CSEE4840 Project Design Document

Homomorphic Encryption Accelerator

Lanxiang Hu, Liqin Zhang, Enze Chen
Department of {Electrical Engineering, Computer Science}
Columbia University
New York, NY
{lh3116, lz2809, ec3576}@columbia.edu

CONTENTS

I Introduction .......................................................... 2
II System Block Diagram ............................................. 2
III Theory .................................................................................. 3
   III-A Motivation ................................................................. 3
   III-B Formulation ............................................................... 3
   III-C Encryption and Key Switching .................................... 3
   III-D Decryption ................................................................. 4
   III-E Encrypted Domain Operations .................................... 4
      III-E1 Addition ............................................................... 4
      III-E2 Linear Transform .................................................. 4
      III-E3 Weighted Inner Product ....................................... 5
      III-E4 Polynomial ............................................................ 5
      III-E5 Examples ............................................................. 5
IV Design .................................................................................... 6
   IV-A Software ................................................................. 6
      IV-A1 Client ................................................................. 6
      IV-A2 Server ............................................................... 8
      IV-A3 Client-Server Communication ................................ 9
   IV-B Hardware ................................................................. 10
      IV-B1 Client-side Accelerator ............................................ 10
      IV-B2 Server-side Accelerator ........................................ 13
   IV-C Hardware/Software Interface ..................................... 16
      IV-C1 Client-side Kernel .................................................. 16
      IV-C2 Server-side Kernel ................................................ 16
V Resource Budgets ............................................................. 16

References ............................................................................. 17
I. INTRODUCTION

In the past few decades, it has witnessed evolution in cryptographic techniques and growing numbers of applications. In Zhou and Wornell’s work [1], the fully homomorphic encryption scheme they proposed encrypts integer vectors to deliberately generate malleable ciphertext and to allow computation of arbitrary polynomials in the encrypted domain. In this scheme, specific integer vector operations including addition, linear transformation and weighted inner products are supported. Building upon that and by taking combinations of these primitive operations, arbitrary polynomial can be effectively computed with high accuracy.

This fully homomorphic encryption scheme is useful for applications in cloud computation, when one be interested in learning low dimensional representations of the stored encrypted data without exposing either data plaintext or operation plaintext to the server.

Moreover, in the dawn of the DL explosion for smartphones and embedded devices, it’s possible for the devices deployed with DL models to be interested in accessing cloud data and make inferences on them. However, many DL-based models deployed on embedded devices are not well-protected; a research in 2019 showed that out of 218 DL-based Android apps, only less than 20% of them use encryption [2]. Homomorphic encryption, on the other hand, can resolve this problem easily by the DL models to malleable ciphertexts.

II. SYSTEM BLOCK DIAGRAM

In this work, we present the Integer Vector Homomorphic Encryption scheme on embedded devices as a prototype for encrypting vector-valued functions. We will use softwares run by the processor as client and server in the encryption scheme, and hardwares implemented on FPGA as the accelerators. The system block diagram can then be drawn as shown in Fig. 1.

Figure 1. Overview of the Software/Hardware Architecture Design
III. THEORY

A. Motivation

Consider the following scenario. Assume there is a client whose data is stored in a cloud server and the data is encrypted. The client decides to make a hidden query on the data without letting the cloud server learn the nature of the data or anything about the query.

To achieve this goal, fully homomorphic encryption is adopted to make ciphertext malleable. According to the homomorphic encryption scheme proposed by Zhou and Wornell, both plaintext and ciphertext considered in this case are integer-valued vectors with an integer-valued matrix as the secret key. Mathematically, we can formulate the problem as follows.

B. Formulation

Let the plaintext be \( x \in \mathbb{Z}^m \) and the corresponding ciphertext be \( x \in \mathbb{Z}^n \) with a large scalar \( w \), a secret key \( S \in \mathbb{Z}^{m \times n} \), and an error term \( e \in \mathbb{Z}^m \) such that

\[
S \cdot \mathbb{E} = w \cdot x + e
\]  

To keep the error term small while applying multiple linear operations in the encrypted domain, we want to assume \( \| S \| \ll w \).

Moreover, consider an arbitrary linear operator \( A \in \mathbb{R}^{n \times m} \), to keep the result as integer vector and the scheme self-consistent, the following operations along with their notations will be used throughout this presentation and will be implemented in the accelerators.

- \( \langle a \rangle \) for scalar \( a \in \mathbb{R} \), define \( \langle a \rangle \) to round \( a \) to the nearest integer.
- \( \langle a \rangle \) for vector \( a \in \mathbb{R}^n \), define \( \langle a \rangle \) to round each entry \( a_i \) in \( a \) to the nearest integer.
- \( \langle a \rangle \) for vector \( a \in \mathbb{R}^n \), define \( \langle a \rangle := \max_i \langle a_i \rangle \).
- \( \langle A \rangle \) for matrix \( A \in \mathbb{R}^{n \times m} \), define \( \langle A \rangle := \max_{ij} \langle A_{ij} \rangle \).
- \( \langle A \rangle \) for matrix \( A \in \mathbb{R}^{n \times m} \), define \( \langle A \rangle := \max_{ij} \langle A_{ij} \rangle \).

C. Encryption and Key Switching

To encrypt \( x \), let \( wI \) be the original secret key. Consider a key-switching operation that can change a secret-key-ciphertext pair to another with a new secret key while keeping the original plaintext encrypted. Without loss of generality, let the plaintext currently be encrypted as ciphertext \( c \in \mathbb{Z}^n \), we want to devise a new secret-key-ciphertext pair \( (S', c') \) such that

\[
S' \cdot c' = S \cdot e
\]

The first step is to convert \( S \) and \( e \) into intermediate bit representation \( S^* \) and \( \cdot c \) with \( c = S \cdot e \). The bit representation follows \( \langle c \rangle := \max \{ |c_i| \} = 1 \) to prevent the transformed error term \( e' \) from growing too large to preserve correctness while rounding to the nearest integer.

First of all, pick a scalar \( \ell \) that satisfies \( 2^\ell > |c| \). Assume \( c_i = b_{i0} + b_{i1} 2 + \cdots + b_{i(\ell-1)} 2^{\ell-1} \). We can then rewrite \( c \) in its bit representation following the rule: \( b_i = \left[ b_{i(\ell-1)}, \ldots, b_{i1}, b_{i0} \right] \) with \( b_{ik} \in \{ -1, 0, 1 \} \) and \( k \in \{ \ell - 1, \ldots, 0 \} \). This gives Eq. 3.

\[
c^* = \left[ b_1^T, \ldots, b_n^T \right]^T
\]

Similarly, we can make a bit-representation of the secret key \( S \) to acquire a new key \( S^* \) with Eq. 4.

\[
S^*_{ij} = \left[ 2^{\ell-1} S_{ij}, \ldots, 2 S_{ij}, S_{ij} \right]
\]

And it can be demonstrated [3, 4] that this technique preserves \( S^* \cdot e = S \cdot e \).

Beyond that, the second step is to convert the bit vector representation into a new secret-key-ciphertext pair. Consider a random Gaussian noise matrix \( E \in \mathbb{Z}^{n \times n}, E_{ij} \sim_{i.i.d.} N(0, \sigma^2_E) \) for some \( \sigma_E \), key-switching matrix \( M \in \mathbb{Z}^{n^2 \times n^2} \) along with the new key \( S' \) such that

\[
S'M = S^* + E
\]

In this architecture, whenever the client decides to perform a linear operation on the encrypted integer vectors stored in the server, the key-switching accelerator will compute the key-switching matrix that corresponds to the specific operation and the client will send the matrix along with desired operation to the server. On the other hand, once received these information from the client, the server carries out vectorized calculations in the custom accelerator over the encrypted domain, and returns encrypted computational results to the client.
Consider keys only with the form $S' = [I, T]$, as a identity matrix concatenated horizontally with some matrix $T$, whose choice is not critical for our purposes. Now we can calculate

$$M = \begin{bmatrix} S^* - TA + E \\ A \end{bmatrix}$$

(6)

where $A$ is another random Gaussian matrix $K \in \mathbb{Z}^{(m'-m) \times n^\ell}$, $K_{ij} \sim \text{i.i.d. } \mathcal{N} \left( 0, \sigma_K^2 \right)$ for some $\sigma_K$. Define

$$e' = Mc^*$$

(7)

And it allows us to calculate

$$S'c' = S^*c^* + e'$$

(8)

where $e' = Ec^*$ is the new error term.

**D. Decryption**

With the encryption scheme specified above, given that we know the secret key $S$, large scalar $w$, notice that nearest integer rounding allows us to recover the plaintext by taking

$$x = \left\lfloor \frac{Sc}{w} \right\rfloor$$

(9)

according to the linear relation in Eq. 1 and the fact that the error term is taken from the set $\{ e \in \mathbb{Z}^m | e_i < \frac{w}{2} \}$ with constrained error.

**E. Encrypted Domain Operations**

To dive into the mathematical algorithm in the operations, first we need to define several variables (or registers in hardware design):

**Definition E1** $c_1, c_2$ are two ciphertexts in the big data stored in the server.

**Definition E2** $S$ is the secret key for encryption. To be mentioned, all the ciphertexts are encrypted with the same secret key, and the key only depends on the operation we choose.

**Definition E3** $M$ is the key-switch matrix that contains the information of the operation as well as the switched secret key.

**Definition E4** $x_1, x_2$ are the corresponding plaintexts of ciphertexts $c_1, c_2$. Usually the cipher-plain pairs are predone and the client knows the address of each ciphertexts, so he/she just need to tell which 2 addresses are used.

After that, we can illustrate the algorithms for carrying out each operation in the encrypted domain as follows.

1) **Addition:** It is obvious that

$$S(c_1 + c_2) = w(x_1 + x_2) + (e_1 + e_2)$$

(10)

so the addition of the ciphertexts in the encrypted domain simply follows

$$c' = c_1 + c_2$$

(11)

Notice that $e_1, e_2$ are devised such that the error is contained within $|e_1| + |e_2| \leq w$.

2) **Linear Transform:** Given a linear transformation $G \in \mathbb{Z}^{m^\ell \times m}$, the encrypted result is

$$(GS)c = wGx + Ge$$

(12)

When we consider $GS$ as a secret key, then the equation above is an encryption of plaintext $Gx$. Thus, the client need to create the key-switch matrix $M \in \mathbb{Z}^{(m^\ell+1) \times m^\ell}$ to switch the key from $GS$ to $S' \in \mathbb{Z}^{m \times (m^\ell+1)}$. After getting $M$ and $S'$, the client can send $M$ to the server, and the cloud server simply computes

$$c' = Mc$$

(13)

as the encrypted result for the client to decrypt.
3) **Weighted Inner Product**: Consider some plaintext $x_1, x_2$, their corresponding ciphertexts $c_1, c_2$ and a matrix $H$ with information about weights of the inner products we want to take. The inner product of our interest takes the form

$$h = x_1^T H x_2$$

(14)

Now, we need one mathematical side note to proceed. Consider the following Lemma.

**Lemma E1** For any arbitrary vectors $x, y$ and matrix $M$ with appropriate dimensions, its inner product follows

$$x^T H y = \text{vec}(M)^T \text{vec}(xy^T)$$

(15)

See [1] and [3] for proof of this Lemma.

In order to compute Eq. 14 in the encrypted domain, we can leverage Lemma E1 to derive the proposition specified below.

**Proposition E1** Consider secret key $S = \text{vec}(S_1^T H S_2)^T$ and ciphertext $c = \begin{bmatrix} \text{vec}(c_1 c_2^T) \\ w \end{bmatrix}$ corresponding to the plaintext of the inner product $x_1^T H x_2$. And for some error term $e$ independent of $e_1$ and $e_2$ for $c_1$ and $c_2$ they satisfy the following condition,

$$\begin{bmatrix} \text{vec}(S_1^T H S_2) \\ \text{vec}(c_1 c_2^T) \\ w \end{bmatrix} = w x_1 H x_2 + e$$

(16)


Notice that instead of a key switching matrix here, the operator to be applied to $\begin{bmatrix} \text{vec}(c_1 c_2^T) \\ w \end{bmatrix}$ in calculating the weighted inner product is a row vector $\text{vec}(S_1 H S_2)^T$. Because of the vectorization operation, the width of operator $\text{vec}(S_1 H S_2)^T$ after key switching is $n^2$.

To this end, we can leverage this property of the row vector, and concatenate $m'$ such operators, each of which corresponds to a weight matrix $H_j, j \in \{1, \ldots, m'\}$, together to be a key-switching matrix $S'$ so that $m'$ such weighted inner products can be carried out simultaneously.

Namely, we now have

$$S' \begin{bmatrix} \text{vec}(c_1 c_2^T) \\ w \end{bmatrix} = wp + e$$

(17)

where vector $p$ contains $m'$ entries of weighted inner products, each of which corresponds to an inner product $x_1^T H_j x_2$.

We can then apply the key-switching algorithm as specified in Eq. 3 to Eq. 8 to $S'$ and obtain a corresponding key switching matrix $M \in \mathbb{Z}^{n^2 \times (m'+1)}$ along with a new secret key $S'' \in \mathbb{Z}^{m' \times (m'+1)}$. The final ciphertext is therefore

$$c'' = M \begin{bmatrix} \text{vec}(c_1 c_2^T) \\ w \end{bmatrix}$$

(18)

4) **Polynomial**: Building upon the three aforementioned operations, we can now synthesize polynomial operations with weighted inner products introduced above to calculate multiple arbitrary degree polynomials in parallel. All we need is to expand the $x$ and $S'$ and thereby account for constant terms in polynomials. Let the modified input vector $x_p = [1, x_1, x_2, \ldots, x_n]^T$.

The new ciphertext then becomes $c' := [w, c_1, \ldots, c_n]^T$. We also need to extend the secret key $S$ because we simply added a constant factor $1$ to $x$:

$$S' := \begin{bmatrix} 1 & 0 \\ 0 & S \end{bmatrix}$$

(19)

Thus, given any inner product weight matrices $\{H_j\}$, we can calculate its key-switch pair $M$ so that each $x_p^T H_j x_p$ can be calculated in accordance with Eq. 16, or more compactly with Eq. 17 to deal with all $\{H_j\}$ in parallel. The steps follow the ones introduced in the Weighted Inner Product section. For degree 2 polynomials, one inner product simply does the computation. For higher-degree polynomials, notice that higher-order polynomials can be calculated based on lower-order polynomials.

5) **Examples**

- **Key Switching Example** Consider the case $\ell = 3$, ciphertext $c = [1, -2]$, and

$$S = \begin{bmatrix} 1 \\ 3 \\ 2 \\ 4 \end{bmatrix}$$

(20)

The corresponding bit representation of $c$ and $S$ are therefore

$$c^* = [1, 0, 0, 0, -1, 0]$$

(21)
\[ S^* = \begin{bmatrix} 4 & 2 & 1 & 8 & 4 & 2 \\ 12 & 6 & 3 & 16 & 8 & 4 \end{bmatrix} \] (22)

- **Polynomial/Weighted Inner Product Example** To calculate the polynomial \( f(x) = x_2^2 - 4x_1x_3 \), we can express it as an weighted inner product \( x^T H x' \), and

\[
H = \begin{bmatrix} 0 & 0 & -2 \\ 0 & 1 & 0 \\ -2 & 0 & 0 \end{bmatrix} \] (23)

IV. DESIGN

A. Software

Software serves as both the client and the server in this project, each of which takes different responsibilities.

1) **Client:** The client-side software serves as the function caller to general homomorphic encryption, and key switching operation that generates public keys for one of the four operations discussed above. Note that key switching is not needed for vector addition. Once the client-side functions are called, homomorphic encryption and key switching for each of the primary operation will be done in the key switching matrix accelerator by making a syscall and invoking the corresponding device driver. The following client-side operations are sketched in pseudocode to illustrate their functionalities. Notice that a library of client-side syscalls is also needed for client user program to talk to the kernel.

```c
static long key_switching_ioctl(struct file *f, unsigned int cmd, arg*) {
    switch (cmd) {
        case WRITE_BIT_REPR_MATRIX:
            // copy matrix S from user, send it to key switching accelerator
            break;
        case READ_BIT_REPR_MATRIX:
            // read matrix S^* from device driver, send it to user program
            break;
        case GETRANDOM_MATRIX:
            // get a random matrix with the get_random_matrix module in hardware
            break;
        case WRITE_MATRIX_MULT:
            // write to the matrix multiplication module
            break;
        case READ_MATRIX_MULT:
            // read from the matrix multiplication module
            break;
        case WRITE_MATRIX_VECTORIZE:
            // write to the matrix vectorization module
            break;
        case READ_MATRIX_VECTORIZE:
            // read from the matrix vectorization module
            break;
        case WRITE_NINT_VECTOR_DIVIDE:
            // write to the nint_vector_divide module
            break;
        case READ_NINT_VECTOR_DIVIDE:
            // read from the nint_vector_divide module
            break;
    }
}

// a struct that contains two matrices
struct client_mat_pair {
    mat* A;
    mat* B;
} mat_pair;

// secret key generator with seed T, such that S' = [I, T]
mat get_secret_key (mat &T) {
    // generator an identity matrix
    // concatenate it with T
    return [I, T];
```
// perform key switching operation
// it returns new secret key S' and key-switching matrix M
mat_pair key_switch (mat &T, mat &S)
{
    // call key_switching_accelerator in hardware
    ioctl(fd, WRITE_BIT_REPR_MATRIX, &S); // compute key switch matrix in hardware
    mat S_star;
    ioctl(fd, READ_BIT_REPR_MATRIX, &S_star);
    mat A, E;
    ioctl(fd, GET_RANDOM_MATRIX, &A);
    ioctl(fd, GET_RANDOM_MATRIX, &E);
    mat TA;
    ioctl(fd, WRITE_MATRIX_MULT, &T, &A);
    ioctl(fd, READ_MATRIX_MULT, &TA);
    M = [S_star - TA + E]
    // concatenate M vertically with A
    S_prime = get_secret_key(&T);
    mat_pair result;
    result.A = &S_prime;
    result.B = &M;
    return result;
}

mat decrypt (mat &S, mat &c, int w)
{
    mat Sc;
    mat R;
    ioctl(fd, WRITE_MATRIX_MULT, &S, &c);
    ioctl(fd, READ_MATRIX_MULT, &Sc);
    ioctl(fd, WRITE_NINT_VECTOR_DIVIDE, &Sc, w);
    ioctl(fd, READ_NINT_VECTOR_DIVIDE, &R);
    return R;
}

// to perform addition operation, key switching is not needed. See Eq. 11.

// call server side linear transform operation, send the key switch matrix M from GS
// -> S' to the server. See Eq. 12.
mat_pair client_linear_trans(mat &G, mat &S, mat &T)
{
    mat GS;
    ioctl(fd, WRITE_MATRIX_MULT, &G, &S);
    ioctl(fd, READ_MATRIX_MULT, &GS);
    return key_switch(&T, &GS);
}

// call server side inner product operation
// returns M, the key switch matrix from vec(S^t H S) to S, multiple rows of vec(S^t H
// -> S) can be made into S'
// notice that this code only takes vec(S^t H S) rather than S' as the operator
mat_pair client_inner_product(mat &H, mat &S, mat &T)
The function specified above only works for degree-2 polynomial as an example demonstrate the workflow. To implement arbitrary degree-d polynomial, more sophisticated functions are needed to factorize polynomials into a sequence of weighted inner product operations.

2) **Server**: The server-side software serves as the data center where user information are stored. It takes pre-configured and encrypted data and performs encrypted domain operations in the server-side hardware to accelerator computations. All computations can be decomposed into a combination of vector additions, linear transformations and weighted inner products. See the following functions for pseudocode.

```c
// Here we assume the device drivers has been properly initialized and allocated with memory, // The following syscalls write and read from the key-switching device drivers. // each operation specified below corresponds to a call that interacts with I/O memory // corresponds to the device drivers.
static addition_ioctl(struct file *f, unsigned int cmd, arg*) {
    switch (cmd) {
    case VEC_ADDITION:
        // perform vector addition for input vectors c1 and c2
        // return the result in one cycle as a vector
    }
}

static linear_transform_ioctl(struct file *f, unsigned int cmd, arg*) {
    switch (cmd) {
    case WRITE_LINEAR_TRANSFORM:
        // write matrix M and vector c to linear transform accelerator
    case READ_LINEAR_TRANSFORM:
        // read result of the linear transform from device driver, send it to user program
    }
}

static inner_prod_ioctl(struct file *f, unsigned int cmd, arg*) {
```
switch (cmd) {
    case WRITE_WEIGHTED_INNER_PROD:
        // write key-switching matrix W along with vector c1, c2, and scalar w to
        // weighted inner product accelerator
        case READ_WEIGHTED_INNER_PROD:
        // read result of weighted inner product device driver, send it to user
        // program
    }
}

// server side addition operation, return the vector addition of two ciphertexts
vec serv_addition(vec &c1, vec &c2)
{
    vec c;
    ioctl(fd, VEC_ADDITION, &c_1, &c_2, &c);
    return c;
}

// server side linear transform operation, returns c’==S(Gx) given c=Sx and M
mat serv_linear_trans(mat &M, vec &c)
{
    vec c_prime;
    ioctl(fd, WRITE_LINEAR_TRANSFORM, &M, &c);
    ioctl(fd, READ_LINEAR_TRANSFORM, &c_prime);
    return c_prime;
}

// given two ciphertexts and the keyswitch matrix, computes a single weighted inner
// product
int serv_inner_product(vec c1, vec c2, mat M, int w)
{
    int a;
    ioctl(fd, WRITE_WEIGHTED_INNER_PROD, &c1, &c2, &M, w);
    ioctl(fd, READ_WEIGHTED_INNER_PROD, a);
    return a;
}

// given two ciphertexts and the keyswitch matrix, computes result of a degree 2
// polynomial
int serv_deg2_polynomial(vec c1, vec c2, mat M, int w)
{
    return serv_inner_product(c1, c2, M, w);
}

The function specified above only works for degree-2 polynomial. To generalize the function that computes arbitrary degree-
d polynomial, more sophisticated functions are needed to factorize polynomials into a sequence of weighted inner product
operations.

3) Client-Server Communication: Query functions that can be called by the client to send information to the server in
order to carry out specific computations include:

Query functions that can be called by the server to send information to the server in order to carry out specific computations
include:

// Assume a socket or pipe is used for communication between client and server.
// Assume the socket/pipe has been initialized.
int send_query(int sockfd, size_t msg, size_t msgSize)
{
    // send operation type
    // send address for ciphertext 1
10

send address for ciphertext 2 (if any)

// send key-switching matrix (if any)

return success;

size_t listen(int sockfd, size_t *msgSize)

{  
  // receive signal
  // read message (32-bit integer ciphertext)
  // return NULL if fails
  return ciphertext;

}

All the function calls via syscall can be interpreted as an operation done in the hardware.

11

// Assume a socket or pipe is used for communication between client and server.

// Assume the socket/pipe has been initialized.

size_t handle_query(int sockfd, size_t *msgSize)

{  
  // receive signal
  // parse operation type
  // parse address for ciphertext 1
  // parse address for ciphertext 2 (if any)
  // parse send key-switching matrix (if any)
  // return NULL if fails

  // acquire c1 and c2 according to the addresses
  // return pointer to a struct query
  return &query;
}

int write_back(int sockfd, size_t msg, size_t msgSize)

{  
  // send ciphertext through socket
  return success;
}

B. Hardware

1) Client-side Accelerator: The client-side accelerator is mostly responsible for key-switching operations to get the correct key-switching matrices for calculating linear transformation, weighted inner product and polynomial. After the key switching completes, a new Secret key $S'$ will be derived and retained by the client to decrypt encrypted result once the server has done computation, and a key-switching matrix $M$ will also be generated and sent to the server to carry out different computations.

- Key-switching accelerator: The key-switching accelerator is the only device driver in the client-side hardware. It composes of two major sub-modules, one module for getting bit-representation of a vector corresponding to Eq. 3, one module for getting bit-representation of a matrix corresponding to Eq. 4 and one module for getting random Gaussian matrix in order to calculate Eq. 6. We can sketch the pseudocode for each module as follows. Note that for vectors and matrices exceeding the size capacity of the accelerators, the kernel is responsible for breaking the query into reasonable sizes that fits the accelerators (maximum row and column number as 256).
input logic write,
input logic chipselect,
input logic [7:0] width,
input logic [31:0] c_i,

output logic [7:0] output_length,
output logic [31:0] c_star_i);

logic read_enable;
logic start_comp;
logic [7:0] write_index;
logic [7:0] read_index;

// initialize a DMEM to store input vector.
dmem dmem_0(...);
always_ff @(posedge clk) begin
  if (reset) begin
    // handle reset conditions
  end else if (chipselect && write) begin
    // preload each element into DMEM
    // keep track loaded index number
    // once all elements loaded, set computational flag to true
  end
end

// do computations when computational flag is true
always_ff @(posedge clk) begin
  if (start_comp) begin
    // perform bit-representation conversion for vector c
    // keep track output length
    // once finished, set read_enable to true
    // set computational flag to false
  end
end

// read each element every clock cycle
always_ff @(posedge clk) begin
  if (read_enable) begin
    // spitting out one element each cycle with appropriate index
    // once finished, set read_enable to false
  end
end
endmodule
genvar j;
genvar gen
// initialize DMEM blocks, each to store an output row vector
genenerate
always_ff @(posedge clk) begin
    if (reset) begin
        // handle reset conditions
    end else if (chipselect && write) begin
        // preload element in each row into DMEM
        // keep track index and loaded row numbers
        // once all loaded, set computational flag to true
    end
endgenerate

always_ff @(posedge clk) begin
    if (reset) begin
        // handle reset conditions
    end else if (chipselect && write) begin
        // preload element in each row into DMEM
        // keep track index and loaded row numbers
        // once all loaded, set computational flag to true
    end
end

// do computations when computational flag is true
always_ff @(posedge clk) begin
    if (start_comp) begin
        // perform bit-representation conversion for secret key S
        // keep track output length and width
        // once finished, set read_enable to true
        // set computational flag to false
    end
end

// read element in each row every clock cycle
always_ff @(posedge clk) begin
    if (read_enable) begin
        // spitting out element in each row each cycle
        // once finished, set read_enable to false
    end
end
endmodule

module get_random_matrix(input logic clk,
input logic reset,
...,
input logic gen,
input logic chipselect,
input logic [7:0] length,
input logic [7:0] width,
output logic [31:0] S_star_ij);
logic [31:0] seed;
logic [7:0] generated_row_num;
// generate multiple LFSR instances to create a Gaussian random variable at each cycle
lfsr lfsr_0(...);

genvar j;
genvar gen
// initialize DMEM blocks, each to store an output row vector
genenerate
always_ff @(posedge clk) begin
    if (reset) begin
        // handle reset conditions
    end else if (chipselect && gen) begin
        // generate a Gaussian random variable and store in DMEM
        // keep track index and row numbers
        // once finished, set read_enable to true
    end
end
endgenerate

always_ff @(posedge clk) begin
    if (reset) begin
        // handle reset conditions
    end else if (chipselect && gen) begin
        // generate a Gaussian random variable and store in DMEM
        // keep track index and row numbers
        // once finished, set read_enable to true
    end
end

// read each element every clock cycle
always_ff @(posedge clk) begin
Notice that more parallelism can be acquired with this design if we wrap each of the module presented above with a higher-level module that instantiates multiple instances simultaneous, one for every row vector. This way, we can achieve linear speedup proportional to the number of instances initiated.

Several helper modules corresponding to some operations defined in Definition B1 ~ Definition B5 are needed including:

```plaintext
module dmem(...);
    // Initialize DMEM of appropriate sizes
endmodule

module lfsr(...);
    // Linear-feedback shift register as random number generator
endmodule

module mat_mult(...);
    // Perform matrix multiplication
endmodule

module nint_vector_divide(...);
    // Divide each entry in a vector by w
    // Round division to nearest integer according to Definition B3
endmodule
```

2) **Server-side Accelerator**: Since the operations over the encrypted domain have four different types, each of which involves different number and size of inputs, we want to use four different accelerators, each of which serves as a distinct device driver in the kernel. Also, in order to be robust, each device driver needs to have sufficiently large memory in order to handle with large key-switching matrices in various tests. This memory block will be implemented as DMEM and will be discussed in more details in section VI.

At a high level, each device driver consists of two major steps: **MEMORY READ** that uses the address given by kernel to fetch the ciphertexts from memory for calculation, and **OPERATION** that performs specific vector operations and generates computational results. Note that each device driver corresponding to each of the four operations takes different number of ciphertexts and perform different computations (as mentioned in the Encrypted Domain Operation section).

In other words, four device driver instances are needed including addition accelerator, linear transformation accelerator, weighted inner product accelerator and polynomial accelerator respectively.

- **Addition Accelerator**:
  The vector addition accelerator takes the simple form of performing a element-wise addition of two vectors. In the pseudocode presented below, the vector addition accelerator adds two 16-element vectors together at one cycle. Queries about ciphertexts stored at \(c_{1\_addr}\) and \(c_{2\_addr}\) will be read from the database, and each element will be sent to one port of the following module.

```plaintext
module 16elem_addition(input logic clk,
    input logic reset,
    ...,
    input logic write,
    input logic chipselect,
    input logic [31:0] c_1_1,
    ...,
    input logic [31:0] c_1_16,
    ...,
    input logic [31:0] c_2_1,
    ...,
    input logic [31:0] c_2_16,
    ...,
    output logic [31:0] c_1,
    ...,
    output logic [31:0] c_16);
    // Instantiate 16 32-bit integer adders
    adder adder_1{...};
    ...
```
adder adder_16(...);

// Read adders' outputs
endmodule

- **Linear Transformation Accelerator:**
  The linear transformation accelerator simply takes a matrix multiplication of the key-switching matrix received from the client and apply it as a linear operator to ciphertext $c$ stored at address $c_{addr}$. Queries about ciphertext stored at $c_{addr}$ will be read from the database and each 16-element row of the key-switching matrix $M$ will be sent to the accelerator from the server each cycle, with each of these elements takes one input port of the following module.

```verilog
module 16elem_linear_transform(input logic clk,
                             input logic reset,
                             ..., input logic write,
                             input logic chipselect,
                             input logic [7:0] width,
                             input logic [7:0] length,
                             input logic [31:0] W_i1,
                             ..., input logic [31:0] W_i16,
                             input logic [31:0] c_1,
                             ..., input logic [31:0] c_16,
                             output logic [31:0] y_i));

logic [7:0] current_row_num;
always_ff @(posedge clk) begin
    if (reset) begin
        // handle reset conditions
    end else if (chipselect && write) begin
        // take inner product of S_i with c
        // keep track of current row numbers
        // read the result as an integer at the next cycle
    end
endmodule
```

- **Weighted Inner Product Accelerator:**
  Recall Eq. 18, for this example pseudocode implementation of the weighted inner product module, consider ciphertext as a 4-element vector $c$. 4-element vector $c$ is chosen because matrix vectorization after outer production yields $4^2 = 16$ element output. Queries about ciphertexts $c_1, c_2$ stored at $c_{1,addr}$ and $c_{2,addr}$ will be read from the database and sent to the accelerator along with the scalar $w$ in Eq. 18 in the first cycle. Each 16-element row of the key-switching matrix $M$ at one cycle.

```verilog
module 16elem_weighted_inner_prod(input logic clk,
                                   input logic reset,
                                   ..., input logic write,
                                   input logic chipselect,
                                   input logic [31:0] c_1_1,
                                   ..., input logic [31:0] c_4_1,
                                   ..., input logic [31:0] c_1_2,
                                   ..., input logic [31:0] c_4_2,
                                   input logic [7:0] width,
                                   input logic [7:0] length,
                                   input logic [31:0] W_i1,
                                   ..., input logic [31:0] W_i16,
```

• Polynomial Accelerator:

Note that calculating polynomial is essentially a glorified weighted inner product according to our scheme. Therefore, having let the kernel figure out how to perform a sequence of weighted inner products to obtain a polynomial of degree \( d \), we can potentially customize an accelerator for a polynomial of, say degree 2, that supports inner product for two ciphertexts \( c_1, c_2 \) as 16-element vectors. The only difference is that \( W \) is now derived from a secret key of the form

\[
\text{Eq. 19.}
\]

Therefore, as long as we modify the weighted inner product module presented above to a slightly larger size that fits the key-switching matrix corresponding to \( S' \rightarrow S'' \) in Eq. 18, the accelerator can do its job. In our design, we action of scheduling and arranging accumulative weighted inner product is done in the kernel. Only
individual weighted inner products are performed in hardware. Several helper modules corresponding to some operations defined in Definition B1 ∼ Definition B5 are needed including:

```
module vectorize(...);
    // Merge two vectors in to one
    // Can be used to vectorize a matrix according to Definition B5
endmodule

module nint_vector_divide(...);
    // Divide each entry in a vector by w
    // Round division to nearest integer according to Definition B3
endmodule

module outer_product(...);
    // Perform outer product according to Eq.18;
endmodule

module adder(...);
    // 32-bit integer scalar adder;
endmodule

module dmem(...);
    // Initialize DMEM of appropriate sizes
endmodule
```

C. Hardware/Software Interface

1) Client-side Kernel: For the client-side kernel, the only device driver can be accessed is the key-switching accelerator. Recall from Eq. 4 to Eq. 6, for each key-switching operation, in order to compute $S^*$, $M$ and $S'$, the module needs to take matrix $S$ as input, along with two random Gaussian matrices that can be generated by the hardware itself (see the Client-side accelerator section).

In this design, we plan to support key-switching matrix of up to 16 elements of width and 256 elements of lengths (height, number of rows). Namely, from Eq. 5, it follows $n' \leq 256$, $n \ell \leq 16$.

At the lowest level based on our hardware design, the key-switching operation is done by taking element-wise inputs and store them into multiple DMEM blocks, each of which as a row vector. However, notice that this operation can be further parallelized if we modify the bit_repr_matrix module such that processes each row vector at a time. Alternatively, it can also be done by instantiating multiple instances of bit_repr_vector to convert each row of $S$ to its bit representation at one cycle.

With this architecture, there will be 16 32-bit-wide input ports for the key-switching accelerator to deliver one row of the secret key $S$ at a time and achieve a significant speedup.

2) Server-side Kernel: For the client-side kernel, the device drivers that can be accessed include addition accelerator, linear transformation accelerator, weighted inner product accelerator and polynomial accelerator.

In this design, we plan to support vector operations with up to 16 elements at a time in one cycle. Each element in a vector is taken as a 32-bit signed integer in agreement with the scheme. Specifically, that means the addition accelerator can calculate the sum of two 16-element vectors $c_1, c_2$ in one cycle. The linear transformation accelerator can compute the inner product of one 16-element row vector $W_i$ with one 16-element column vector $c$ and completes the desired linear transformation in $m$ cycles given $W \in \mathbb{Z}^{m \times 16}$. The weighted inner product accelerator and the polynomial accelerator take in 16-element row vector of $W$ each cycle along with $c_1, c_2, w$ at the first cycle, and they generate the output all at once after some cycles of operations to compute intermediate steps.

As discussed in the Server-side accelerator section, one thing to notice is that polynomial accelerator is essentially the same as the weighted inner product accelerator with a slightly different key-switching matrix and ciphertext. Therefore, it’s important for the kernel and the polynomial accelerator device driver to handle one weighted inner product operation at a time. The order at which weighted inner products will be carried out and the exact steps are handled by the server-side software as discussed in Server-side software section.

V. Resource Budgets

For the client-side accelerator, notice that we plan to support key-switching matrix of up to 16 elements of width and 256 elements of lengths. Since each element is a 32-bit integer, each row in the key-switching matrix takes a DRAM of size $16 \times 32$, so the maximum size for a DRAM that stores each row is $M_i = 512$ bits. For a key-switching matrix with $m \leq 256$ rows, it takes up to 256 such DRAM blocks to store all rows. This costs $256 \times 512 = 131072$ bits = 16384B = 16kB. Let each of this memory module be call Dmem in the system block diagram as shown in Fig. 1. We can create 8 such DRAM modules as client-side accelerator’s cache, and it takes 16kB $\times 8 = 128$kB of memory on FPGA.
For the server-side accelerator, we as well plan to support key-switching matrix of up to 16 elements of width and 256 elements of lengths. This is the same as the client-side accelerator because the key-switching matrix serves as the public key and need to be used by the server to carry out linear transformation and polynomial computations. We want to create 16 such DRAM modules as server-side accelerator’s cache for not only key-switching matrix but also results from outer product and vector addition, so it takes $16 \times 16 = 256$ kB of memory on FPGA.

Some registers might also be needed to store real-time vector inputs, results including nearest integer vector division, matrix vectorization and linear transformation results while doing linear transformation. Assume they take up to 256kB of memory on FPGA.

Combine all of them together, we need around 640kB of memory on FPGA. From The Cyclone® V FPGA core architecture’s specifications, it states that the FPGA comprises of up to 12 Mb of embedded memory arranged as 10 Kb (M10K) blocks. Namely, we have more than 1 MB of memory available on FPGA.

Therefore, we can assume that there is sufficient memory on FPGA for all computations required.

REFERENCES