Vector Homomorphic Encryption Accelerator

CSEE4840 Final Project
Lanxiang Hu, Liqin Zhang, Enze Chen
Outline

• Introduction to Homomorphic Encryption Scheme.
• System Block Diagram.
• Theory.
  o Cryptographical operations.
  o Encrypted domain operations.
• Hardware Implementation and Simulation.
• Software Implementation.
• Hardware-Software Interface.
• Challenge and Conclusion.
Introduction to Homomorphic Encryption Scheme

Fig1. Most Homomorphic Encryption schemes: The cloud has access to the function $f$, and the client sends encrypted $x$ to the cloud for computation.

Fig2. The scheme used in our project. The cloud computes $f(x)$ without knowing either $x$ or $f(\cdot)$. 
System Block Diagram

User-space functions & API

Kernel-space functions
Device drivers

Hardware
(Computational units)
Vector Homomorphic Encryption Accelerator

Original System Block Diagram
Theory — Overview

• Cryptographical Operations:
  o Encryption: $c = E(x)$, choose $c$ such that $Sc = wx + e$, $S$ is secret key.
  o Decryption: $x = D(c)$, decrypt $c$ with $S$, $x = \text{int}(Sc/w)$.
  o Key Switching:
    ➢ Switching Secret key from $S$ to a new key $S' = [I, T]$ such that $Sc = S'c'$.
    ➢ Return Key Switching Matrix $M$. Key Switching Matrix $M$ encodes computational details.
    ➢ Send $M$ to server.
    ➢ Server simply uses $M$ to carry out computation by performing linear transformation to $c$.

• Encrypted Domain (Integer Vector) Operations:
  o Addition of two vectors.
  o Linear Transformation.
  o Weighted inner product.
Theory — Cryptography

- Security: Arithmetic Logic Units perform computations in encrypted domain, and the results can be only be decrypted by the client with the secret key.
- Application Scenarios: Without direct access to ciphertext in the cloud, the client can get computational results while the cloud server is agnostic about computational details.

**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
Theory — Encrypted Domain Operations

**ADDITION**

\[ S(c_1 + c_2) = w(x_1 + x_2) + (e_1 + e_2) \]

**SOLUTION**

Client

Keep \( S \) the same. Send \( c_1, c_2 \)

Server

\[ c' = c_1 + c_2 \]
Theory — Encrypted Domain Operations

LINEAR TRANSFORMATION

\[(GS) \mathbf{c} = wG\mathbf{x} + Ge\]

SOLUTION

Client

Create \( M \) for \( s' = GS \), send \( \mathbf{c}, M \)

Server

\[c' = M \mathbf{c}\]
Theory — Encrypted Domain Operations

WEIGHTED INNER PRODUCT

\[ h = x_1^T H x_2 \]

\[ x_1^T H x_2 = \text{vec} \left( (M)^T \right) \text{vec} \left( x_1 x_2^T \right) \]

\[ \text{vec} \left( (S^T H S)^T \right) \left[ \frac{\text{vec} \left( c_1 c_2^T \right)}{w} \right] = w x_1 H x_2 + e \]

SOLUTION

Client

Create \( M \) for \( S' = \text{vec} \left( (S^T H S)^T \right) \). Send \( M, \omega, c_1, c_2 \)

Server

\[ c'' = M \left[ \frac{\text{vec} \left( c_1 c_2^T \right)}{w} \right] \]
Theory — Encrypted Domain Operations

POLYNOMIAL

$$\mathbf{x}_p = [1, x_1, x_2, \ldots, x_n]^T$$

$$\mathbf{c'} := [w, c_1, \ldots, c_n]^T$$

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

$$h = \mathbf{x}_p^T H \mathbf{x}_p$$

SOLUTION

Client

Create $M$ for $S'' = \text{vec} \left( (S')^T H S' \right)^T$. Send $M, \omega, c'$

Server

$$\mathbf{c''} = M \begin{bmatrix} \text{vec} \left( \mathbf{c'}^T \mathbf{c'} \right) \\ \omega \end{bmatrix}$$
Hardware Implementation and Simulation

• Key-Switching Modules:
  o Take bit-representation of a vector (at most 8-element wide, 32-bit each).
  o Take bit-representation of a matrix (at most 8 by 8 in size, 32-bit each).
  o Get a random matrix with integer entries (at most 8 by 8 in size, 16-bit each).
  o Get a noise matrix with small integer entries (at most 8 by 8 in size, 4-bit each).

• Encrypted-Domain Computational Modules:
  o Vector addition (at most 16-element wide, 32-bit each).
  o Linear Transformation (supports linear operator of at most 16 by 16 in size, 32-bit each).
  o Weighted Inner Product (supports linear operator of at most 16 by 16 in size, 32-bit each).
Implementation and Simulation — Key-Switching Modules

BIT REPRESENTATION OF VECTOR: convert a vector into its bit representation.

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: $b_i = \begin{bmatrix} b_{i(\ell-1)}, \ldots, b_{i1}, b_{i0} \end{bmatrix}^T$ with $b_{ik} \in \{-1,0,1\}, k \in \{\ell-1, \ldots, 0\}$. And this gives Eq. 3.

$$c^* = \begin{bmatrix} b_1^T, \ldots, b_n^T \end{bmatrix}^T$$

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

$$S_{ij}^* = \begin{bmatrix} 2^{\ell-1}S_{ij}, \ldots, 2S_{ij}, S_{ij} \end{bmatrix}$$

Moore machine
Implementation and Simulation — Key-Switching Modules

Vector Homomorphic Encryption Accelerator
Implementation and Simulation — Key-Switching Modules

BIT REPRESENTATION OF MATRIX: convert a matrix into its bit representation.

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

\[
S^*_i = \begin{bmatrix}
2^{i-1}S_{ij}, \ldots, 2S_{ij}, S_{ij}
\end{bmatrix}
\] (4)

```c
7 // secret key
8 int S[2][8] = { {0x6, 0x5, 0x0, 0x3, 0x9, 0x3, 0x3, 0x6}, // 0-7
9 {0x9, 0x7, 0x6, 0x8, 0x2, 0x0, 0x6, 0x1} }; // 8-15

11 // l = 4
12 int S_star[2][32] = { {0x30, 0x18, 0xc, 0x6}, // 0-3
13 {0x28, 0x14, 0xa, 0x5}, // 4-7
14 {0x0, 0x0, 0x8, 0x0}, // 7-11
15 {0x18, 0xc, 0x6, 0x3}, // 12-15
16 {0x48, 0x24, 0x12, 0x9}, // 16-19
17 {0x18, 0xc, 0x6, 0x3}, // 20-23
18 {0x18, 0xc, 0x6, 0x3}, // 24-27
19 {0x30, 0x18, 0xc, 0x6}, // 28-31
20 {0x48, 0x24, 0x12, 0x9}, // 0-3
21 {0x38, 0x1c, 0xe, 0x7}, // 4-7
22 {0x30, 0x18, 0xc, 0x6}, // 7-11
23 {0x40, 0x20, 0x10, 0x8}, // 12-15
24 {0x10, 0x8, 0x4, 0x2}, // 16-19
25 {0x0, 0x0, 0x0, 0x0}, // 20-23
26 {0x30, 0x18, 0xc, 0x6}, // 24-27
27 {0x8, 0x4, 0x2, 0x1}}; // 28-31
```
GET RANDOM MATRIX: get an integer-valued random matrix.

```c
// seeds below can be modified
logic [15:0] seed_0 = 16'd1;
logic [15:0] seed_1 = 16'd2;
logic [15:0] seed_2 = 16'd3;
logic [15:0] seed_3 = 16'd4;
logic [15:0] seed_4 = 16'd5;
logic [15:0] seed_5 = 16'd6;
logic [15:0] seed_6 = 16'd7;
logic [15:0] seed_7 = 16'd8;
// seed above can be modified
logic signed [15:0] lfsr_out_0;
logic signed [15:0] lfsr_out_1;
logic signed [15:0] lfsr_out_2;
logic signed [15:0] lfsr_out_3;
logic signed [15:0] lfsr_out_4;
logic signed [15:0] lfsr_out_5;
logic signed [15:0] lfsr_out_6;
logic signed [15:0] lfsr_out_7;

// generate multiple LFSR instances to create a Gaussian random variable at each cycle
// 8 16-bit LFSR
lfsr lfsr_0(.clk(clk), .resetn(reset), .seed(seed_0), .lfsr_out(lfsr_out_0));
lfsr lfsr_1(.clk(clk), .resetn(reset), .seed(seed_1), .lfsr_out(lfsr_out_1));
lfsr lfsr_2(.clk(clk), .resetn(reset), .seed(seed_2), .lfsr_out(lfsr_out_2));
lfsr lfsr_3(.clk(clk), .resetn(reset), .seed(seed_3), .lfsr_out(lfsr_out_3));
lfsr lfsr_4(.clk(clk), .resetn(reset), .seed(seed_4), .lfsr_out(lfsr_out_4));
lfsr lfsr_5(.clk(clk), .resetn(reset), .seed(seed_5), .lfsr_out(lfsr_out_5));
lfsr lfsr_6(.clk(clk), .resetn(reset), .seed(seed_6), .lfsr_out(lfsr_out_6));
lfsr lfsr_7(.clk(clk), .resetn(reset), .seed(seed_7), .lfsr_out(lfsr_out_7));
```

Moore machine with LFSR pseudorandom number generator.
Implementation and Simulation — Encrypted Domain Operations

**ADDITION** : Each time add n elements of input.
Implementation and Simulation — Encrypted Domain Operations

LINEAR TRANSFORMATION: Each time do inner product of n elements of c and M, sum the result after a whole line has been calculated. Do several epochs until whole M is scanned.

https://drive.google.com/file/d/1cqC6TUnnxU2AczAaUOS2lxPEVCOCCsqM/view?usp=sharing
Implementation and Simulation — Encrypted Domain Operations

‘Batch Size’ = n : Deal with n elements of input vectors at a time.

WEIGHTED INNER PRODUCT :

STAGE 1: Load input c1 and input c2. Do outer product of c1 and c2.
STAGE 2: Vectorize the output in STAGE 1. Divide the vector by w.
STAGE 3: Do linear transformation of the output, using same theory as in LINEAR TRANSFORMATION.
Implementation and Simulation — Encrypted Domain Operations
Vector Homomorphic Encryption Accelerator
Software Implementation

- Matrix operation library.
- Client-side operation library.
- Server-side operation library.
- Syscall library.
- Kernel code for device driver.
Software Implementation

- Matrix operation library.
- Client-side operation library.
- Server-side operation library.
- Syscall library.
- Kernel code for device drivers.
Hardware-Software Interface

- AXI master-slave pair:
  - 32-bit write and read data.
  - Addresses control types of operation each write/read corresponds.
- As a result, control signals need to be sent off by the user as well (more complicated software coding).

```plaintext
const logic [3:0] load_op_type = 4'h0;
const logic [3:0] load_data_type = 4'h1;
const logic [3:0] load_width = 4'h2;
const logic [3:0] load_length = 4'h3;
const logic [3:0] load_input = 4'h4;
const logic [3:0] load_addition = 4'h0;
const logic [3:0] linear_transform = 4'h1;
const logic [3:0] weighted_inner_product = 4'h2;
```
Hardware-Software Interface

User Homomorphic Encryption library

- client_functions.c
- server_functions.c

Matrix library

- math.h

Device driver
- kernel code
- key_switch.c
- encrypt_domain.c

Memory mapped registers

- register for keyswitching device driver
- register for encrypted domain device driver

Avalon AXI master

- key_switch.sv
- encrypted_domain.sv

Avalon AXI slave

- bit_repr_vector.sv
- bit_repr_matrix.sv
- get_rand_matrix.sv
- get_noise_matrix.sv

Avalon AXI master

- vector_addition.sv
- linear_transform.sv
- weighted_inner_prod.sv

Hardware top - level

- ifsr.sv
- its4.sv
Vector Homomorphic Encryption Accelerator
Vector Homomorphic Encryption Accelerator
Challenges

• Dimension scalability.

• Intermediate data caching.

• Interplay among different top-level external and submodule internal control signals.

• User-friendliness.
  ○ Need to manage control signal manually.
Lesson Learned

• Simplify pipeline logic for Avalon bus communication.

• Might be easier to use shared SDRAM rather than implementing memory blocks from scratch so that intermediate results can be more easily cached with SDRAM on hardware.

• Figure out what to put on registers is important and might make life much easier and avoid evoking too many syscalls.