CS 203.4860	Secure Multi-Party Computation	Spring 2023

Homework 6: Report

Lecturer: Dr. Adi Akavia *Student(s): Nir Segal, Hallel Weinberg, Michael Rodel*

Abstract

In this homework we implemented the actively secure BeDOZa protocol with MACs and Oblivious Transfer for computing functions.
We wrote code in Python that uses this protocol to compute any general function and specifically the following function:

$$f_{\vec{a},4}(x) = \begin{cases} 1 & a_1x_1 + a_2x_2 \geq 4 \\ 0 & otherwise \end{cases}$$

Our notebook colab notebook is here:
`https://colab.research.google.com/drive/1R3nxZolFg_X5wgRGJUBjrSrJ1RTTY8qS?usp=sharing`.

1 Introduction

Motivation. When 2 parties want to exchange information, they have to use a secure protocol to ensure that the information remains confidential and protected from any malicious actors who may try to intercept it. The enhanced BeDOZa protocol achieves security against malicious adversary (security with abort), so it provides a way for two parties to exchange information securely. In addition, the Oblivious Transfer makes it possible to perform the safe computation even without the assistance of an honest party (the dealer).

Secure Computation Technique. The BeDOZa protocol [BDOZ10] achieves security against malicious adversary (security with abort) by using an arithmetic circuit, which is a circuit that describes the output of a function for all possible inputs. The two parties compute the output of the function by using Secret Sharing so they do not reveal their inputs to each other. This ensures that even if an attacker is listening in, it can not determine the inputs of the parties and therefore cannot compute the output.
The BeDOZa protocol achieves security against malicious unbounded adversary by using a ElGamal OT (oblivious transfer) and circuit evaluation. The complexity of the protocol is $O(circuit - size)$.[Wik]
The protocol uses pre-processing phase using ElGamal OT (passive technique), which enables Alice and Bob to transfer essential data between to support the protocol they going to apply.
Alice's input is written on an n-size wire $(x_1, ..., x_n)$, and Bob's input also written on an n-size wire $(x_n + 1, ..., x_{2n})$. The output wire is a L-size write. We represent the calculation in a tree (such that the leaves are the inputs and the root is the output. The nodes which

are not the root or the leaves represent (ADD mod p) or (MULT mod p) gates (both types are with constant or two wires). This tree contains d layers, such that the inputs to the gates are at layer $i \in 1, .., d$ are from layers $< i$.
In addition, it uses authenticated secret sharing. This procedure is gained with m-hom MAC security. A m-hom MAC scheme is $(m, \varepsilon) - secure$ if every adversary A wins the security game with probability at most ε. The goal of the MACs is to keep the integrity of the messages whose cross between Alice and Bob. The MACs are represented: (k, t, x). When k is the key, t is the tag and x is the message. When one party send to other party a message, the party actually sends the MAC of the message and the other party verifies the MAC. When the MAC is verified if the following thing happens: $Ver(k_i, t', x') = accept$, and if MAC is not verified we send abort.

Application. In our application we defined 2 classes: Alice and Bob, with both responsible for the offline phase and the online phase.. We first randomly generate $\vec{a}, \vec{x}$ to be used as input to the protocol and then we communicate between the classes using a few lines of code:

1. We perform the offline phase: Alice and Bob $u_A, v_A, w_A, u_B, v_B, w_B$ using OT protocol.
2. Alice and Bob generate r_A, r_B.
3. Alice and Bob generate keys and tags for $u_A, v_A, w_A, u_B, v_B, w_B, r_A, r_B$ using OT protocol.
4. We perform the online phase: we initialize Alice and Bob with their inputs $\vec{a}, \vec{x}$.
5. Alice and Bob share their inputs.
6. For all layer in the circuit:
7. Alice computes $z_A, k_A, t_A, MULTS$ according to the gates in the layer and sends $MULTS$ to Bob.
8. Bob computes z_B (he uses $MULTS$ only if the gate is $MULT([x], [y]))$ and sends z_B to Alice if z_B contains the Bob's output.
9. Alice verifies the answer, and updates her z_A for all the $MULT([x], [y])$ gates in this layer by using what Bob computed.

Finally, Alice reconstructs $z = (z_A + z_B) mod p$ and outputs z.
Note that actually Given $\vec{a}$ and $\vec{x}$, z is equal to the result of the function $f_{\vec{a},4}(x_1, x_2)$ for $\vec{a}, \vec{x}$ above.

Empirical Evaluation. We conducted experiments as follows: for each possible input $\vec{a} = (a_1, a_2), \vec{x} = (x_1, x_2)$ we computed using enhanced BeDOZa protocol an output z.
In short, the results we received correspond to the expected output of the function $f_{\vec{a},4}(x_1, x_2)$ for $\vec{a}, \vec{x}$ the aforementioned.

2 Preliminaries

In this paper we used the arithmetic circuit we built in homework 1 (in figures 3 and 4).

2.1 Against Malicious Attacker

The definition of malicious attacker is someone who can disobey the protocol. Our goal is to recognize if some party does not follow the protocol (and in case some party did this, the other party will send 'abort' or some default value, making the attack valueless).

2.2 Secret Sharing

To be able execute the protocol we need every party to have a secret. And if we reconstruct all these secrets we will get a valuable information. In our case if we will reconstruct all the secrets we will get all the initial inputs. In our case we have a dealer who spreads all these secrets to all the parties in a safe way, it means we can't infer from the traffic what is the combinations of the secrets.

2.3 The MACs

The Security Game for m-hom MAC goes like this: adversary A can query challenger C on messages $x_1, \ldots, x_m$ of his choice (adaptively) and receive corresponding tags $t_1, \ldots, t_m$, where $t_i \leftarrow Tag(k_i, x_i)$. for $k_i = (\alpha, \beta_i)$ for $\alpha, \beta_1, \ldots, \beta_m \leftarrow_R \mathbb{Z}_p$. A outputs (i, t', x') and A wins if $Ver(k_i, t', x') = accept$ and $x' \neq x_i$.
Specifically, here we use the following m-Time MAC procedure: the Dealer generates samples $\alpha_A, \beta_{A,1}, \ldots, \beta_{A,m} \leftarrow_R \mathbb{Z}_p$ and $\alpha_B, \beta_{B,1}, \ldots, \beta_{B,m} \leftarrow_R \mathbb{Z}_p$, and outputs the keys $k_{A,i} = (\alpha_A, \beta_{A,i})$, $k_{B,i} = (\alpha_B, \beta_{B,i})$ and the tags $t_{A,i} = \alpha_B x_A + \beta_{B,i}$, $t_{B,i} = \alpha_A x_B + \beta_{A,i}$ to the parties A, B.
When a party gets a message (a MAC), it uses the function $OpenTo()$, and after the party opens the message it uses its Ver function to verify the message (the MAC): it outputs accept if $t_i = \alpha x + \beta_i$, and rejects (or returns some default value) otherwise.
Then, we perform authenticated secret sharing with wires' values between Alice and Bob by secret sharing the input wires, that propagates secret sharing layer by layer, and once obtained a secret sharing of the output wire, open (=reconstruct).

2.4 Oblivious Transfer

Oblivious Transfer is a protocol with 2 participants (Receiver and Sender). The Receiver want one of the messages the Sender holds. The Receiver sends a choice (an index of which of the messages he wants) and the Sender sends him back the message according to his choice.

2.4.1 1-out-of-2 OT Functionality

The Receiver has an index $i \in \{0,1\}$ and the Sender has two messages m_0, m_1. In the end of the protocol, the Receiver gets m_i and the Sender gets $None$.

2.4.2 1-out-of-n OT Functionality

The Receiver has an index $i \in Z_n$ and the Sender has n messages $m_0, m_1, ..., m_{n-1}$. In the end of the protocol, the Receiver gets m_i and the Sender gets none. To achieve this, the Receiver sends the index $i_j \in \{0,1\}$, the index equals 1 when $j = i$, starting from $j = 0$ till $j = n-1$. The Sender sends two messages back for each i_j. If $i_j = 0$ he sends r_j (a random variable), and if $i_j = 1$ he sends $m_j + \sum_{k=0}^{j} r_k$ (when r_ks are random variables). The Receiver gets n messages from the Sender: $r_0, ..., r_{i-1}, m_j + \sum_{k=0}^{i} r_k, r_{i+1}, ..., r_{n-1}$. The Receiver extracts m_i from all these messages and finally gets what he wants.

2.5 ElGamal Cryptosystem

ElGamal is an implementation of 1-out-of-2 Oblivious Transfer when the protocol is passive. ElGamel cryptosystem consists of 4 algorithms $(Gen, Enc, Dec, Ogen)$ that together make a secure protocol for OT.

Gen: Generate secret and public keys.

Enc: Use the secret key to encrypt the message.

Dec: Use the public key to decrypt the encrypted message.

OGen: Generate a dummy public key.

3 Protocols

3.1 Secure Computation Technique: BeDOZa Protocol With MACs And Oblivious Transfer

Parties: Alice A and Bob B.
Functionality: $f : \mathbb{Z}_p^n \times \mathbb{Z}_p^n \to \mathbb{Z}_p^n \times \perp$, $(x, y) \to (f(x, y), \perp)$.
Circuit: An arithmetic circuit $C : \mathbb{Z}_p^n \times \mathbb{Z}_p^n \to \mathbb{Z}_p$.

L wires $x_1, \ldots, x_L$: $x_1, \ldots, x_n$- Alice's input. $x_{n+1}, ..., x_{2n}$- Bob's input. x_L- output wire.

d layers s.t. inputs to gates at layer $i \in \{1, .., d\}$ are from layers $< i$.

Gates: ADD with constant or of two wires. MULT with constant or of two wires.

Wires' values are secret shared between Alice and Bob:

1. Secret share input wires.
2. Propagates secret sharing layer by layer.
3. Once obtained a secret sharing of the output wire, open (=reconstruct).

Notation: $[x]$ denotes a secret sharing of $x \in \{0, 1\}$,

where Alice holds $x_A \in \mathbb{Z}_p$ and Bob holds $x_B \in \mathbb{Z}_p$ and where: (x_A, x_B) is uniform random in $(\mathbb{Z}_p)^2$ subject to: $(x_A + x_B)$ mod p $= x$.

3.1.1 The Algorithm

Parties: Alice A with input x and circuit c_A, Bob B with input y and circuit c_B. t is the number of mult gates in the circuits c_A, c_B.

3.1.2 Sub-Protocols

Algorithm 1 The Offline Phase | secret share beaver triples and MACs

1: **Repeat t times:**

2: Sample a "Beaver triples": $u, v \leftarrow_R \mathbb{Z}_p$ and $w = (u \cdot v)\ mod\ p$:

a. Alice randomly generates u_A, v_A.

b. Bob randomly generates u_B, v_B, w_B.

c. Alice and Bob use $1 - out - of - p^2\ OT$ to calculate w_A.

Alice is the receiver with choice input $i = p \cdot u_A + v_A$.

Bob is the Sender with message $m_i = ((i//p + u_B) \cdot (i\ mod\ p + v_B) - w_B)\ mod\ p$ i.e. $m_i = ((u_A + u_B) \cdot (v_A + v_B) - w_B)\ mod\ p$ (i is the iteration number of the for in algorithm 5 i.e. in each iteration a new message m_i will be generated).

NOTE The sub-protocol is described in algorithm 5.

3: Alice and Bob generate keys for u_A, v_A, w_A and u_B, v_B, w_B.

4: Alice and Bob use $1 - out - of - p\ OT$ to generate tags for u_A, v_A, w_A.

Alice is receiver with choice input $i = u_A$ for example (or v_A/w_A).

Bob is sender with message $m_i = \alpha_B \cdot i + \beta_B$.

5: Alice and Bob use $1 - out - of - p\ OT$ to generate tags for u_B, v_B, w_B.

Alice is sender with message $m_i = \alpha_A \cdot i + \beta_A$.

Bob is receiver with choice input $i = u_B$ for example (or v_B/w_B).

6: **Repeat** $2 \cdot n$ **times:**

7: Sample a a random value r: $r \leftarrow_R \mathbb{Z}_p$:

a. Alice randomly generates r_A.

b. Bob randomly generates r_B.

8: Alice and Bob generates keys for r_A and r_B: (α_A, β_A) and (α_B, β_B).

9: Alice and Bob use $1 - out - of - p\ OT$ to generate tags for r_A.

Alice is receiver with choice input $i = r_A$.

Bob is sender with message $m_i = \alpha_B \cdot i + \beta_B$.

10: Alice and Bob use $1 - out - of - p\ OT$ to generate tags for r_B.

Alice is sender with message $m_i = \alpha_A \cdot i + \beta_A$.

Bob is receiver with choice input $i = r_B$.

Algorithm 2 The Online Phase | Securely evaluate a circuit C with $\#MULT \leq t$

11: Alice and Bob share their input wires:

$$[x_i] = (x_{iA}, x_{iB}) \leftarrow Share(A, x_i) \text{ for } i = 1, .., n-1$$

$$[x_i] = (x_{iA}, x_{iB}) \leftarrow Share(B, x_i) \text{ for } i = n, .., 2n$$

12: For each layer i, Alice and Bob evaluate all gates in layer i using algorithms 8 and 9.

13: Alice and Bob verify the output value x^L and reconstruct it: $(z, \perp) \leftarrow OpenTo(A, [x^L])$.

Algorithm 3 Sub-protocol | ElGamal Cryptosystem

1: $Gen(1^k)$**:**

1) Sample $sk \leftarrow_R \{0, ..., q-1\}$.

2) Generate $pk = (g, h)$ such that $h = g^{sk}\ mod\ p$.

3) Output (sk, pk).

2: $Enc_{pk}(m)$**:**

1) Sample $r \leftarrow_R \{0, ..., q-1\}$.

2) Generate $C = (c_1, c_2)$ such that $c_1 = g^r\ mod\ p$ and $c_2 = m \cdot h^r\ mod\ p$.

3) Output C.

3: $Dec_{sk}(C)$**:**

1) Generate $m = c_2 \cdot c_1^{-sk}$ i.e. $m = m' \cdot h^r \cdot (g^r)^{-sk}\ mod\ p = m' \cdot (g^{sk})^r \cdot (g^r)^{-sk} = m'$.

2) Output (sk, pk).

4: $OGen(r)$**:**

1) Sample $s \leftarrow_R \{0, ..., p-1\}$.

2) Generate $h = s^2\ mod\ p$.

3) Output $pk = (g, h)$.

Algorithm 4 Sub-protocol | Passive 1-Out-Of-2 Oblivious Transfer from ElGamal

1: $OT_2(p, q, g)$**:** ▷ use ElGamal subprotocol found in algorithm 3

Receiver:

Receiver has a choice bit- 0 or 1.

1) $pk_0, sk = Gen()$.

2) $pk_1 = OGen(random())$.

3) $receiver\_choice = choice$.

Sender:

Sender has 2 messages- m_0 and m_1 and $(pk0, pk1)$.

1) $c_0 = Enc_{pk_0}(m_0)$.

2) $c_1 = Enc_{pk_1}(m_1)$.

3) Send (c_0, c_1) to receiver.

Receiver:

Receiver has $receiver\_choice$, (c_0, c_1) and sk.

1) If $receiver\_choice == 1$ do $m' = c_1$, else $m' = c_0$.

2) $dec'_m = Dec_{sk}(m')$

3) Output dec'_m.

Algorithm 5 Sub-protocol | Passive 1-Out-Of-n Oblivious Transfer from ElGamal

$OT_n(p, q, g)$**:** ▷ use ElGamal in algorithm 3 and OT-2 in algorithm 4
1. Sample $curr\_r \leftarrow_R \{0, ..., p-1\}$. 2. $r \leftarrow 0$. 3. $sum\_r \leftarrow 0$.
for i between $\{0, ..., n-1\}$ **do**

Receiver:

Receiver has a $choice \in \{0, ..., p-1\}$.

$pk_0, sk = Gen()$.
$pk_1 = OGen(random())$.
if $choice == i$ **then**
$receiver\_choice = 1$.
end if
Else $receiver\_choice = 0$.

Sender:

Sender has message m_i and $(pk0, pk1)$.

if $i \neq 0$ **then**
$r = r + curr\_r \ mod \ p$.
end if
$c_0 = Enc_{pk_0}(m + r)$.
$c_1 = Enc_{pk_1}(curr\_r)$.
Send (c_0, c_1) to receiver.

Receiver:

Receiver has $receiver\_choice$, (c_0, c_1) and sk.

if $receiver\_choice == 1$ **then**
$m' = c_0$
end if
Else $m' = c_1$
$curr\_output = Dec_{sk}(m')$.
if $receiver\_choice! = 1$ **then**
$sum\_r = sum\_r + curr\_output \ mod \ p$.
$curr\_r = curr\_output$.
end if
Else Output $dec'_m = curr\_output - sum\_r \ mod \ p$.
end for

Algorithm 6 Sub-protocol | Sharing input wires

1: $Share(A, x_i)$**:**

1) The dealer D outputs a random authenticated secret sharing $[r]$.
2) Alice and Bob run $(r, \perp) \leftarrow OpenTo(A, [r])$.
3) Alice sends Bob $d = x_i – r$.
4) Alice and Bob compute $[x_i] = [r] + d$.

2: $Share(B, x_i)$**:**

1) The dealer D outputs a random authenticated secret sharing $[r]$.
2) Alice and Bob run $(r, \perp) \leftarrow OpenTo(B, [r])$.
3) Bob sends Alice $d = x_i – r$.
4) Alice and Bob compute $[x_i] = [r] + d$.

Algorithm 7 Sub-protocol | Opening secret shared values

1: $OpenTo(A, [x])$**:**

1) Bob sends x_B and $t_{B,x}$ to Alice.
2) Alice outputs $x = (x_A + x_B)$ mod p if $Ver(k_{A,x}, t_{B,x}, x_B) = accept$ (o/w abort).

2: $OpenTo(B, [x])$**:**

1) Alice sends x_A and t_x^A to Bob.
2) Bob outputs $x = (x_A + x_B)$ mod p if $Ver(k_{B,x}, t_{A,x}, x_A) = accept$ (o/w abort).

3: $Open([x])$**:**

1) Run both $OpenTo(B, [x])$ and $OpenTo(A, [x])$.

Algorithm 8 Sub-protocol | Evaluating ADD gates

1: $ADD([x], c)$**:**

1) Alice outputs $(z_A = x_A + c \ mod \ p, k_{A,z} = (\alpha_{A,x_A}, \beta_{A,x_A}), t_{A,z} = t_{A,x_A})$.
2) Bob outputs $(z_B = x_B, k_{B,z} = (\alpha_{B,x_B}, \beta_{B,x_B} - c \cdot \alpha_{B,x_B}), t_{B,z} = t_{B,x_B})$.

2: $ADD([x], [y])$**:**

1) Alice outputs $(z_A = x_A + y_A \ mod \ p, k_{A,z} = (\alpha_{A,x_A}, \beta_{A,x_A} + \beta_{A,y_A}), t_{A,z} = t_{A,x_A} + t_{A,y_A})$.
2) Bob outputs $(z_B = x_B + y_A, k_{B,z} = (\alpha_{B,x_B}, \beta_{B,x_B} + \beta_{B,y_B}), t_{B,z} = t_{B,x_B} + t_{B,y_B})$.

Algorithm 9 Sub-protocol | Evaluating MULT gates

$MULT([x], c)$**:**

1) Alice outputs $(z_A = x_A \cdot c \ mod \ p, k_{A,z} = (\alpha_{A,x_A}, \beta_{A,x_A} \cdot c \ mod \ p), t_{A,z} = t_{A,x_A} \cdot c \ mod \ p)$.
2) Bob outputs $(z_B = x_B \cdot c \ mod \ p, k_{B,z} = (\alpha_{B,x_B}, \beta_{B,x_B} \cdot c \ mod \ p), t_{B,z} = t_{B,x_B} \cdot c \ mod \ p)$.

$MULT([x], [y])$**:**

1) $[d] \leftarrow SUB([x], [u])$ and $d \leftarrow Open([d])$.
2) $[e] \leftarrow SUB([y], [v])$ and $e \leftarrow Open([e])$.
3) Alice and Bob compute $[z] = ADD(ADD([w], ADD(MULT([v], d), MULT([u], e))), e \cdot d)$.
4) Alice outputs z_A and Bob outputs z_B.

Algorithm 10 Sub-protocol | Evaluating Verification function

$Ver(A, k_i, t, x)$**:**
 Alice outputs accept if $t = \alpha_i \cdot x + \beta_i$, else reject o/w.
$Ver(B, k_i, t, x)$**:**
 Bob outputs accept if $t = \alpha_i \cdot x + \beta_i$, else reject o/w.

3.2 Application: BeDOZa Protocol With MACs And Oblivious Transfer

Algorithm 11 Enhanced BeDOZa Protocol

1. $n \leftarrow 2$. 2. $p \leftarrow 107$. 3. $q \leftarrow 53$. 4. $g \leftarrow 2$.

$number\_of\_curr\_MULT \leftarrow 0$.

Generate two global lists e, d.

Generate two random vectors $\vec{a} = (a_1, a_2), \vec{x} = (x_1, x_2)$.

Generate an arithmetic circuit (drawings in figures 3 and 4) that represent the function:

$$f_{\vec{a},4}(x) = \begin{cases} 1 & a_1x_1 + a_2x_2 \geq 4 \\ 0 & otherwise \end{cases}$$

The circuit is represented by an tree as follows:

The circuit is then made up of a collection of node instances that are interconnected to form a larger computational graph.

Any node in the tree represents a node in a circuit, which can have two input parents and an operator (op) that defines the operation to be performed on the parents' values.

Algorithm 12 The Offline Phase | Alice and Bob

Alice generates 2 arrays u_A, v_A (with $\#MULT\_gates$ cells) of random values from $\mathbb{Z}_p$.

Bob generates 3 arrays u_B, v_B, w_B (with $\#MULT\_gates$ cells) of random values from $\mathbb{Z}_p$.

Alice (receiver with input $i = p \cdot u_A + v_A$) and Bob (sender with message $m_i = ((i//p + u_B) \cdot (i \ mod \ p + v_B) - w_B) \ mod \ p$) use $1 - out - of - p^2 \ OT$ to calculate w_A.

Alice and Bob generate key lists for u_A, v_A, w_A and u_B, v_B, w_B.

Alice (receiver) and Bob (sender) use $1 - out - of - p \ OT$ to generate tags for u_A, v_A, w_A.

Alice (sender) and Bob (receiver) use $1 - out - of - p \ OT$ to generate tags for u_B, v_B, w_B.

Alice and Bob generate 2 arrays r_A and r_B (with $2 \cdot n$ cells) of random values from $\mathbb{Z}_p$.

Alice and Bob generate key lists for r_A and r_B.

Alice (receiver) and Bob (sender) use $1 - out - of - p \ OT$ to generate tags for r_A.

Alice (sender) and Bob (receiver) use $1 - out - of - p \ OT$ to generate tags for r_B.

Algorithm 13 The Online Phase | Alice and Bob

Alice and Bob share their input wires:

1. Bob sends $r_B[0, ..., n-1]$ to Alice.
2. Alice computes $r \leftarrow OpenTo(A, [r[0, ..., n-1]])$.
3. Alice computes $d[i] = x[i] - r[i]$ for any $i \in \{0, ..., n-1\}$ and sends d to Bob.
4. Alice and Bob compute $[x_i] \leftarrow ADD([r_i], d_i)$ for any $i \in \{0, ..., n-1\}$.
5. Alice sends $r_A[n, ..., 2n-1]$ to Bob.
6. Bob computes $r \leftarrow OpenTo(B, [r[n, ..., 2n-1]])$.
7. Bob computes $d[i] = x[i] - r[i]$ for any $i \in \{n, ..., 2n-1\}$ and sends d to Alice.
8. Alice and Bob compute $[x_i] \leftarrow ADD([r_i], d_i)$ for any $i \in \{n, ..., 2n-1\}$.

Alice and Bob initialize their *visited* list with all the sons of the roots of the circuit.
Alice and Bob initialize their *occurred_nodes* list with all the roots of the tree.
while there is a node in the circuit that has not yet been occurred **do**

Algorithm 14 Alice computes $(z_A, k_{A,z}, t_{A,z}), MULT$ and sends $MULTS$ to Bob

Create empty list *mults*.
for every node in $visited_A$ list **do**
  if the node's gate is $ADD([x], c)$ **then**
    $node.value, node.key, node.tag = ADD([x], c)$.
  end if
  if the node's gate is $ADD([x], [y])$ **then**
    $node.value, node.key, node.tag = ADD([x], [y])$.
  end if
  if the node's gate is $MULT([x], c)$ **then**
    $node.value, node.key, node.tag = MULT([x], c)$.
  end if
  if the node's gate is $MULT([x], [y])$ **then**
    $q \leftarrow number\_of\_curr\_MULT$
    $node.value = q$.
    $d_A \leftarrow (x_A - u_A)\ mod\ p$.
    $e_A \leftarrow (y_A - v_A)\ mod\ p$.
    In order to compute e, d: $MULTS[i] = [d_A, e_A, q]$) mod p.
    Add *node* to *mults* list and $number\_of\_curr\_MULT += 1$.
  end if
  Add *node* to $occurred_nodes$ list.
  for every *son* of *node* **do**
    if 2 *node*'s parents in $occurred_nodes$ list **then**
      Add *son* to *new_visited* list.
    end if
  end for
end for
$visited = new\_visited$ and $new\_visited.clear()$.

Algorithm 15 Bob computes and sends $(z_B, k_{B,z}, t_{B,z})$ to Alice

for every node in $visited_B$ list **do**
  if the node's gate is $ADD([x], c)$ **then**
    $node.value, node.key, node.tag = ADD([x], c)$.
  end if
  if the node's gate is $ADD([x], [y])$ **then**
    $node.value, node.key, node.tag = ADD([x], [y])$.
  end if
  if the node's gate is $MULT([x], c)$ **then**
    $node.value, node.key, node.tag = MULT([x], c)$.
  end if
  if the node's gate is is $MULT([x], [y])$ **then**
    $q \leftarrow MULTS[0][2]$.
    $d_A = MULTS[0][0]$.
    $d_B \leftarrow (x_B - u_B)\ mod\ p$.
    $e_A = MULTS[0][1]$.
    $e_B \leftarrow (y_B - v_B)\ mod\ p$.
    $d[i] \leftarrow OpenTo(B, [d])$.
    $e[i] \leftarrow OpenTo(B, [d])$.
    $node.value, node.key, node.tag = MULT([x], [y], [u[q]], [v[q]], [w[q]], d[-1], e[-1])$.
    Delete $MULTS[0]$.
  end if
  Add $node$ to $occurred_nodes$ list.
  for every son of $node$ **do**
    if 2 $node$'s parents in $occurred_nodes$ list **then**
      Add son to $new\_visited$ list.
    end if
  end for
end for
$visited = new\_visited$.
$new\_visited.clear()$.

Algorithm 16 Alice receives $(z_B, k_{B,z}, t_{B,z})$ from Bob

for $mult$ in $mults$ list **do**
  $q \leftarrow mult.value$.
  $mult.value, mult.key, mult.tag = MULT([x], [y], [u[q]], [v[q]], [w[q]], d[0], e[0])$.
  Delete $d[0]$, $e[0]$.
end for
Delete $mults$.
end while

Algorithm 17 End Of The Protocol

Alice computes and outputs $z \leftarrow OpenTo(A, [z])$.

4 Implementation

The code is written in Python. Numpy library is the only one required to run the code.
Our code generates 2 random inputs $\vec{a} = (a_1, a_2)$ and $\vec{x} = (x_1, x_2)$ and returns an output z, the result of $f_{\vec{a},4}(x_1, x_2)$.
The code is here:
`https://colab.research.google.com/drive/1R3nxZolFg_X5wgRGJUBjrSrJ1RTTY8qS?usp=sharing`.

5 Empirical Evaluation

Each row in the table is an experiment: we conducted 256 experiments on all possible inputs of a_1, a_2 and x_1, x_2 for which we got output z:

Table 1: Our experiments.

Begin of Table				
x_1	x_2	a_1	a_2	z
0	0	0	0	0
0	0	0	1	0
0	0	0	2	0
0	0	1	0	0
0	0	2	0	0
0	0	0	3	0
0	0	1	1	0
0	0	2	1	0
0	0	1	2	0
0	0	2	2	0
0	0	3	0	0
0	0	1	3	0
0	0	2	3	0
0	0	3	1	0
0	0	3	2	0
0	0	3	3	0
0	1	0	0	0
0	1	0	1	0
0	1	0	2	0
0	1	1	0	0
0	1	2	0	0
0	1	0	3	0
0	1	1	1	0
0	1	2	1	0
0	1	1	2	0
0	1	2	2	0

Continuation of Table 1				
x_1	x_2	a_1	a_2	z
0	1	3	0	0
0	1	1	3	0
0	1	2	3	0
0	1	3	1	0
0	1	3	2	0
0	1	3	3	0
0	2	0	0	0
0	2	0	1	0
0	2	0	2	1
0	2	1	0	0
0	2	2	0	0
0	2	0	3	1
0	2	1	1	0
0	2	2	1	0
0	2	1	2	1
0	2	2	2	1
0	2	3	0	0
0	2	1	3	1
0	2	2	3	1
0	2	3	1	0
0	2	3	2	1
0	2	3	3	1
1	0	0	0	0
1	0	0	1	0
1	0	0	2	0
1	0	1	0	0
1	0	2	0	0
1	0	0	3	0
1	0	1	1	0
1	0	2	1	0
1	0	1	2	0
1	0	2	2	0
1	0	3	0	0
1	0	1	3	0
1	0	2	3	0
1	0	3	1	0
1	0	3	2	0
1	0	3	3	0
2	0	0	0	0
2	0	0	1	0
2	0	0	2	0

Continuation of Table 1				
x_1	x_2	a_1	a_2	z
2	0	1	0	0
2	0	2	0	1
2	0	0	3	0
2	0	1	1	0
2	0	2	1	1
2	0	1	2	0
2	0	2	2	1
2	0	3	0	1
2	0	1	3	0
2	0	2	3	1
2	0	3	1	1
2	0	3	2	1
2	0	3	3	1
0	3	0	0	0
0	3	0	1	0
0	3	0	2	1
0	3	1	0	0
0	3	2	0	0
0	3	0	3	1
0	3	1	1	0
0	3	2	1	0
0	3	1	2	1
0	3	2	2	1
0	3	3	0	0
0	3	1	3	1
0	3	2	3	1
0	3	3	1	0
0	3	3	2	1
0	3	3	3	1
1	1	0	0	0
1	1	0	1	0
1	1	0	2	0
1	1	1	0	0
1	1	2	0	0
1	1	0	3	0
1	1	1	1	0
1	1	2	1	0
1	1	1	2	0
1	1	2	2	1
1	1	3	0	0
1	1	1	3	1

Continuation of Table 1				
x_1	x_2	a_1	a_2	z
1	1	2	3	1
1	1	3	1	1
1	1	3	2	1
1	1	3	3	1
2	1	0	0	0
2	1	0	1	0
2	1	0	2	0
2	1	1	0	0
2	1	2	0	1
2	1	0	3	0
2	1	1	1	0
2	1	2	1	1
2	1	1	2	1
2	1	2	2	1
2	1	3	0	1
2	1	1	3	1
2	1	2	3	1
2	1	3	1	1
2	1	3	2	1
2	1	3	3	1
1	2	0	0	0
1	2	0	1	0
1	2	0	2	1
1	2	1	0	0
1	2	2	0	0
1	2	0	3	1
1	2	1	1	0
1	2	2	1	1
1	2	1	2	1
1	2	2	2	1
1	2	3	0	0
1	2	1	3	1
1	2	2	3	1
1	2	3	1	1
1	2	3	2	1
1	2	3	3	1
2	2	0	0	0
2	2	0	1	0
2	2	0	2	1
2	2	1	0	0
2	2	2	0	1

Continuation of Table 1				
x_1	x_2	a_1	a_2	z
2	2	0	3	1
2	2	1	1	1
2	2	2	1	1
2	2	1	2	1
2	2	2	2	1
2	2	3	0	1
2	2	1	3	1
2	2	2	3	1
2	2	3	1	1
2	2	3	2	1
2	2	3	3	1
3	0	0	0	0
3	0	0	1	0
3	0	0	2	0
3	0	1	0	0
3	0	2	0	1
3	0	0	3	0
3	0	1	1	0
3	0	2	1	1
3	0	1	2	0
3	0	2	2	1
3	0	3	0	1
3	0	1	3	0
3	0	2	3	1
3	0	3	1	1
3	0	3	2	1
3	0	3	3	1
1	3	0	0	0
1	3	0	1	0
1	3	0	2	1
1	3	1	0	0
1	3	2	0	0
1	3	0	3	1
1	3	1	1	1
1	3	2	1	1
1	3	1	2	1
1	3	2	2	1
1	3	3	0	0
1	3	1	3	1
1	3	2	3	1
1	3	3	1	1

Continuation of Table 1				
x_1	x_2	a_1	a_2	z
1	3	3	2	1
1	3	3	3	1
2	3	0	0	0
2	3	0	1	0
2	3	0	2	1
2	3	1	0	0
2	3	2	0	1
2	3	0	3	1
2	3	1	1	1
2	3	2	1	1
2	3	1	2	1
2	3	2	2	1
2	3	3	0	1
2	3	1	3	1
2	3	2	3	1
2	3	3	1	1
2	3	3	2	1
2	3	3	3	1
3	1	0	0	0
3	1	0	1	0
3	1	0	2	0
3	1	1	0	0
3	1	2	0	1
3	1	0	3	0
3	1	1	1	1
3	1	2	1	1
3	1	1	2	1
3	1	2	2	1
3	1	3	0	1
3	1	1	3	1
3	1	2	3	1
3	1	3	1	1
3	1	3	2	1
3	1	3	3	1
3	2	0	0	0
3	2	0	1	0
3	2	0	2	1
3	2	1	0	0
3	2	2	0	1
3	2	0	3	1
3	2	1	1	1

Continuation of Table 1				
x_1	x_2	a_1	a_2	z
3	2	2	1	1
3	2	1	2	1
3	2	2	2	1
3	2	3	0	1
3	2	1	3	1
3	2	2	3	1
3	2	3	1	1
3	2	3	2	1
3	2	3	3	1
3	3	0	0	0
3	3	0	1	0
3	3	0	2	1
3	3	1	0	0
3	3	2	0	1
3	3	0	3	1
3	3	1	1	1
3	3	2	1	1
3	3	1	2	1
3	3	2	2	1
3	3	3	0	1
3	3	1	3	1
3	3	2	3	1
3	3	3	1	1
3	3	3	2	1
3	3	3	3	1
End of Table				

Note that for each a_1, a_2, x_1, x_2 we got z which corresponds to the result of the function $f_{\vec{a},4}(x_1, x_2)$ for these a_1, a_2, x_1, x_2:

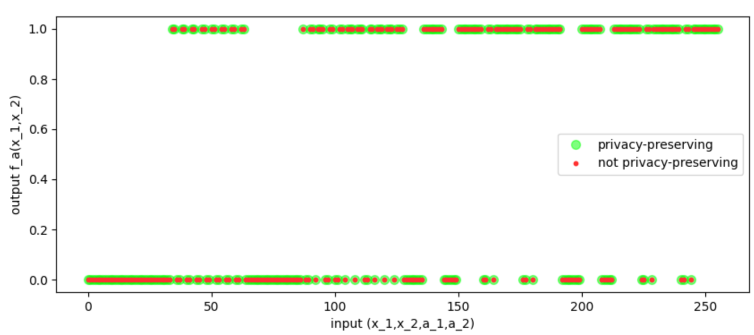

Figure 1: Comparison between privacy-preserving computation and not privacy-preserving computation

Note: You can see our tests file here: https://colab.research.google.com/drive/19-HjA2zUSsp2gk8oDMTsmTFyQ9yFkTI3?usp=sharing and the Benchmark tests file here: https://colab.research.google.com/drive/1N1CgFOL9nwuS8u1Ubnwp2wiMsExpjubF?usp=sharing.

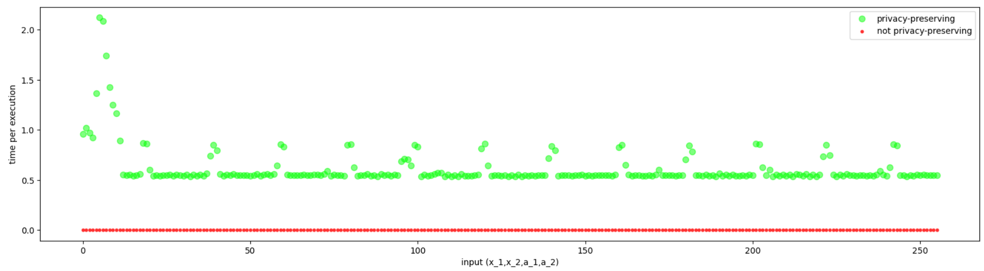

Figure 2: Running time comparison between privacy-preserving computation and not privacy-preserving computation

6 Conclusions

As shown in the results in the previous seciton, We see that the proposed approach yields correct results for $f_{\vec{a},4}(x_1, x_2)$. Therefore, the output correctness of the proposed approach is not compromised by considering privacy.
By using BeDOZa protocol, we were able to maintain participants' (Alice and Bob) privacy because the private data didn't need to be disclosed for computations.

References

[BDOZ10] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. Cryptology ePrint Archive, Paper 2010/514, 2010. https://eprint.iacr.org/2010/514.

[Wik] Wikipedia. Adversary.

Appendices

A Drawings of The Arithmetic Circuit From Homework 1

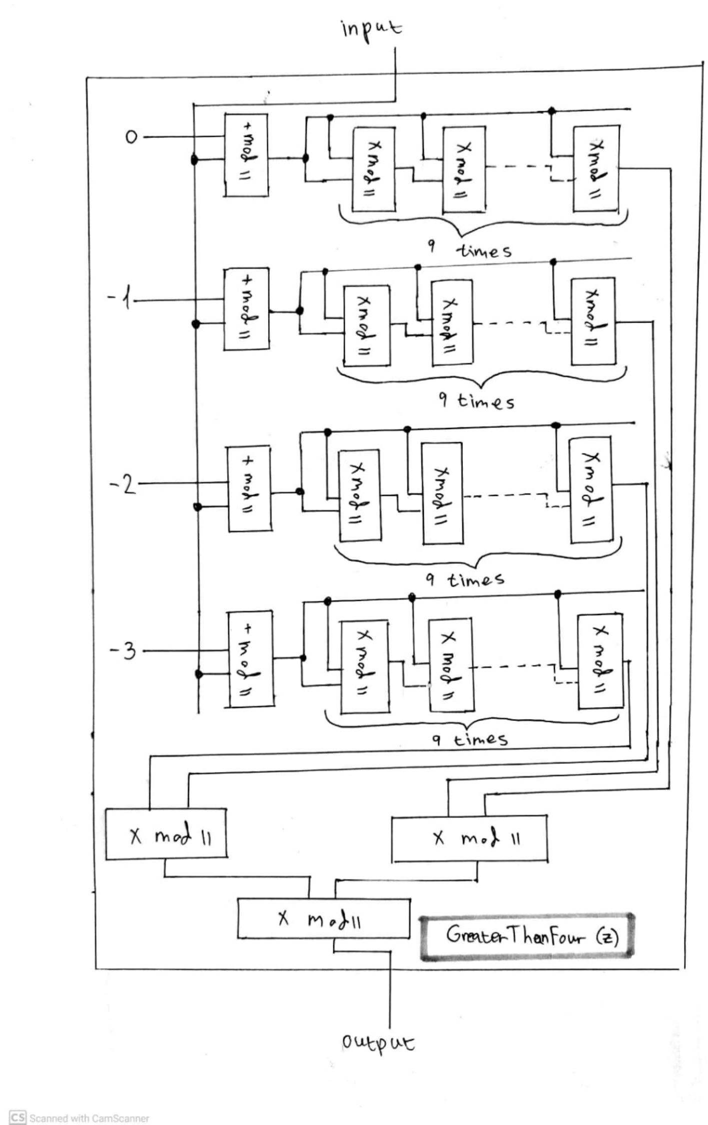

Figure 3: A black box of Greater Than

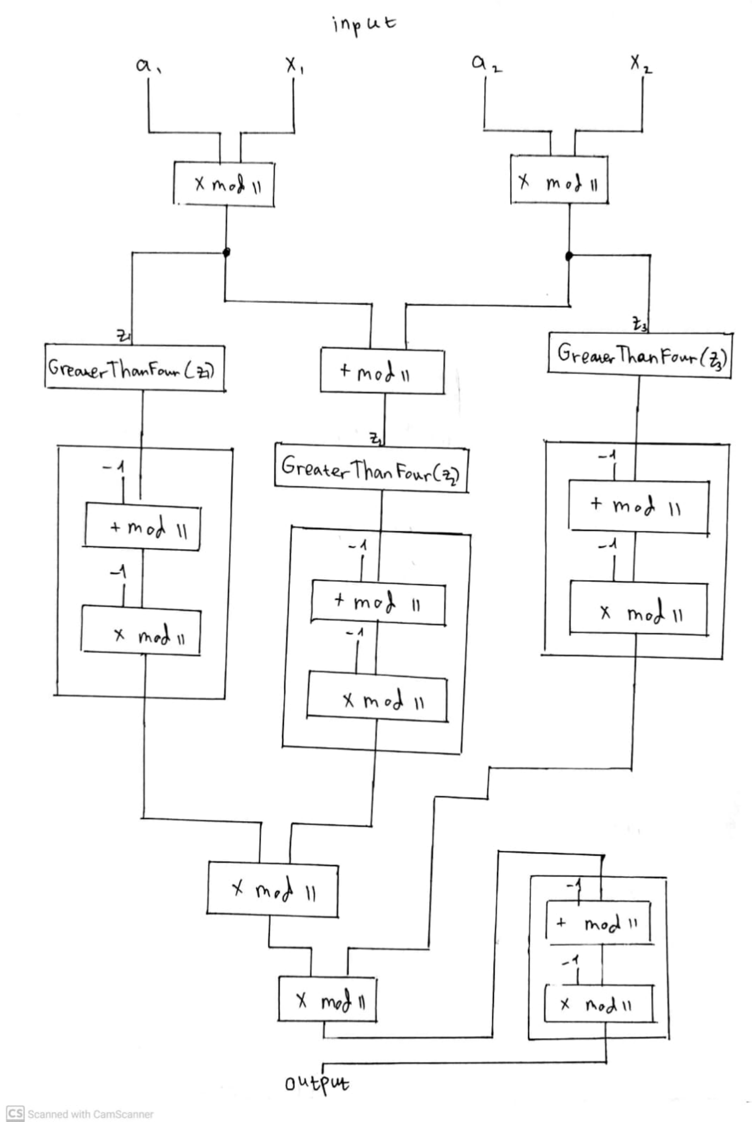

Figure 4: The entire Arithmetic circuit