Evolving Cryptography: Advancements Through Fundamentals

Course Project Report

Course Code: CS670

Course Mentor: Aditya Vadapalli

Name: Raaghav Jain
Roll Number: 220844
Date: 18/04/2024

Department of Computer Science
IIT Kanpur

Objectives

- Implement a CPIR protocol based on Quadratic Residues and Quadratic Non-Residues.
- Perform oblivious comparisons and conditional oblivious swaps to maintain data privacy.
- Optimize the protocol for efficiency in terms of computation and communication.

CPIR protocol based on Quadratic residues

We first define the notion of *Quadratic residues*:
Defination: A number β is called a quadratic residue (QR) modulo **m** if there exists at least one solution **Z** to the given congruence:

$$Z^2 \equiv \beta \pmod{m}$$

and if no such Z exists, then β is termed as a **quadratic non-residue**(QNR).
Using the Legendre symbol notation,

$$\left(\frac{\beta}{m}\right) = \begin{cases} 1 & \beta \text{ is QR} \\ -1 & \beta \text{ is QNR} \end{cases}$$

Some useful properties:

1. $\left(\frac{1}{m}\right) = 1$
2. $\left(\frac{ab}{m}\right) = \left(\frac{a}{m}\right)\left(\frac{b}{m}\right)$

$\longrightarrow$ ab is a QR if both a and b are QNR $\mid (-1).(-1) = 1$
$\longrightarrow$ ab is a QR if both a and b are QR $\mid (1).(1) = 1$
$\longrightarrow$ ab is a QNR if only a or b is a QNR and the other is QR $\mid (1).(-1) = -1$

These three properties are the result of the properties of Legendre symbol.

Algorithm: Suppose the database D is stored in the server S where
$D = (X_1, X_2 ..., X_{n-1}, X_n) \quad \mid X_i \in \{0, 1\}$
The user who wants to retrieve the k^{th} bit generates $n-1$ QRs denoted by $a_1, a_2, a_3, .., a_{k-1}, a_{k+1}, .., a_n$ and a QNR b_k modulo m (which is a secret).

The user sends these $n-1+1$ numbers to S as

$$
\begin{array}{lcl}
a_1 & \rightarrow & U_1 \\
a_2 & \rightarrow & U_2 \\
. & & . \\
. & & . \\
. & & . \\
a_{k-1} & \rightarrow & U_{k-1} \\
b_k & \rightarrow & U_k \\
a_{k+1} & \rightarrow & U_{k+1} \\
. & & . \\
. & & . \\
. & & . \\
a_n & \rightarrow & U_n
\end{array}
$$

Now, S will return the following product:

$$J = U_1^{X_1} \cdot U_2^{X_2} \cdot ... \cdot U_n^{X_n}$$

We use this secrecy of m as without knowing the value of m, S **CANNOT** distinguish between a QR and a QNR (*quadratic residue assumption*)

Knowing J, the user evaluates $\left(\frac{J}{m}\right) = \left(\frac{U_1^{X_1}}{m}\right) \cdot \left(\frac{U_2^{X_2}}{m}\right) \cdot .... \cdot \left(\frac{U_n^{X_n}}{m}\right) \leftarrow$ prop.2

Notice that:

$$\left(\frac{U_z^{X_z}}{m}\right) = \begin{cases} 1 & X_m = 1 \text{ or when } U_z \text{ is a QR or both} \\ -1 & X_m = 1 \text{ and } U_z \text{ is a QNR} \end{cases}$$

This implies that all the terms of $\left(\frac{J}{m}\right)$ will simplify as:

$$\left(\frac{J}{m}\right) = \left(\frac{U_k^{X_k}}{m}\right) = \begin{cases} 1 & X_k = 0 \leftarrow \text{ prop.1} \\ -1 & X_k = 1 \leftarrow \text{ defination} \end{cases}$$

Thus, the user receives the value of X_k bit from D without disclosing k.

Is this SPIR?

For this CPIR to be an SPIR, the user should only get the value of X_k bit and nothing else.

We analyse our product J to check this.

Fundamental theorem of arithematic

Every positive integer $n > 1$ can be expressed uniquely as $n = p_1^{a_1} \cdot p_2^{a_2} \cdot \ldots \cdot p_k^{a_k}$,

where $p_1, p_2, \ldots, p_k$ are distinct prime numbers and $a_1, a_2, \ldots, a_k$ are positive integers.

How is this relevant?

Using the theorem stated above, we conclude that J can also be written in such a form and the power a_i of any prime p_i will be some weighted partial sum of the bits of the database D.

Example: Taking m =17, we have the following QRs $\{1, 2, 4, 8, 9, 13, 15, 16\}$ and QNRs $\{3, 5, 6, 7, 10, 11, 12, 14\}$ let the $D \in (0, 1)^6$.
According to the protocol, the user now chooses $6 - 1 = 5$ QRs which are {2,8,9,13,15} and 1 QNR that is 14 for retrieving X_3.
The user now sends $U_1, U_2 ... U_6$ and in return, S send the product J. On writing the product J in the format of product of primes:

$$J = 2^{(X_1+3 \cdot X_2+X_3)} \cdot 3^{(2 \cdot X_4+X_6)} \cdot 5^{X_6} \cdot 7^{X_3} \cdot 13^{X_5}$$

Now for all such primes having power of unibit dependency, the user could know this bit as the user knows U_t, and:

$$X_t = \begin{cases} 1 & \text{if } U_t \mid J \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

(Here specifically t can be 3,5 or 6)

Thus, the user knows more than just X_3

When can this be seen?

During the choice of the QRs, if there exists a QR U_i such that:

$$gcd(U_i, U_j) = 1 \quad \forall \ j \neq i \tag{2}$$

then, X_i is retrievable other than the QNR one. The value of the bit is described in (1) above.

Hence, the quadratic residue based PIR scheme is **NOT** a symmetric PIR scheme. It is symmetric in a specific case, when we are not able to locate any such U_i holding (2).

Is this computationally easy?

To find such unibit powers, the user needs to express each of the U_i(s) as the product of powers of primes which is easy for the chosen set if factorizing them is easy.(since, U_i can be chosen small by the user and thus factorizing becomes easy). Then the user calculates the product of these (P) and express them in powers of primes.

$$\begin{aligned}
P &= U_1 \cdot U_2 \cdot U_3 \cdot \ldots \cdot U_n \\
&= (p_1^{\alpha_1} \cdot p_2^{\beta_1} \cdot \ldots) \cdot (p_1^{\alpha_2} \cdot p_2^{\beta_2} \cdot \ldots) \cdot \ldots \\
&= p_1^{\sum \alpha_i} \cdot p_2^{\sum \beta_i} \cdot \ldots
\end{aligned}$$

For any prime p_k if it's power that is $\sum \gamma_i = 1$, it means that p_k can only divide the product J at max once, which further implies the value of the bit, the database is holding there.

```cpp
// This code is created to retrieve QRs and QNRs based on
   database size and bit size of prime P"
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <gmp.h>

// Function to check if a number is a quadratic residue modulo p
bool isQuadraticResidue(mpz_t a, mpz_t p) {
    mpz_t result;
    mpz_init(result);
    mpz_powm_ui(result, a, 2, p); // result = a^2 mod p
    bool isQR = mpz_legendre(a, p) == 1; // Check if result is 1
    mpz_clear(result);
    return isQR;
}

// Function to generate a random prime number of specified bit
   length
```

```
void generatePrimes(mpz_t p, int bit_length) {
    gmp_randstate_t state;
    gmp_randinit_mt(state);
    gmp_randseed_ui(state, time(NULL));
    mpz_urandomb(p, state, bit_length);
    mpz_nextprime(p, p);
    gmp_randclear(state);
}

// Function to generate QRs and QNRs modulo p
void generateQRsAndQNRs(std::vector<mpz_t>& QRs,
    std::vector<mpz_t>& QNRs, mpz_t p) {
    mpz_t a, result;
    mpz_inits(a, result, NULL);

    for (mpz_set_ui(a, 1); mpz_cmp(a, p) < 0; mpz_add_ui(a, a,
        1)) {
        if (isQuadraticResidue(a, p)) {
            mpz_t qr;
            mpz_init_set(qr, a);
            QRs.push_back(qr);
        } else {
            mpz_t qnr;
            mpz_init_set(qnr, a);
            QNRs.push_back(qnr);
        }
    }

    mpz_clears(a, result, NULL);
}

// Function to choose random QRs and QNR
void chooseRandomElements(const std::vector<mpz_t>& QRs, const
    std::vector<mpz_t>& QNRs, std::vector<mpz_t>& selectedQRs,
    mpz_t& selectedQNR, int n) {
    srand(time(NULL));
    std::vector<bool> chosen(QRs.size(), false);
    while (selectedQRs.size() < n - 1) {
        int index = rand() % QRs.size();
        if (!chosen[index]) {
            mpz_t qr;
            mpz_init_set(qr, QRs[index]);
            selectedQRs.push_back(qr);
```

```cpp
            chosen[index] = true;
        }
    }

    int index = rand() % QNRs.size();
    mpz_init_set(selectedQNR, QNRs[index]);
}

int main() {
    int bit_length = 10; // Length of the prime number in bits
    int n = 10; // Size of the database

    mpz_t p;
    mpz_init(p);
    generatePrimes(p, bit_length);

    std::vector<mpz_t> QRs, QNRs;
    generateQRsAndQNRs(QRs, QNRs, p);

    std::vector<mpz_t> selectedQRs;
    mpz_t selectedQNR;
    chooseRandomElements(QRs, QNRs, selectedQRs, selectedQNR, n);

    std::cout << "Prime number P: ";
    mpz_out_str(stdout, 10, p);
    std::cout << std::endl;

    std::cout << "Selected QRs: " << std::endl;
    for (const auto& qr : selectedQRs) {
        mpz_out_str(stdout, 10, qr);
        std::cout << " ";
    }
    std::cout << std::endl;

    std::cout << "Selected QNR: ";
    mpz_out_str(stdout, 10, selectedQNR);
    std::cout << std::endl;

    mpz_clear(p);
    for (auto& qr : QRs) mpz_clear(qr);
    for (auto& qnr : QNRs) mpz_clear(qnr);
    for (auto& qr : selectedQRs) mpz_clear(qr);
    mpz_clear(selectedQNR);
```

```
    return 0;
}
```

Oblivious Datastructures

- **Problem Context**:
  - Designing an Oblivious Min Heap implementation within a secure multiparty computation framework.
- **Objective**:
  - Develop protocols for secure Insert and Extract Min operations on a shared array (A) split into additive shares (A0 and A1) between two parties (P0 and P1).
- **Challenges**:
  - Maintaining the heap property ($A0[i]+A1[i] \leq A0[2i]+A1[2i]$ and $A0[i]+A1[i] \leq A0[2i+1]+A1[2i+1]$) after each operation without leaking sensitive information.
  - Ensuring efficient and secure communication and computation between the parties.
- **Components**:
  - **Insert Operation**:
    * Expand the array to accommodate a new element using additive shares of the new element (M0 and M1).
    * Restore the heap property using an oblivious comparisons black box to compare and adjust node values.
  - **Extract Min Operation**:
    * Remove the root element (A0[1], A1[1]) and restructure the heap to maintain the heap property.
    * Utilize oblivious comparisons to ensure correctness and security in heap restoration.
- **Methodology**:

- Implement cryptographic protocols and algorithms for secure multiparty computation to achieve oblivious operations on the shared data structure.
- Utilize the concept of additive sharing and oblivious comparisons to maintain data privacy and integrity while performing heap operations.

- **Expected Outcome**:
  - A secure implementation of an Oblivious Min Heap that allows multiple parties to perform Insert and Extract Min operations on shared data without compromising confidentiality or introducing vulnerabilities.

INSERT operation

a) P_0 writes M_0 in $A_0[n+2]$ while P_1 writes M_1 in $A_1[n+2]$

b) By simply writing these in the last index will not ensure the condition that the value of M is greater than or equal to its parent. There will exist a unique place for M_0 and M_1 such that $M_0 + M_1 \geq$ parent node of M in its path till the root node. The contrary is very much possible by just appending M_0 and M_1 at $A_0[n+2]$ and $A_1[n+2]$ respectively thus violating the heap property.

c) to ensure the correct positioning of the shares or M such that the heap property is ensured, we use the two functionalities:

   i. **Oblivious comparison**:

   $\longrightarrow$ compares two values (X,Y) having additive shares such that $x_0 + x_1 = X$ and $y_0 + y_1 = Y$

   $\longrightarrow$ P_0 sends x_0, y_0 while P_1 sends x_1, y_1

   $\longrightarrow$ functionality returns c_0 and c_1 to both the servers.

   $\longrightarrow$ if $X < Y$ then $c_0 + c_1 = 1$ or else $c_0 + c_1 = 0$

   (using this as a black box)

   ii. **Oblivious conditional swap** $f(A, B, C)$:

   $\longrightarrow$ takes in six values $(A_0, A_1, B_0, B_1, c_0, c_1)$ having additive shares such that $A_0 + A_1 = A$ and $B_0 + B_1 = B$ and swaps the value based on some condition $C = c_0 + c_1$.

   $\longrightarrow$ P_0 sends A_0, B_0 while P_1 sends A_1, B_1

$\longrightarrow$ functionality updates the A_0 and A_1 to both the servers.
$\longrightarrow$ finally, the swap occurs based on C

for this question we are required to construct such a functionality according to the condition required to ensure the heap property. For this, we obtain this C from oblivious comparison functionality

- For every node $A_{(0/1)}[i]$, observe that the index of it's parent node will be
  $p(i) = \frac{(2i-1)+(-1)^i}{4}$
- Implement oblivious comparison between the child node and parent node such that if child node < parent node, then $c_0 + c_1 = C = 1$ or else $C = 0$
- Mathematically,

$$A_0[i] + A_1[i] < A_0[p(i)] + A_1[p(i)] \longrightarrow C = 1$$

$$while$$

$$A_0[i] + A_1[i] \geq A_0[p(i)] + A_1[p(i)] \longrightarrow C = 0$$

- If we call the updated shares after applying $f()$ as $A'_0[i]$ for $A_0[i]$ and $A'_1[i]$ for $A_1[i]$, then,

$$A'_0[p(i)]+A'_1[p(i)] = A_0[p(i)]+A_1[p(i)]+C\cdot(A_0[i]-A_0[p(i)]+A_1[i]-A_1[p(i)])$$

$$while$$

$$A'_0[i]+A'_1[i] = A_0[i]+A_1[i]-C\cdot(A_0[i]-A_0[p(i)]+A_1[i]-A_1[p(i)])$$

- These set of relations show that when C is 1, the swap of values occur while they remain the same when C is 0.
- Thus, we complete this functionality by using *Secure-Multiparty Computation*'s multiplication functionality, we compute two shares z_0 and z_1 such that,

$$z_0 + z_1 = (c_0 + c_1) \cdot (A_0[i] - A_0[p(i)] + A_1[i] - A_1[p(i)])$$

  and send the new update values to various servers as following:

$$P_0 \longleftarrow A'_0[p(i)] = A_0[p(i)] + z_0 \text{ and } A'_0[i] = A_0[i] - z_0$$

$$P_1 \longleftarrow A'_1[p(i)] = A_1[p(i)] + z_1 \text{ and } A'_1[i] = A_0[i] - z_1$$

- This will ensure an oblivious conditional swap which will swap the values of parent and child if parent has a larger value than the child, otherwise, will remain the same

Finally, we get the following protocol:

$\longrightarrow$ Place M_0 and M_1 at $A_0[n+2]$ and $A_1[n+2]$ respectively

$\longrightarrow$ Apply oblivious condition swap between $A_{(0/1)}[n+2]$ and $A_{(0/1)}[p(n+2)]$

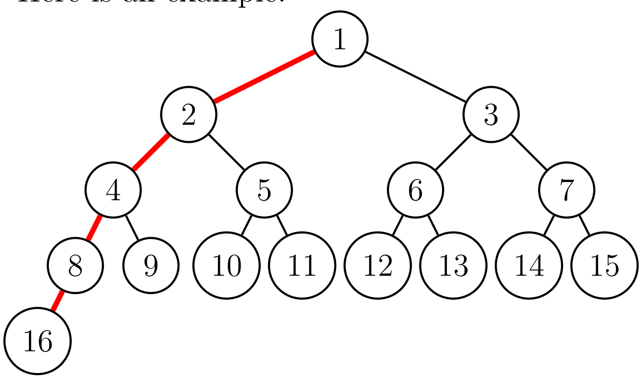
$\longrightarrow$ Then apply this again between $A_{(0/1)}[p(n+2)]$ and $A_{(0/1)}[p(p(n+2))]$

$\longrightarrow$ Continue this, until $p(p(p...(n+2)))$ becomes 1 i.e the path reaches until the root of the heap.

This will place M_0 and M_1 and the required positions while swapping only the required data-blocks but computationally indistinguishable for server hence, done obliviously

---

Here is an example:



Showing for P_0

Here, M_0 is added at the 16^{th} position.

$f(A[16], A[8], C) \longrightarrow f(A[8], A[4], C) \longrightarrow f(A[4], A[2], C) \longrightarrow f(A[2], A[1], C)\ |END$

EXTRACT MIN operation

a) $\longrightarrow$) Extract the current root node shares

$\longrightarrow$) Place the value at $A_0[n+2]$ and $A_1[n+2]$ at $A_0[1]$ and $A_1[1]$ (I am taking n+2 as it is the last data block after M is added. If shares of M were not added, then we must use shares of $A[n+1]$ as the shares of new root node)

b) The heap property is violated as simply placing the last data block shares as the root node shares will not include the inequality among the root node and its child nodes. We will have to re-arrange this which is shown in the next part.

c) **Oblivious algorithm to sort and restore the heap property**: For this, we will use:

   i. Oblivious comparison

   ii. Conditional Oblivious Swap

   iii. *Distributed ORAMs* read and write functionality.(described below)

   P0 and P1 hold shares D_0 and D_1 of the database D, shares i_0^* and i_1^* of the index i^* they wish to access, and shares M_0 and M_1 of the update value M. All shares are additive, so that $i_0^*+i_1^* = i^*$ and similar. The outputs of the read operation are $read_0$ and $read_1$, with the property that $read_0 + read_1 = D[i^*]$. D'_0 and D'_1 are the outputs of the write operation, with $D'_0[i] + D'_1[i] = D[i]$ for $i \neq i^*$, and $D'_0[i^*] + D'_1[i^*] = D[i^*] + M$.

Algorithm:

→ Start with $A[i]$ such that $i = 1$;(Root node)
→ i_0, i_1 be the additive shares if i
→ Compute the additive shares for index $2 \cdot i$ and $2 \cdot i + 1$ [For example: $2i = (2i_0) + (2i_1) \mid 2i + 1 = (2i_0) + (2i_1 + 1)$]
→ Use **Duoram Read** to obtain shares of values of $A[2i]$ and $A[2i + 1]$ (use shares calculated in the previous step.)
→ Obliviously compare these two by the **Oblivious Comparison** such that $c_0 + c_1 = 0$ means $A[2i + 1] > A[2i]$ while $c_0 + c_1 = 1$ otherwise
→ Let $i^{\#}$ having shares $i_0^{\#}$ and $i_1^{\#}$ be the index of smaller child node of $A[i]$. Then

$$i_0^{\#} = 2i_0 + c_0$$
$$i_1^{\#} = 2i_1 + c_1$$

→ Read the value at i and $i^{\#}$ [(i_0, i_1) & $(i_0^{\#}, i_1^{\#})$ are the shares] using Duoram read and use **conditional oblivious swap** to obtain values to be written at these datablocks.
→ Write the value obtained using **Duoram Write** functionality
→ Now, set $i = i^{\#}$ and repeat this until you reach a leaf.