manner push the indices $p+1$ and h. Both the push and pop stack operations are encrypted as mentioned in section VIII-B.

In the quick sort, push or pop operation to or from the stack continues till $(stack\_top \geq 0)$. However, implementation of this step (in encrypted quick sort) requires a comparison between encrypted $stack\_top$ and Enc(0), which in turn generates an encrypted result. To solve this issue, the entire loop is run for the size n of the array, where n data is being sorted (again we cannot terminate depending on the stack being empty). The result is still functionally correct, as there will be redundant operations which does not make any change to the array in the homomorphic sense. Likewise for partitioning, each time the run is done over the entire array length. However, for a given call to the partition function, thus sorting happens from the running index l to h (the current left and right indices of the array), for ranges outside this limit, no change or sorting is done in the homomorphic sense. This ofcourse has an adverse effect on the time complexity, and in fact provides bad timing than the comparison based sorts, because of increased stack operations. However, as discussed, this redundancy is mandatory since without decryption one cannot reveal the index position of the stack or determine the termination condition. Finally, the proposed sorting algorithm on encrypted data meets the security requirement as explained in VII since all the performed intermediate operations are encrypted and an encrypted sorted array is produced as the final result. In the next section, we formally analyze the timing requirements of our proposed sorting schemes (comparison as well as partition based schemes) to decide which encryption scheme is most suitable while working on encrypted data.

TABLE I
COMPARISON OF DIFFERENT SORTINGS

Sorting	No. of elements	Average Timing requirement (sec)
Bubble Sort	5	235
	10	1527
	40	21565
Insertion Sort	5	290
	10	1602
	40	21757
Quick Sort	5	776
	10	4102
	40	46757

IX. TIMING REQUIREMENT FOR SORTING SCHEMES ON ENCRYPTED DATA

In this section, we compare different proposed sorting techniques on encrypted data in terms of their time complexities.

TABLE II
ESTIMATE OF DIFFERENT SUBOPERATIONS FOR SORTING

	Operation	Overhead(sec)
1.	FHE add(XOR)	0.04
2.	FHE mul(AND)	0.05
3.	FHE recrypt	0.04

TABLE III
TIMING ANALYSIS FOR LAZY SORT

Number of data	Fully homomorphic sorting Time (sec)	Lazy sort time (sec)		
		W=2	W=6	W=10
10	1527	923	1623	2150
20	5300	3092	4292	5492
30	11980	6507	8307	9065
40	21750	11350	13544	15972

We consider to sort n data stored in the cloud and encrypted with FHE scheme.

To perform encrypted comparison sort on this database we need to have n encrypted comparisons in each of the n iterations as explained in section VI-A. Hence, as a worst case analysis, it requires n^2 comparisons. Let the time for each encrypted comparison be T_c and the total time for comparison sort T_{comp} can be computed as $T_{comp} = n^2 * T_c$.

For performing quick sort on encrypted data as explained in section VIII, partitioning and stack handling are the two main operations. Let the partitioning time be T_p and the stack handling time be T_s and hence for each iteration the sorting time is $T_p + T_s$. Now, as stated above iteration count depends on push or pop operations to or from the stack continues till $(stack\_top \geq 0)$ and to handle the difficulty of program termination, this loop iterates for n times (and not terminate depending on the stack being empty) maintaining the correct functionally. Hence, at this stage the total time requirement for quick sort $T_{quicksort}$ becomes $T_{quicksort} = n * (T_p + T_s)$.

Now, for each partition operation let the main array is divided into two subarrays. Now, each of the elements of each partition need to be compared with pivot, but the comparison loop should iterate for n times (maximum length of main input array) for each partition sub-array (since the actual partition index is encrypted and actual partition length is not known). Hence, for each iteration partition time becomes $T_p = 2{*}n{*}T_c$ and the overall time becomes:

$$T_{partition} = n * (T_p + T_s) \quad (7)$$
$$= n * (2 * n * T_c + T_s) \quad (8)$$
$$= 2 * n^2 * T_c + n * T_s \quad (9)$$

Hence, the time complexity of quick sort on encrypted data is no better than comparison sort. Further, the stack operation adds extra overhead to this timing requirement. Since, the required time for encrypted push and pop operations proportionally increases with the increase of the stack size, this stack timing overhead T_s is also very high. Actual timing requirements of the sorting schemes presented in table I also confirms our analysis in this issue.

However, as can be observed that the average time for performing the encryptions increases quadratically with the number of elements. Now, with the objective of improving the performance of comparison sort, we investigate the cause of such a timing requirement. From table II, we observe that the underlying recrypt operations are the main reason for such a timing requirement (since this operation need to be executed after each of the homomorphic operations to reduce the noise level). We can observe from the table that the timing requirement of FHE_add (which includes time for addition operation + time for recrypt) is almost equal to timing requirement of recrypt operation. Further, timing requirement of FHE_mul (which includes time for multiplication + time for recrypt) also confirms that the reason of large timing requirement of FHE operations is mainly due to the presence of recrypt operation, which is the main denoising operation of FHE scheme. Next, we introduce a method called lazy computations, where the sorting is performed in two phases. In the first phase we remove recrypts carefully to result in an almost correct output with some errors present due to the removal of recrypt operations. The first phase, which we called a lazy phase is followed by another phase where the error is removed. In the next section, we explain the lazy sort, where the first step is erroneous but fast. The second step is precise but uses suitable sorting algorithm, which performs well on almost sorted data.

X. LAZY SORTING

In this section we introduce the concept of lazy-sorting, which we propose as an effective way of reducing the overheads due to the costly recrypt operations. In context to this lazy iterative sorting, the overall process is divided into two steps: the first applies Bubble sort with reduced recrypt operations. The second phase applies Insertion sort on the resultant array. The rationale behind such an approach is that the selected removal of recrypt operation introduces error in the results. But if the removal of recrypt is done carefully, the error is controlled resulting in an almost sorted array, which can be sorted efficiently by the Insertion Sort phase, which runs in linear time on nearly sorted data.

Thus, minimization of recrypt after a certain threshold may introduce some error in the comparison decision (swap) of fully homomorphic encrypted data. Hence, this will in turn introduce some error in the sorting decision. The term **error** indicates an element is placed in wrong position in the final sorted array. Now, if we perform any erroneous comparison sort with minimized recrypt, it will take comparatively less time due to the use of comparison circuit with reduced number of recrypts and results an almost sorted array. Finally, we apply insertion sort, which works in linear time for an almost sorted array.

Before introducing error on encrypted data with the reduction of recrypt, we perform an analysis on unencrypted data to observe how much erroneous swaps can be tolerated to result in an almost sorted array. In this experiment, we have

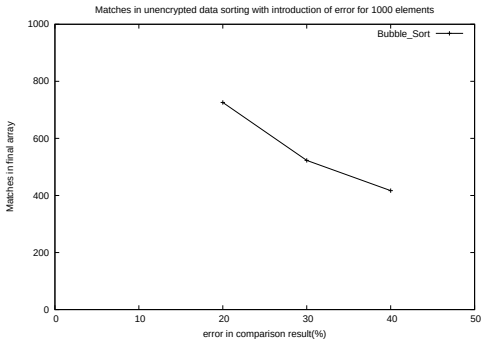

Fig. 12. Allowable Error analysis on unencrypted data

introduced a few percentage of wrong comparisons to observe how much error is introduced in the resultant array. The plot in figure 12 shows that with an around 30% error in comparison, 60% data are placed in the proper position. This is an average of several experiments with more than 1000 data, which shows around 700 data are properly placed by performing Bubble sort with reduced recrypt.

Futher reduction of recrypt to introduce error

Now, according to the trend in figure 12, it is evident that it is not possible to remove all the recrypts since it will introduce 100% error and result in positioning large number of elements in the wrong places of the final array. For this reason, careful choice of removable recrypts are necessary. We have identified the recrypts present in FHE_subtract, which is one of the main submodule of FHE_swap. In this function, we have removed the recrypt operations required to correct the values of carry bit ($cout$) of the addition result using FHE_fulladd. This in turn reduces the time requirement in every iteration of addition and finally reduces the time for swap operation. However, use of this modified swap operation results correct output with 70% accuracy. This results in an almost sorted array. Subsequently, we apply insertion sort, which has linear time complexity for almost sorted array. Table III shows the advantage of this scheme. Lazy sort approach helps to improve the performance significantly, where general Bubble sort in homomorphic domain requires around 21500s. The table shows with increased value of window required time for sorting increases. In our experiment, we obtain fully sorted array with window size $w = 10$.

XI. Experimental details

All the above mentioned sorting algorithms have been evaluated for correctness on a Linux Ubuntu 64-bit machine with $i686$ architecture 1.6GHZ processor using the FHE modules of Scarab library [15]. We have chosen arbitrary integer data (not structured data as mentioned in [29]) and encrypted with encryption algorithm of Scarab library. We see from our results that techniques like divide and conquer (although theoretically feasible) on FHE data, does not improve the performance while working on unencrypted processor. On the contrary, because of the excessive book-keeping and stacking operations, it has an adverse effect on the performance. Thus, the best option seems to restrict on iterative sorting algorithms.

Experimental challenges

Among different sorting techniques, implementation of bubble sort on FHE data is quite straightforward. In spite of the fact that Bubble sort and Insertion sort both are comparison based, there is some basic difference in their implementation techniques. In each iteration of insertion sort, single element is inserted in the proper position. Thus any element is placed in k-th location provided all the previous $k-1$ elements are smaller (or greater). Thus, handling of an encrypted while loop is necessary which imposes design challenge while working on unencrypted processor. In this case, we consider a special window based technique and assume that insertion position of the any element lies within an window. While performing insertion sort on arbitrary data set, this window is considered to be whole array length. However, in case of implementing Lazy sort, where insertion sort is applied on almost sorted array a smaller window size is sufficient to result a fully sorted array (as shown in table tb:sortwthminRecrypt).

XII. Conclusion

The application of Fully Homomorphic Encryptions (FHE) to cloud computing is a coveted goal. Inspite of its promises, there are several challenges of implementing operations on FHE data. The paper deals with the problems of searching and sorting as two common operations on encrypted database. We formally relate the ability to search in encrypted database to a chosen plaintext adversary and develop a technique for performing search on array encrypted with FHE. Subsequently, the paper shows methods to perform comparison based sorting on encrypted data. The work also addresses the application of common algorithmic techniques like divide and conquer, by taking an example of Quick sort and shows several challenges of applying such algorithms on FHE data. Relating to CPA resistance, the article shows that although theoretically feasible, the division or partition step on encrypted data does not lead to any advantage due to the inability to detect the exact partition index. Further, the paper presents several new data structures like encrypted array with encrypted index, encrypted stack and accompanying push and pop operations to realize recursive programs on encrypted data. Time complexity and simulation results have been provided to show while the methods are correct, the performance need to be improved. Finally, we propose a technique called Lazy sort, which shows a significant improvement in performance over traditional comparison and partition based sort on encrypted data. To the best of our knowledge, our work has made the first attempt of implementing such operations on FHE unstructured data and to our belief, this work leaves a hope that with further tricks (like further reduction of costly recrypt operations and suitable hardware accelerations) FHE computations can be indeed practical for cloud computing.

REFERENCES

[1] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Foundations of Secure Computation, Academia Press*, pp. 169–179, 1978.

[2] C. Gentry, "Computing arbitrary functions of encrypted data," *Commun. ACM*, vol. 53, no. 3, pp. 97–105, Mar. 2010.

[3] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," *IACR Cryptology ePrint Archive*, p. 616, 2009.

[4] J.-S. Coron, A. Mandal, D. Naccache, and M. Tibouchi, "Fully homomorphic encryption over the integers with shorter public keys," in *Proceedings of the 31st annual conference on Advances in cryptology*, ser. CRYPTO'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 487–504.

[5] M. Naehrig, K. Lauter, and V. Vaikuntanathan, "Can homomorphic encryption be practical?" in *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, ser. CCSW '11. New York, NY, USA: ACM, 2011, pp. 113–124.

[6] Y. Ramaiah and G. Kumari, "Towards practical homomorphic encryption with efficient public key generation," *ACEEE International Journal on Network Security*, vol. 3, no. 4, p. 8, October 2012.

[7] M. Akinwande, "Advances in homomorphic cryptosystems." *J. UCS*, vol. 15, no. 3, pp. 506–522, 2009.

[8] D. Stehle and R. Steinfeld, "Faster fully homomorphic encryption," Cryptology ePrint Archive, Report 2010/299, 2010, http://eprint.iacr.org/.

[9] H. Perl, Y. Mohammed, M. Brenner, and M. Smith, "Fast confidential search for bio-medical data using bloom filters and homomorphic cryptography." in *eScience*. IEEE Computer Society, 2012, pp. 1–8.

[10] ——, "Privacy/performance trade-off in private search on bio-medical data," *Future Generation Computer Systems*, 2014.

[11] S. Rass and D. Slamanig, *Cryptography for Security and Privacy in Cloud Computing*. Norwood, MA, USA: Artech House, Inc., 2013.

[12] A. Chatterjee, M. Kaushal, and I. S. Gupta, "Accelerating sorting of fully homomorphic encrypted data," in *Proceedings of the 14th International Conference on Cryptology in India*, ser. INDOCRYPT '13, 2013 (Accepted).

[13] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, 2009, crypto.stanford.edu/craig.

[14] D. Stehle and R. Steinfeld, "Faster fully homomorphic encryption," Cryptology ePrint Archive, Report 2010/299, 2010, http://eprint.iacr.org/.

[15] https://github.com/hcrypt project/libScarab.

[16] *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011.

[17] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive verifiable computing: Outsourcing computation to untrusted workers," in *Proceedings of the 30th Annual Conference on Advances in Cryptology*, ser. CRYPTO'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 465–482. [Online]. Available: http://dl.acm.org/citation.cfm?id=1881412.1881445

[18] S. Bugiel, S. Nürnberger, A.-R. Sadeghi, and T. Schneider, "Twin clouds: Secure cloud computing with low latency," in *Proceedings of the 12th IFIP TC 6/TC 11 International Conference on Communications and Multimedia Security*, ser. CMS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 32–44. [Online]. Available: http://dl.acm.org/citation.cfm?id=2046108.2046113

[19] A. reza Sadeghi, T. Schneider, and M. Win, "Token-based cloud computing secure outsourcing of data and arbitrary computations with lower latency," Workshop on Trust in the Cloud, 2010.

[20] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, ser. SP '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 44–. [Online]. Available: http://dl.acm.org/citation.cfm?id=882494.884426

[21] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *J. ACM*, vol. 43, no. 3, pp. 431–473, May 1996. [Online]. Available: http://doi.acm.org/10.1145/233551.233553

[22] A. C. Yao, "Protocols for secure computations," in *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, ser. SFCS '82. Washington, DC, USA: IEEE Computer Society, 1982, pp. 160–164. [Online]. Available: http://dx.doi.org/10.1109/SFCS.1982.88

[23] M. Bellare, A. Boldyreva, and A. O'Neill, "Deterministic and efficiently searchable encryption," in *Proceedings of the 27th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 535–552. [Online]. Available: http://dl.acm.org/citation.cfm?id=1777777.1777820

[24] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano, "Public key encryption with keyword search," in *EUROCRYPT*, 2004, pp. 506–522.

[25] K. Kurosawa and Y. Ohtaki, "Uc-secure searchable symmetric encryption." in *Financial Cryptography*, ser. Lecture Notes in Computer Science, A. D. Keromytis, Ed., vol. 7397. Springer, 2012, pp. 285–298.

[26] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: Improved definitions and efficient constructions," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, ser. CCS '06. New York, NY, USA: ACM, 2006, pp. 79–88. [Online]. Available: http://doi.acm.org/10.1145/1180405.1180417

[27] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries."

[28] E. Shen, E. Shi, and B. Waters, "Predicate privacy in encryption systems," in *Proceedings of the 6th Theory of Cryptography Conference on Theory of Cryptography*, ser. TCC '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 457–473.

[29] D. Liu, E. Bertino, and X. Yi, "Privacy of outsourced k-means clustering," in *9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014*, 2014, pp. 123–134.

[30] D. Stinson, *Cryptography: Theory and Practice,Second Edition : section 5.9.1*, 2nd ed. CRC/C&H, 2002.

[31] G. S. Çetin, Y. Doröz, B. Sunar, and E. Savaş, "Low depth circuits for efficient homomorphic sorting," Cryptology ePrint Archive, Report 2015/274, 2015.

[32] F. Baldimtsi and O. Ohrimenko, "Sorting and searching behind the curtain," in *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers*, 2015, pp. 127–146.

[33] J. Katz and Y. Lindell, *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007.

[34] M. Brenner, H. Perl, and M. Smith, "How practical is homomorphically encrypted program execution? an implementation and performance evaluation," in *11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2012, Liverpool, United Kingdom, June 25-27, 2012*, 2012, pp. 375–382.

[35] A. Chatterjee, M. Kaushal, and I. Sengupta, "Accelerating sorting of fully homomorphic encrypted data," in *Progress in Cryptology - INDOCRYPT 2013 - 14th International Conference on Cryptology in India, Mumbai, India, December 7-10, 2013. Proceedings*, 2013, pp. 262–273.

Ayantika Chatterjee received her B.Tech. degree from WBUT, India and the MS degree from School of Information Technology, Indian Institute of Technology Kharagpur, India. She also worked at Wipro Technologies, Bangalore. She is an ongoing PhD student in School of Information Technology, IIT Kharagpur. Her research interest includes Cryptography and VLSI design.

Indranil Sengupta presently working as Professor in the Department of Computer Science and Engineering, and the Managing Director of Science and Technology Entrepreneurs Park (STEP, IIT Kharagpur, India. Backed by teaching and research experience of 26 years, research interests include cryptography and network security, VLSI design and testing, and reversible/quantum computing. Previously the Heads of the Department of Computer Science and Engineering and also the School of Information Technology, IIT Kharagpur, Prof. Sengupta has several research contributions in different international jounals and conferences.