FURISC: FHE Encrypted URISC Design

Ayantika Chatterjee, Indranil Sengupta
cayantika@gmail.com, isg@iitkgp.ac.in

***Abstract*—This paper proposes design of a Fully Homomorphic Ultimate RISC (FURISC) based processor. The FURISC architecture supports arbitrary operations on data encrypted with Fully Homomorphic Encryption (FHE) and allows the execution of encrypted programs stored in processors with encrypted memory addresses. The FURISC architecture is designed based on fully homomorphic single RISC instructions like *Subtract Branch if Negative* (SBN) and *MOVE*. This paper explains how the use of FHE for designing the ultimate RISC processor is better in terms of security compared to previously proposed somewhat homomorphic encryption (SHE) based processor. The absence of randomization in SHE can lead to Chosen Plaintext Attacks (CPA) which is alleviated by the use of the FHE based Ultimate RISC instruction. Furthermore, the use of FURISC helps to develop fully homomorphic applications by tackling the *termination* problem, which is a major obstacle for FHE processor design. The paper compares the MOVE based FHE RISC processor with the SBN alternative, and shows that the later is more efficient in terms of number of instructions and time required for the execution of a program. Finally, an SBN based FURISC processor simulator has been designed to demonstrate that various algorithms can indeed be executed on data encrypted with FHE, providing a solution to the termination problem for FHE based processors and the CPA insecurity of SHE processors simultaneously.**

***Index Terms*—Fully Homomorphic encryption, Cloud, URISC.**

I. Introduction

Cloud computing evolves a new paradigm to increase computing and storage capability using external service providers. Using the concept of "loan of software" and hardware, cloud mitigates the need of large resources. However, every solution comes with more new problems. Hence, use of cloud to store any sensitive data may lead to security hindrance. To establish a successful and trustworthy service, it is expected that the cloud service provider will protect the privacy of the information stored in cloud and to achieve this, different techniques have been acquired both in client side and server side. In spite of this, external attackers may penetrate while internal attackers may compromise information.

Concerns regarding privacy and security are the biggest hurdles for the adoption of cloud computing by security-conscious enterprises [1]. Cryptographic techniques can provide a solution to this cloud security problem. Other than providing privacy through anonymity, classical encryption-decryption techniques are beneficial. Users can store encrypted form of potentially sensitive data in such public server to maintain the confidentiality. However, this solution requires extra overhead in case of processing the stored data. Every time for any simple processing on stored data, decryption is necessary. Further, costly encryption operation is required to upload the data back to the cloud. In this way, sensitive data need to be transferred to and from the server and it is repeatedly exposed to adversary. Another major drawback is that huge amount of cloud resources can only be used for storing data only, can never be used for processing of critical information. To avoid these issues, it is required to delegate the ability to process the data without decrypting it. In this scenario, homomorphic encryption scheme is the only answer to this problem [2].

As it was discussed in [1], direct computation on encrypted data can be achieved by adding interaction and using secured hardware. However, Rivest et al. first introduced the concept of $privacy\ homomorphism$ [3]. Homomorphic encryption scheme is a public encryption scheme which allows algebraic manipulations on ciphertexts [1]. That implies any user who is only given two ciphertexts Encrypt(m_1, pk) and Encrypt(m_2, pk) of elements of group $(G_2, *)$, can compute as $Encrypt(m_1, pk) \oplus Encrypt(m_2, pk)$ without the knowledge of secret key and plaintexts. Operations $*$ and $\oplus$ depend on the choice of encryption scheme. Hence, with the use of such encryption scheme, cloud can process encrypted data without knowing the actual data and result.

However, capability of processing directly on single encrypted data does not suffice the requirement of fully secured computation. If the computation flow remains unencrypted in secured processing, that may leak sensitive information. Hence, present researchers are exploring to develop secured encrypted processors where data as well as computations both are encrypted. In [4], a Turing complete encrypted One Instruction Set Computer (OISC) has been proposed based on partially homomorphic Paillier encryption scheme. However, this design suffers from a few limitations from the security point of view. Firstly, to design the encrypted memory the encryption scheme is considered to be deterministic and that makes the design susceptible to Chosen Plaintext Attack (CPA). However, underlying somewhat homomorphic encryption scheme is incapable of designing encrypted memory supporting randomized encryption scheme. This randomization is supported only by Fully homomorphic encryption (FHE) as an underlying scheme.

In literature, researches on FHE are taking place in different directions. Encrypted bitwise additions and multiplications are defined in [5] and implemented using integers in [6] and [7]. Further efficiency enhancement on fully homomorphic encryption has been reported in [8], [9] and [10]. In [11] and [12] advancements have been proposed to implement faster encryption schemes. Further, in [13] and [14] recent developments of FHE have been discussed. In [15]–[17], searching and sorting on FHE data have been investigated. To accelerate the performance of FHE, use of hardware has been also investigated in [18].

In [19], authors have given an initial layout of designing FHE encrypted processors. However, determination of termination point of any encrypted program or identifying the end point of any encrypted loop are major challenges in case of designing FHE based processor. In [20], a proposed solution of this problem is to define a possible maximum loop length and the loop or program terminates once the maximum value is achieved. This solution requires large number of redundant operations in a program as number of loops increases. In our work, we explore how this problem can be better handled with client intervention and message passing protocol between client and server. Further, we combine the flexibility of processing arbitrary operations on encrypted data by FHE scheme with simplicity of unit reduced instruction set architecture (URISC) and investigate the benefit of applying such design to solve the termination problem.

Our contribution in this paper is to develop an encrypted processor able to perform arbitrary computations on encrypted data with encrypted instructions. However, while working in encrypted domain, handling different machine opcode is difficult since, same opcode generates (bitwise) different ciphers due to the randomization property of encryption scheme. This motivates us to design FHE encrypted unit reduced instruction set computer (FURISC) architecture which works with single opcode rather than a multi-instruction processor. The URISC is considered the penultimate reduction of Reduced instruction set computer [1], which is capable of synthesizing a complete set of operations with the help of single instruction. Here, we provide the design of Subtract and Branch if Negative (SBN) and Move operation based FURISC architecture and finally explain how the encrypted CPA secured FURISC architecture is capable of handling the encrypted loop termination problem in a more practical way. With examples of basic sorting and searching techniques we show the timing requirement of actually computing different arbitrary operations on encrypted data.

The rest of our paper is organized as follows: In section II, we discuss the preliminaries of homomorphism, specially the FHE scheme. Next, section III gives the justification of designing FHE based OISC. In section IV, we explain our proposed design of FHE based SBN processor and it is compared with Move based FHE processor in section V. Finally, we compare our design with existing works in section VII and conclude in section VIII highlighting some possible future works.

II. Preliminaries

Before going to the detailed design of the proposed encrypted processor, we first discuss the basic principle operation of the FHE scheme. Fully Homomorphic encryptions provide a mechanism to perform arbitrary computations over encrypted data. The promise shown in the work of Gentry [2] had been followed by several improvements to develop more efficient realizations of this technique, which has potential applications for performing privacy preserving operations, that is 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. Homomorphisms and Fully Homomorphic Encryption Scheme

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 between the two sets with its reverse function t' and an operation $\oplus$. The system is a homomorphism, if $\forall(p_1, p_2) \in P$, $(p_1 \oplus p_2) = t'(t(p_1) \ominus t(p_2))$. If there are two functions $\oplus$ and $\otimes$, such that $\forall(p_1, p_2) \in P$, $(p_1 \oplus p_2) = t'(t(p_1) \ominus t(p_2))$ and $\forall(p_1, p_2) \in P$, $(p_1 \otimes p_2) = t'(t(p_1) * t(p_2))$. This is called an algebraic homomorphism. Operations $\oplus$ and $\otimes$ on plaintext may be similar or may be different with the operations $\ominus$ and $*$ performed on ciphertext. The obvious practical implication is the possibility to transform the two members p_1 and p_2 into the range of C, thus applying some sort of encryption, and having the operations $\oplus$ and $\otimes$ (or equivalent operations) performed by a third party. The result can then be decrypted back into the range of P. An algebraically homomorphic crypto-system can be described as a 6-tuple $H_1 = (P, C, t, t', \oplus, \otimes)$ where P and C denote the plain-text space and the ciphertext space, respectively, whereas t and t' denote the encryption and decryption functions. $\oplus$ and $\otimes$ tag the two algebraic operations. In group homomorphic encryption scheme (GHE), the encryption function forms group homomorphism and the encryption scheme allows an operation on ciphertexts being equivalent to some binary operations on corresponding plaintexts [1].

B. Fully Homomorphic Encryption

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 FHE scheme is explained in [2]. The scheme has the security parameter 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:

1) $KeyGen(\lambda)$: 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 $pk \leftarrow (p, \overrightarrow{y})$.
2) $Encrypt(pk, m)$: 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 $Encrypt$.
3) $Decrypt(sk, c^*)$: 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) $\sum_t S_t z_t = \sum_t c S_t y_t = \frac{c}{p} \bmod 2$.

The above encryption allows arbitrary computations on encrypted data by defining 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 [6], [21]. In our work, while performing homomorphic operations, we have re-used the homomorphic modules proposed in Scarab library [22].

C. Scarab library

Scarab library is an implementation of a FHE scheme using large integers. This scheme is based on the proposed work in [23] with some modifications in recrypt operation. In [23], authors have constructed a modified FHE scheme with relatively small key and ciphertext size from a somewhat homomorphic scheme based on Gentry's work [2]. This modification has smaller message expansion and key size than Gentry's original scheme and also allows efficient fully homomorphic encryption over any field of characteristic two. Hence, this work is more practical in case of applying FHE to real applications and this is the building block of the Scarab library.

The implementation of this library uses the GNU Multiple Precision Arithmetic Library (GMP) for large integers and Fast Library for Number Theory (FLINT) as helping libraries. Detailed encryption-decryption scheme along with the modifications in recrypt operation has been vividly explained in [24]. In the following sections, we explore the design of FURISC processor using the modules present in Scarab library.

III. Implementing Homomorphic Encryption using a Ultimate RISC Instruction

Ultimate RISC (URISC) is the minimalistic perspective to computer architecture design, where a single instruction is used to perform all computations. In this section, we first outline the rationale of using URISC for realizing FHE algorithms.

A. Justification of Single Instruction Processor for Encrypted Data

Fully Homomorphic Encryption (FHE) provides an avenue for performing arbitrary computations on encrypted data. However, capability to operate on encrypted data alone is not sufficient for secured computation. In order to ensure that the control flow of the program is secured it is necessary that the address space is also encrypted. Consider, the program snippet, $\texttt{if(a[i]} > \texttt{a[j])}\ \texttt{i} = \texttt{i} + 1;$ It can be observed that if the data is encrypted, the outcome of the comparison is also encrypted which leads to the fact that, to update the index of the array the index also needs to be encrypted. Thus we develop a processor architecture wherein the data and the memory content is encrypted. Using the standard load-store paradigm of RISC processors thus the program, which is comprised of the instructions from the Instruction Set Architecture (ISA) is also encrypted. In this section, we study the motivation of using a single RISC instruction, URISC, to build such a processor. We also address several related issues and discuss the motivation of choosing a FHE based URISC, which we call as FURISC.

1) Why a single Instruction?: As discussed, the memory content, which stores both the data and the instructions, have to be in an encrypted format. It may be mentioned, that for protections against Chosen-Plaintext Attacks (CPA) and other stronger forms of adversaries, the encryption algorithms are randomized. This implies that the same plaintext, m can be encrypted to different ciphertexts, $c = Enc(m, r)$, where Enc is the encryption algorithm and r is the random input[1]. Thus a computer which has multiple instructions in its Instruction Set Architecture (ISA) will lead to the situation where with varying keys the same instruction would give rise to different encrypted instructions, and hence varying opcodes. This would make the functioning of the computer infeasible. URISC provides a unique opportunity in this context. A URISC is an abstract machine, which uses only a single instruction and other necessary instructions are composed from the single instruction set [25]. Thus, the URISC is *Turing Complete*, and one can perform all computations using a single instruction. This resolves the confusion regarding varying opcode in case of a standard RISC or CISC processor, which has multiple instructions in their ISA.

2) Pitfalls of using Somewhat Homomorphic Schemes: Why Fully Homomorphic Encryption?: For designing encrypted processors, somewhat homomorphic schemes are the first choice since FHE scheme suffers from performance issues. In [4] and [26], authors have explored the design of encrypted one instruction set processor based on the Paillier based encryption, which is an additive homomorphic encryption scheme. The underlying instruction which is a subleq (alias SBN) is a single instruction whose arithmetic computation is a subtraction on two operand values. In the same instruction, depending on whether the result is positive or negative, the Program Counter (PC) gets updated to the next address or an instruction mentioned as another operand of the SBN instruction. Since, Paillier encryption supports subtraction on encrypted data this is a promising choice to develop a processor for performing arbitrary computations on encrypted data (decompose the program using encrypted SBN instructions, and subsequently execute them using Paillier Encryption algorithm).

Unfortunately, the design suffers from a serious deficiency. The PC needs to be updated based on an encrypted condition after the subtraction. Thus while the subtraction is supported by the underlying SHE, the update of the PC needs an

[1]The decryption is however always deterministic algorithm.

encrypted decision making module which can be realized by an encrypted multiplexer. To explain, consider a decision block, where depending on an encrypted condition c', output y' may be a' or b' (all the variables are encrypted). The decision block can be realized by a multiplexer $y' = a'(c') + b'(\overline{c}')$, where the computations of the right hand side are homomorphic. Thus, design of a multiplexer on encrypted data and control requires capability to perform both encrypted addition and multiplication, which is not supported by any somewhat homomorphic scheme.

In order to make these decisions, the design proposed in [4] uses sign lookup memory table for storing sign for encryptions of numbers. Moreover, it is assumed that the encryption is deterministic and the public key of the encryption is unknown to the adversary. In several real life scenarios such a restriction may not be feasible and the deterministic encryption can make the processor computations vulnerable to chosen plaintext attack (CPA) [27].

This motivates us to look into replacing SHE with FHE, since FHE supports both addition and multiplication, which in turn is capable of designing an encrypted decision module, namely the multiplexer. This provides the flexibility of making branch decisions and PC updations even when the encryption is randomized, without the use of any static encryption table and making the encryption deterministic. All these issues motivate to design an encrypted processor with FHE as the underlying encryption scheme.

3) Issue in Fully Homomorphic Processor : Termination problem: Effort to design FHE based processor has been first made in [20]. However, a major open problem is to detect and handle the termination of encrypted processes. Since, any encrypted process is locked in the cipher-space, all the intermediate termination conditions are encrypted. Hence, it is impossible to identify the termination points of loop termination or process termination from unencrypted domain. One possible solution of encrypted loop termination problem as proposed in [20], is by defining the maximum number of cycles, that need to be performed to safely execute the encrypted program. However, this incurs extra overhead of redundant operations. Moreover, when any program consists of numbers of loops, termination of each loop is handled in the same way by mentioning maximum number of possible cycles. Hence, large number of redundant FHE operations, required to handle each loop termination further increase the cost of overhead. When performance of FHE operations is a major hurdle, this solution of termination handling is an added bottleneck to performance.

Here, we propose a different approach of handling termination problem in a better way in any cloud-server setting, more specifically when the server is a public cloud. We consider authorized clients to encrypt FHE data and store them in the cloud server, where homomorphic processing on the encrypted data is supposed to take place generating encrypted results. From the security point of view, if client (or any other adversary) can identify the termination point of any encrypted process without having access to the secret key, then it is a potential threat to the underlying cryptosystem. Hence, the determination of the event of termination should always require the decryption key. However, server does not have access to the secret key or decryption capability and hence it requires client intervention to handle the termination problem. Next, we show how message passing protocol between server and client can be a better option to handle this termination problem.

B. Client Intervention to Handle Termination

Homomorphic operations perform directly on encrypted data and produce the final encrypted results in the cloud server. However, real world works on unencrypted data, hence authorized client may decrypt the encrypted result finally at the client side if the unencrypted value is required for further processing. This capacity of decryption in the client end can be used to solve the problem of encrypted loop termination without leaking any critical information.

In Fig. 1, we explain a generalized encrypted loop execution and termination with client intervention. Here, we define an unencrypted variable $loopHndl$ at the server side such that loop is getting executed if $loopHndl = 1$. FHE_Compare is an encrypted module which compares between encrypted loop counter variable (enc_i) and encrypted value of maximum loop count and generates $compResult$, which is sent to client in each iteration of loop. $compResult$ value is decrypted at the client side and if it is 1 (indicates enc_i reached the maximum value and the loop should be terminated) an interrupt is generated in the client side to set the value of $loop\_End$ to 1. This unencrypted value is sent back to the server as $loopHndl$ and based on this value the control exits from the loop. Since, $loopHndl$ is not directly related to the critical information of the instruction executed in the loop, this unencrypted traffic from client to server does not reveal any sensitive information. Additionally, server and client can settle on some symmetric encryption key and the client can sent encrypted value of $loopHndl$ to the server and server can then decrypt it.

However, this way of handling loop termination requires client intervention and decryption of the $loopHndl$ signal for each loop. In practical scenario, any program should consist of multiple loops and hence large number of message passing from client to cloud server as well as decryption of each signal need to be handled separately. This incurs extra overhead in terms of network bandwidth, synchronization and decryption operation. Subsequently, we shall explain how this multiple message passing for termination can be reduced in an efficient way if the underlying processor is an encrypted FURISC architecture and termination can be handled with *single* message passing between client and server.

C. Solving Termination Problem in Fully Homomorphic Processor using FURISC

As explained, rather than designing an overall FHE processor we prefer a single instruction architecture, since URISC supports Turing complete computation obviating the need of different machine opcodes. Different types of single instructions for modeling of URISC are [25]:

- Subtract and branch if less than or equal to zero.

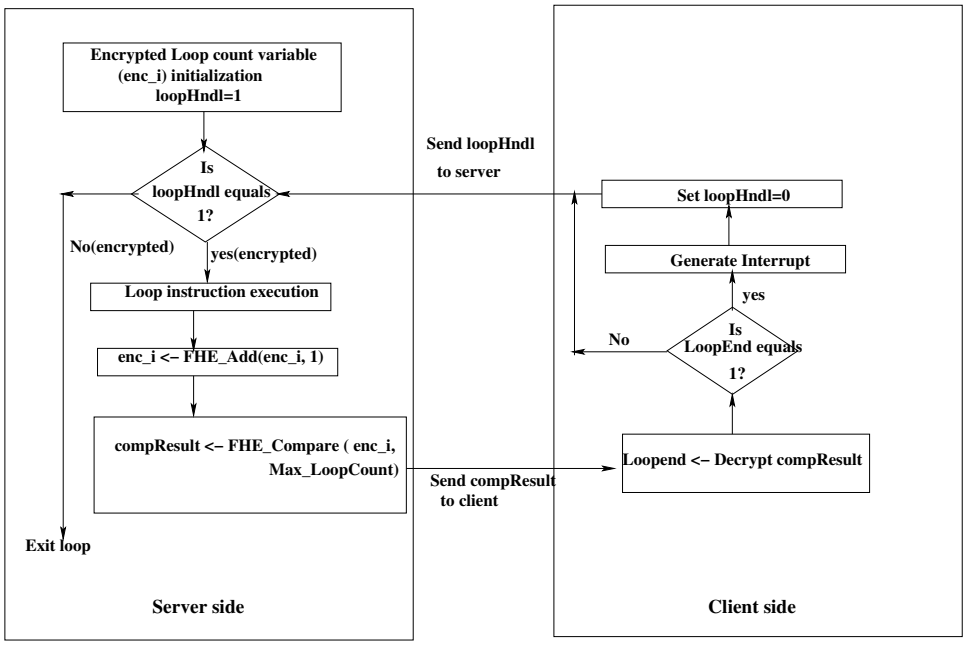

Fig. 1. Example of encrypted loop handling

- Subtract and branch if negative (SBN).
- Reverse subtract and skip if borrow
- Move.

Here, we take the example of SBN instruction based FURISC and show how it can be advantageous to handle termination problem of FHE processes in an efficient way with *single* client-server interaction in comparison to *multiple* client server interactions. In SBN instruction, the operandum is subtracted from operandam and the execution proceeds to next-address if the subtraction result is negative.

More formally, the instruction is represented as:

```
SBN A, B, C :
   Mem[B] = Mem[A] - Mem[B];
       if (Mem[B]< 0)
             goto C
       else goto next instruction
```

In case of encrypted processes, let the loop handling condition (or termination condition) be decided based on the subtraction results of address `A` and address `B` contents (`Mem[A]` and `Mem[B]`) and it is stored in `Mem[B]`. Depending on the value of `Mem[B]`, PC can jump to instruction within the loop itself or proceed to the next instructions out of the loop, once loop end condition has been reached. Multiple loops can be handled in the same way and in such scenario, client-server message passing for each loop is not necessary for multiple loop handling. Finally, once the final termination condition has been reached PC can jump to a dedicated predefined *End of program* location. Client can get the information of program termination by a single message passing from this particular location. In the next section, we first explore how to design URISC processor using FHE as an underlying encryption scheme and gradually explain this solution of encrypted program termination with more examples.

IV. Design of FURISC

In this section, we discuss the design basics of FURISC processor based on two primitive URISC instructions : SBN and MOVE. Here we consider 4-tuple format of SBN instruction and explain how to design an SBN based FURISC.

Let `A'`, `B'` and `C'` be FHE encrypted memory addresses and `Mem'[A']` and `Mem'[B']` be the encrypted contents of the respective addresses. With these parameters, fully homomorphic SBN instruction can be represented as:

```
SBN A', B', resultant', C' :
 resultant' = Mem'[A'] - Mem'[B'];
   if (resultant' < enc (0))
         goto C'
   else goto next instruction
```

Implementation of this FHE based SBN instruction requires the following steps:

- *Encryption Phase:* Memory addresses `A`, `B`, `C` should be encrypted by FHE to `A'`, `B'`, `C'` and contents of the addresses are stored in encrypted format.
- *Memory read-write:* Contents of memory address `A'` and memory address `B'` need to be fetched. *Encrypted memory module* handles memory read and write operation which will be explained in a subsequent section.
- *FHE_Subtraction:* The subtraction of `Mem'[A']` and Mem'[B'] is performed by FHE_Sub module of FURISC processor and stored in register `resultant'`.
- *Branching or program counter (PC) Updation:* If (`resultant' < enc(0)`), the execution control proceeds to `C'` or to next instruction pointed by encrypted PC i.e (`PC' + 1`). This branch updation is handled by *FHE_Branch* module, which is again part of Encrypted ALU of FURISC.
- Value of the register `resultant'` is finally updated to

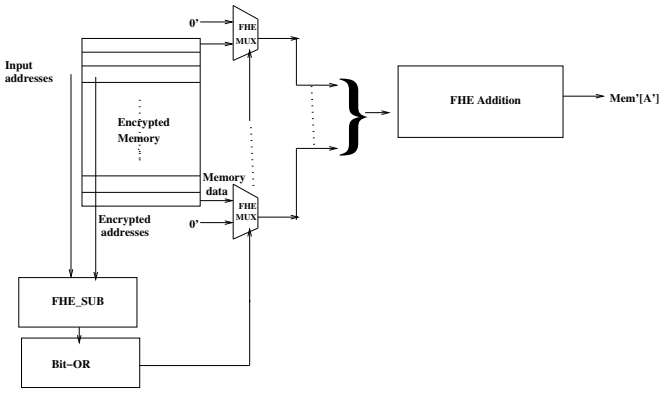

Fig. 2. Encrypted memory read module for FURISC

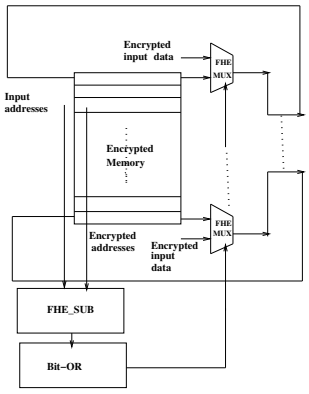

Fig. 3. Encrypted memory write module for FURISC

certain memory or register address location according to URISC instructions.

In the following subsection, we shall explain how to implement the mentioned modules using the Fully Homomorphic primitive circuits mentioned in HElib [22] like FHE_Add (Add ciphertext bits (XOR)), FHE_Mul (Multiply ciphertext bits (AND)), FHE_Fulladd (Add with carry in and carry out) and FHE_Halfadd (Add with carry out).

A. Encrypted memory module

Encrypted memory in FURISC design requires manipulation of encrypted data as well as encrypted addressing. The main design challenge of designing such memory is that the underlying encryption algorithm is randomized. Hence, initially encrypted data may be stored in a certain encrypted address. During memory-fetch encryption of same address gives a different result (bitwise values are different). That makes the content fetching more difficult from a particular address of memory. Hence, an encrypted decision making module is required for encrypted memory read-write as proposed in [20].

Fig. 2 and Fig. 3 describe how encrypted memory works. In our design, the base address of the memory is encrypted and the next locations are determined incrementing (homomorphically) the base address consecutively. Encrypted data are stored in these encrypted addresses. To fetch data from any of these locations, the input encrypted memory address need to be matched with the encrypted locations of the memory. Since every time the encryption algorithm generates different encrypted values for same address, homomorphic address matching technique is required. Hence, to search a particular memory, input encrypted address is subtracted from each location address of encrypted memory by FHE_SUB module (will be explained in section IV-B) and bitwise OR of the subtraction result is computed using FHE_OR module (Bit-OR module in the diagram). Output of the FHE_OR module is fed as the selection lines of FHE multiplexers (FHE_MUX) attached to each location of memory. In case of memory read, if any match is found (Output of FHE_OR module is $enc(0)$), data from the matched location is fetched to a temporary register (otherwise ($enc(0)$) value is added to the register value). In case of memory write, input encrypted data is written in the matched location based on selection of FHE multiplexers (FHE_MUX). Thus, FHE multiplexer (FHE_MUX) modules are used to check encrypted matching. Once match is found memory read-write operation is performed from or to the matched memory.

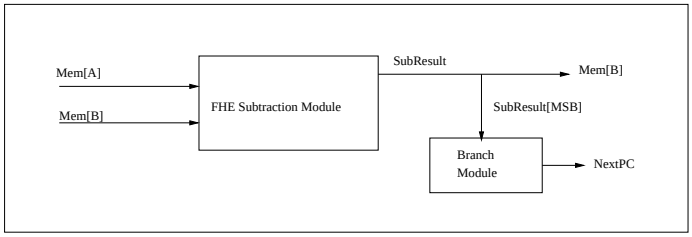

Fig. 4. Encrypted ALU module for FURISC

However, directly following this approach of encrypted memory design incurs large overhead. Main drawback of this design is that for every instruction, memory read or write operation requires search through the whole memory to find the exact matched location. In our design, we separate instruction memory and data memory to reduce the search space. Further, we dedicate a separate unencrypted bit to mark the active memory space. Each time, instructions of a specific program are loaded to the memory, those recently loaded locations are marked as active. Once the program terminates, the attached memory locations are marked as inactive. Thus, for every program counter (PC) updation search is restricted in active instruction memory locations only. Again, for data read or write from or to memory, search is restricted only within active data memory.

B. Encrypted ALU module

The main arithmetic operations of this ALU for FURISC processor is FHE subtraction and PC updation as shown in Fig. 4. The ALU module mainly consists of a fully homomorphic subtraction module (FHE_Sub) and FHE branch module (FHE_branch).

FHE_Sub **module:** FHE Subtraction is implemented by adding one number with the 2's complement of another. The subtraction module is designed by performing homomorphic

addition of one ciphertext with 2's complement of another ciphertext.

FHE_branch **module:** According to the principle of SBN instruction, branching operation decides whether the program control will next proceed to address `C'` or to the next address of program counter (`PC' + enc(1)`). Since all the operations will take place in encrypted domain in FURISC, the next proceeding address should also be encrypted. For this reason, FHE_MUX is used with two inputs, `C'` and the incremented `PC' + enc(1)`. The branching depends on the decision if the subtraction result of `Mem'[A']` and `Mem'[B']` is negative. Hence, the most significant bit (MSB) of the subtraction result is treated as the selection line ($MSB = enc(1)$ indicates the value as negative).

C. Overall architecture

Fig. 5 shows the overall architecture with encrypted memory module and encrypted ALU. SBN functionality is realized with the following steps with this architecture:

- Register *A'*, *B'* and *C'* hold the address values as mentioned in the SBN instruction parameter.
- Initially, address of *A'* is taken into *PC'* and the memory content is fetched from the *Encrypted Memory* by memory read operation. *Memory Read/Write Module* works as mentioned in section IV-A. The fetched value is stored in register *Mem'[A']*.
- Similarly, contents of memory address *B'* is stored in *Mem'[B']*. Selection of *A'* or *B'* is controlled by *sel*, the selection line of associated FHE_MUX.
- Subtraction operation is performed using the *FHE ALU* module and the result is stored in *Resultant* register.
- Further, MSB of *Resultant* register value is fed as selection to a *FHE_mux* for PC updation and the next PC address is determined from the two inputs *(PC + 1)'* and *C'* of the multiplexer depending the selection value. It may be noted that *(PC + 1)'* can be obtained by homomorphically adding the cipher corresponding to 1 with that corresponding to *PC*.
- Depending on the third parameter of the SBN instruction, value stored in the *Resultant* register is updated in the respective memory or register location.

So far we have discussed how to design SBN based FURISC. Another approach of FURISC design is based on *MOVE* instruction, which basically works on copy operation. Intuitively, MOVE based architecture should be better in terms of performance in comparison to subtraction based SBN architecture since copy operation does not require any recrypt operation. Recrypt is the costlier operation during FHE based computations and the main reason for slow performance for any FHE operation. In the next section, we outline a comparison between MOVE and SBN based FURISC and explore which design is actually advantageous in terms of performance.

V. Comparison with MOVE Based URISC

The format of the basic instruction for MOVE based FURISC is:

```
MOVE operandam' operandum'
```

The implication of this instruction is to copy the contents of the operandam' to operandum', where these are two encrypted addresses. The copy can be performed from any location to other (to any memory or register from any memory or register). Hence, the design of a MOVE based architecture only requires memory fetch-write operations and register fetch-write operations. Memory read-write and register operations are performed using encrypted multiplexer as explained in section IV-A.

A. Performance evaluation: SBN vs Move FURISC

In this section, we evaluate which FURISC architecture between SBN and MOVE is worthy to consider in terms of performance. Here, SBN and MOVE based implementations are compared in terms of number of instructions and investigate which one is really faster. Let a program P be implemented by n_1 SBN instructions. Let same program P be implemented by n_2 MOVE instructions. Now, let a single SBN instruction be implemented by m_1 MOVE instructions. Hence, intuitively converting all SBN instructions of program P to MOVE instruction is equivalent to implementing P only with MOVE instructions. That implies, Thus, the code length of the program P using only MOVE instructions is proportional to $n_1 m_1$ MOVE instructions. Similarly, let single MOVE instruction be implemented by m_2 SBN instructions, hence code length of program P is proportional to $n_2.m_2$ instructions. That again implies, $\frac{n_1.m_1}{n_2.m_2} = \frac{n_2}{n_1}$ or $(\frac{n_1}{n_2})^2 = \frac{m_2}{m_1}$. Following code snippets show how single SBN and MOVE instructions can be mapped to their respective MOVE and SBN equivalents.

A single MOVE instruction `MOVE operandam operandum` can be realized by a single SBN instruction:

```
SBN operandam, #00, operandum, #00
```

However, a single SBN instruction: `SBN operandam operandum resultant next-address` can be realized by the following instructions:

```
INVERT operandum
ADD operandam operandum resultant
COMPARE resultant CONSTANT
BRANCH next-address
```

In this instruction sequences, `operandum` is inverted and `(-operandum )` is added to `operandam` and addition result is stored in `resultant`. `resultant` is compared with `CONSTANT` to check if it is negative and branch to `next-address` depending on the value of the `resultant`. All the instructions like `INVERT, ADD, COMPARE, BRANCH` need to be realized by multiple MOVE instructions. Hence, number of MOVE instructions required to implement a single SBN instruction (m_1) is greater than number of SBN instructions equivalent to one MOVE operation (m_2) [25]. That again implies, $m_1 > m_2$ and hence $n_1 < n_2$. Hence, it indicates SBN based URISC architecture requires lesser number of instructions compared to MOVE instruction based URISC to implement any program. In practical scenario, large number of instructions indicate large number of PC

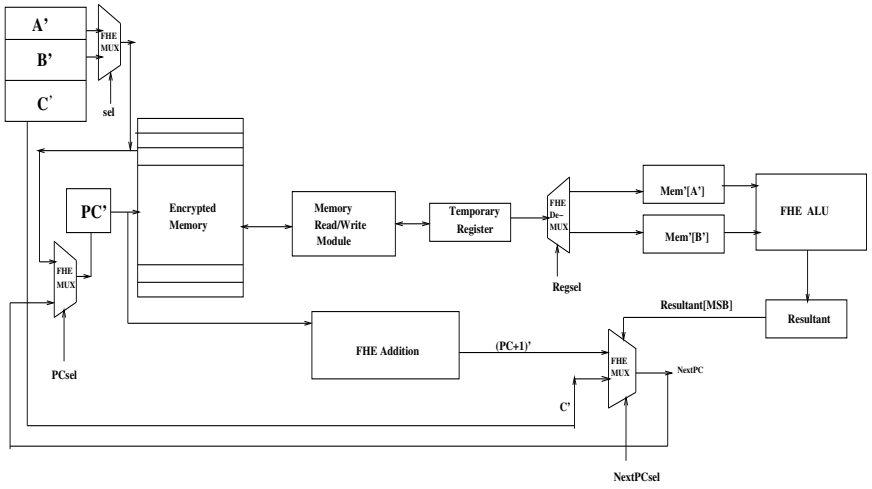

Fig. 5. Overall FURISC Architecture

TABLE I
TIMING REQUIREMENT OF ENCRYPTED OPERATIONS ON FURISC

Operations	CPU cycles
SBN processor implementation	
Fibonacci	$3918*10^8$
Binary Search	$96*10^8$
Quick sort	$12012*10^8$
Move processor implementation	
Fibonacci	$4836*10^8$
Binary Search	$251*10^8$
Quick sort	$156026*10^8$

updation and memory, register handling. In both SBN and MOVE based FURISC architecture, PC updation and memory and register read-write operations require large number of FHE operations, hence that incurs extra timing requirement in terms of CPU cycles. Due to this reason, MOVE based FURISC is not advantageous in terms of performance.

Table I shows the number of required CPU cycles to implement different encrypted functions on encrypted SBN and Move based FURISC. The results are obtained designing C-based simulators of SBN and MOVE based FURISC architectures. The simulators are designed using modules defined in Scarab library and evaluated for correctness on a Linux Ubuntu 64-bit machine with $i686$ architecture 1.6GHZ processor. Among the implemented functions, Fibonacci requires single loop handling, binary search and sort algorithms require multiple loop handling. Here, we show the required CPU cycles for computing Fibonacci value of 100, for performing binary search within 100 data and performing quick sort on a collection of 100 data. The experimental results also conform the theoretical observation that MOVE based processors require higher number of CPU cycles and perform inferior compared to the SBN based counterpart.

VI. FURISC APPLIED TO SOLVE TERMINATION

In this section, we explain with examples how FURISC architecture helps to tackle encrypted loop termination problem. Initially, we start with an example of a simple loop:

```
while(x > y)
  {
      x--;
  }
```

Since, x and y are both encrypted, the termination condition of the loop `while (x > y)` is impossible to comprehend when the code is executed in any general purpose processor. In FURISC architecture, SBN instructions realize the loop in the following way:

```
while' : SBN $1000', $1001', temp', &wend'
         SBN $1000', enc(1), $1000', null
         SBN PC', &while', PC', &while'
wend'    SBN $1000', enc(0), $1000', null
```

Let encrypted x and y be stored in encrypted addresses `$1000'` and `$1001'`, `while'` indicates encrypted starting addresses of while loop execution and `wend'` is the encrypted address where program counter (PC) should jump once the while loop gets terminated. The advantage of designing this FURISC is that PC updation can be controlled by encrypted subtraction operation since PC and all the address locations are encrypted. Thus, when encrypted x is less than encrypted y, subtraction result between contents of `$1000'` and `$1001'` becomes negative. That indicates the encrypted termination condition has been reached and PC now should jump to `wend'`. If this is the only loop present in program then the `wend'` is a no operation (NOP) and PC next jumps to *End of program* location. Otherwise, PC jumps to next instruction of the program. NOP is implemented by subtracting `enc(0)` from the value of `$1000'` location and storing the result back to the `$1000'` location.

In the next example, we shall show how multiple encrypted loop termination is handled using FURISC. We take the example of quick sort which consists of multiple nested loop. Following is the representation of this code realized with SBN instruction architecture.

```
/********* FURISC implementation of
       Quick sort in Appendix  *********/
/***** Starting of if: lines 1-4 ******/
QS':     SBN last', first', temp', &EOP
```

```
        SBN first', enc(0), pivot', null
        SBN first', enc(0), i', null
        SBN last', enc(0), j', null

/*********  while(i<j): line 5 *********/
while1': SBN j', i', temp', &wend1'

         SBN enc(0), pivot', jtemp', null
         SBN $2000', jtemp', temp1', null

/*********  while loop : line 6-7 ******/
while2': SBN enc(0), i', itemp', null
         SBN $2000', itemp', temp', null
         SBN temp1', temp', accumulator',
                                 &while3'
         SBN last', i', product, &while3'
         SBN product, enc(0),temp',&while3'
         SBN enc(0), enc(1), temp'
         SBN i', temp', i'

/********* End of while of line 6  ******/
wend2'   SBN PC', &while2', PC', &while2'

         SBN enc(0), pivot', jtemp', null
         SBN $2000', jtemp', temp1', null

/********  while loop : line 8-9 *******/
while3': SBN enc(0), j', jtemp', null
         SBN $2000', jtemp', temp', null

        SBN temp1', temp', accumulator',
                                 &wend3'
        SBN j', enc(1), j'
        SBN PC', &while3', PC', &while3'

/******* End of while of line 8 *******/
wend3': SBN j', i', temp', &endif'

        SBN enc(0), i', itemp', null
        SBN $2000', itemp', temp', null
        SBN $2000', itemp', (mem_tempi)',
                                  null
        SBN enc(0), j', jtemp', null
        SBN $2000', jtemp', temp1', null
        SBN $2000', jtemp', (mem_tempj)',
                                  null

        SBN  temp', enc(0), (mem_tempj)',
                                   null
        SBN  temp1', enc(0), (mem_tempi)',
                                   null


endif : SBN PC', &while1', PC', &while1'

/****** End of while of line 5 *******/
wend1': SBN enc(0), pivot', itemp', null
        SBN $2000', itemp', temp', null
        SBN $2000', itemp', (mem_tempi)',
                                  null
        SBN enc(0), j', jtemp', null
        SBN $2000', jtemp', temp1', null
        SBN $2000', jtemp', mem_tempj',
                                  null

        SBN  temp', enc(0), mem_tempj',
                                   null
        SBN  temp1', enc(0), mem_tempi',
                                   null

        SBN j', enc(1), j'
        SBN QS', enc(0), PC', PC'


        SBN enc(0), enc(1), temp'
        SBN j', temp', j'
        SBN QS', enc(0), PC', PC'

/********* End of Program *********/
EOP: SBN $2000', enc(0), $2000', null
```

This code snippet shows how easily the nested loop can be handled using this architecture. Here, we consider array `x[ ]` is resided at starting address `$2000'`. At `while1'`, the condition (`i'< j'`) (`i'` and `j'` are the encryption of `i` and `j`) has been checked using (`SBN i', j', temp', wend1'`), where `i'` and `j'` are stored in intermediate registers. With the SBN functionality, `j'` is subtracted from `i'` and the loop condition is checked. When `j'` is less than `i'`, subtraction result is negative and PC' proceeds to end of while (`wend'`). For `while2'`, loop condition is checked by subtracting `i'` from `last'` and if the subtraction result is negative the program flow is branched to `while3'`. Thus, multiple loop is handled without the requirement of any redundant operation.

Once, the termination condition is reached PC should jump to the *End of program* location. Hence, the termination problem reduces to determine whether the PC has reached to *End of program* location. In our design, we have dedicated a particular address location as End of program address (EOP). Since, all the address locations as well as the PC are encrypted, client can not directly know when PC has been reached to EOP. Ideally, client should not know this information without the access to secret key since it will hamper the security of the crypto system. To solve this issue, we consider an encrypted termination-bit, which is set high once PC has reached the EOP address. This termination-bit is send to the client through an encrypted message. Client is capable of decrypting the bit having access to secret key. Once the termination point is reached decryption of the termination-bit generates an interrupt in the client side, so that client can get the information that the program has been terminated. This method is advantageous over the method of using maximum number of cycles, since no redundant operation is required in this process. Further, it is also better in comparison to the proposed method in section III-B, which requires client intervention and message passing for every loop iteration in a single program. Since, each program consist of numerous loops, large number of message passing, network bandwidth, synchronization and decryptions are necessary for loop handling. On the other hand, our proposed method shows only a *single* client-server message passing is capable of handling termination problem while using FURISC architecture no matter what is the size of the program or how many loops are present.

TABLE II
COMPARISON OF FURISC PERFORMANCE WITH HEROIC

Operations	HEROIC Timing (Clock cycles)	FURISC Timing (Clock cycles)
Factorial	$8.45*10^7$	$402.5*10^8$
Fibonacci	$2.74*10^8$	$396*10^8$
Bubble Sort	$1.54*10^8$	$3509*10^8$

TABLE III
COMPARISON WITH MAXIMUM LOOP COUNT ON UNENCRYPTED PROCESSOR

Fibonacci data	Timing with maximum loop count (clock cycle)	FURISC Timing (Clock cycle)
30	$2400*10^8$	$1188*10^8$
60	$2400*10^8$	$2358*10^8$
90	$2400*10^8$	$3528*10^8$
100	$2400*10^8$	$3918*10^8$

VII. COMPARISON WITH EXISTING WORKS

Table II shows a comparison of FURISC performance with SHE based HEROIC proposed in [4], [26]. According to the experimental results, FHE based FURISC requires more clock cycles compared to HEROIC for implementing same operation, but it is advantageous in terms of security improvement and providing CPA resistance to the encrypted processor. Unlike [4], access to public key need not be restricted and the encryption can be randomized for CPA resistance.

We compare the performance of FURISC with the proposed work in [20], where termination condition of FHE process is handled by predefined value of maximum loop count. The disadvantage of this proposed method is that every algorithm is bound to give the worst case performance. For example, in case of binary search algorithm on n FHE data, maximum loop execution count should be prefixed at $O(logn)$ (since from the knowledge of unencrypted binary search $O(logn)$ is the worst case performance timing requirement). In this case, program cannot terminate before and best case performance of $O(1)$ can never be achieved. Hence, this incurs large amount of redundant operations. Similarly, sorting algorithms need to always iterate for $O(n^2)$ times. The main advantage of our proposed technique is that the encrypted program terminates as and when the processing is complete, hence it is possible to achieve the best or average performance of respective algorithms.

Further, table III shows a performance comparison of FURISC with encrypted algorithms executed on unencrypted processors. We choose Fibonacci computation as an example with a fixed maximum loop count. All the implementations are evaluated for correctness on a Linux Ubuntu 64-bit machine with $i686$ architecture 1.6GHZ processor. Result shows whatever be the actual data for Fibonacci computation, always it takes computation time for maximum loop count. Hence, it requires large number of redundant operations for smaller data and number of redundant operation decreases as the data is closer to maximum loop count.

VIII. CONCLUSION

In this work, we present an encrypted URISC architecture with FHE as underlying encryption scheme, which combines the flexibility of performing arbitrary operations on encrypted data due to the property of FHE with design simplicity of URISC architecture. Due to the use of FHE, randomization in memory handling and PC branching solves the CPA Vulnerability issues of previous SHE based design [4]. Further, we also show how this design is advantageous to handle encrypted loop termination problem. As a future work, performance improvement of FURISC can be investigated to make this design more practical for implementing in context of cloud computing.

REFERENCES

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

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

[3] 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.

[4] N. G. Tsoutsos and M. Maniatakos, "Heroic: Homomorphically encrypted one instruction computer," in *Proceedings of the Conference on Design, Automation & Test in Europe*, ser. DATE '14. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2014, pp. 246:1–246:6. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616606.2616907

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

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

[7] 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.

[8] 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.

[9] 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.

[10] A. Silverberg, "Fully homomorphic encryption for mathematicians," *IACR Cryptology ePrint Archive*, vol. 2013, p. 250, 2013. [Online]. Available: http://eprint.iacr.org/2013/250

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

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

[13] V. Vaikuntanathan, "Computing blindfolded: New developments in fully homomorphic encryption," in *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*, 2011, pp. 5–16. [Online]. Available: http://dx.doi.org/10.1109/FOCS.2011.98

[14] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) LWE," *SIAM J. Comput.*, vol. 43, no. 2, pp. 831–871, 2014. [Online]. Available: http://dx.doi.org/10.1137/120868669

[15] 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.

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

[17] 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.

[18] Y. Doroz, E. Ozturk, and B. Sunar, "Accelerating fully homomorphic encryption in hardware," *IEEE Transactions on Computers*, vol. 99, no. PrePrints, p. 1, 2014.
[19] M. Brenner, H. Perl, and M. Smith, "Practical applications of homomorphic encryption," in *SECRYPT 2012 - Proceedings of the International Conference on Security and Cryptography, Rome, Italy, 24-27 July, 2012, SECRYPT is part of ICETE - The International Joint Conference on e-Business and Telecommunications*, 2012, pp. 5–14.
[20] ——, "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.
[21] D. Stehle and R. Steinfeld, "Faster fully homomorphic encryption," Cryptology ePrint Archive, Report 2010/299, 2010, http://eprint.iacr.org/.
[22] https://hcrypt.com//scarab library.
[23] N. P. Smart and F. Vercauteren, "Fully homomorphic encryption with relatively small key and ciphertext sizes," in *Proceedings of the 13th International Conference on Practice and Theory in Public Key Cryptography*, ser. PKC'10, Berlin, Heidelberg, 2010, pp. 420–443.
[24] H. Perl, M. Brenner, and M. Smith, "Poster: an implementation of the fully homomorphic smart-vercauteren crypto-system." in *ACM Conference on Computer and Communications Security*, Y. Chen, G. Danezis, and V. Shmatikov, Eds. ACM, 2011, pp. 837–840.
[25] W. F. Gilreath and P. A. Laplante, *Computer Architecture: A Minimalist Perspective*. Springer Publishing Company, Incorporated, 2012.
[26] N. G. Tsoutsos and M. Maniatakos, "Investigating the application of one instruction set computing for encrypted data computation," in *SPACE*, 2013, pp. 21–37.
[27] J. Katz and Y. Lindell, *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007.

APPENDIX

CODE FOR QUICK SORT ALGORITHM WITH MULTIPLE LOOP

```
1. if(first<last){
2.           pivot=first;
3.          i=first;
4.          j=last;

5.          while(i<j){
6.          while(x[i]<=x[pivot]&&i<last)
7.                  i++;
8.              while(x[j]>x[pivot])
9.                  j--;
10.              if(i<j){
11.                  temp=x[i];
12.                  x[i]=x[j];
13.                  x[j]=temp;
14.              }
15.          }

16.          temp=x[pivot];
17.          x[pivot]=x[j];
18.          x[j]=temp;
19.          quicksort(x,first,j-1);
20.          quicksort(x,j+1,last);
21.     }
22. }
```