CS 758 project

Navneet Agarwal
Sanat Anand

November 2017

Abstract

We give a C++ implementation of the protocol for Non-interactive cryptocomputing for NC_1 described in [SYY99]. The underlying re-randomizable PKE has been implemented as the ElGamal encryption scheme.

1 Introduction

We have implemented [SYY99] in C++. It is one of the earliest works in homomorphic encryption and solves the problem for NC_1 circuits. It gives a design for a cryptocomputer which computes inattentively on encrypted data and relays the result back to Alice. Say Alice has an input $\mathbf{x}$ and Bob has a circuit $\mathbf{C}$. Alice should learn the value $C(x)$ but nothing else about C that cannot be derived from $C(x)$ alone. Similarly, Bob should learn nothing about Alice's input x.

Our work involves using plain C++ (without any supporting libraries) to implement two parties Alice (client) and Bob (server) which play out the [SYY99] protocol. The inattentive evaluation requires us to use a reramdomizable Public Key Encryption and we've used ElGamal Encryption for that. Alice receives an input (and a security parameter) from its user and Bob receives a Circuit from its user. Alice sends encrypted inputs to Bob which runs it on its Circuit as per the scheme and sends the result back. Alice then decrypts it and prints the output.

The code can be found at `https://github.com/navneetagarwal/Shoup_NTL`

2 Protocol Outline

The solution proposed involves an "input provider" (Alice) sending an encryption of its input to a second party, the "crypto-computer" (Bob). The crypto-computer performs some **inattentive computation** on the input received on the circuit given to it and sends back the output. This gives us a protocol for secure computation in a single round without any additional setup.

This requires a notion of inattentive evaluation (proposed in the Sanders paper). This involves the ideal evaluation of any gate correctly where the evaluation procedure is performed independently of the input and output values. The procedure uniformly performs operations which are not a function of the inputs (oblivious, or 'blind' operations). The input and output distributions, in our case, depend on the depth of the gate within the circuit, but otherwise the output element representation does not reveal any information about the input values. It is crucial that the inattentive evaluation gate does not "view" the basic elements constituting the "inputs to the gate" of the circuit. All it does is perform on them obliviously.

Additionally, the paper introduces a notion of random self-reducible encryption schemes which is just another name for re-randomizable PKE scheme. We must use a re-randomizable PKE scheme to encrypt the elements in Alice (we have used ElGamal scheme in our implementation).

The three properties preserved by the computation in the crypto-computer are:

- Correctness: The decode of the output of a gate must be the gate evaluation of the decode of the two inputs
- Hiding: The resulting output element must be distributed uniformly at random in some distribution and doesn't give any information about the specific inputs or the circuit itself.
- Obliviousness: The operations performed on the inputs must be independent of the values and must be fixed functions.

Our implementation requires the circuit to be specified in terms of just OR and NOT gates (AND gates can be converted to them). We show how the inattentive evaluation is done for them.

1. **NOT gate**: We can represent a single bit $x \in \{0,1\}$ by the pair $(x, \bar{x}) \in \{(0,1),(1,0)\}$. The DECODE procedure on input (x,y) outputs the first element x and thus allows us to recover the represented value. We can now define the algorithm NOT that swaps the components of its input (x,y) and outputs (y,x). Since we have $DECODE(NOT(a))) = DECODE(a) + 1 \bmod 2$, we can now negate these representations easily. This simple observation allows us to perform negation on ciphertexts by moving ciphertexts and we do not need to negate under the encryption.
2. **OR gate**: Given two bits x and y, we represent $OR(x,y) = (x, y, x+y, 1)$ where addition is done modulo 2. Here we note that whenever the OR gate evaluates to 1, i.e. $(x,y) \in \{(0,1),(1,0),(1,1)\}$, the quadruple generated has exactly 3 of its members as 1 and one element as 0. And whenever it evaluates to 0 the quadruple generated has exactly 3 of its members as 0 and one element as 1. Therefore, if we randomly shuffle this quadruple, we lose all input information except the input value.

In order to do the OR operation, we define inattentive addition as $Add(x, y) = (x, y)$. We can represent x and y in the OR operation as $(x, 0)$ and $(y, 0)$ respectively. We can randomize this Add object by: For element (x, y) Make a list $L = \{(x, y), (NOT(x), NOT(y))\}$ and output an element from it randomly. This would work as both of them have the same addition value.

All of these operations must be recursively carried out (as seen in the paper) in order to obtain the correct evaluation for the circuit. We have described how we have done it in our code in the next section.

3 Implementation Details

We have made two files alice.cpp and bob.cpp specifying the sending party and the cryptocomputer respectively. We've used network sockets to make connections between them with Alice as the client and Bob as the server.

Alice receives a safe prime p from its environment as the security parameter. We work in the group $\mathbb{QR}_p$ which is the group of quadratic residues of $\mathbb{Z}_p^*$ because we need the DDH assumption for ElGamal encryption. It generates a secret key y and a corresponding public key $Y = g^y$ where we've set $g = 4$ as the fixed generator (It can be seen that 4 is a generator in every Quadratic Residue group of this form). It then sends (p, g, Y) to Bob.

Alice then receives the input from the environment in the form of bits. It encodes 0 as 1 and 1 as 4 (to ensure they're elements in $\mathbb{QR}_p$). We note that we do not need to preserve the multiplicative homomorphic property of the groups as our scheme needs only the re-randomizability of ElGamal. Then it encrypts the encoding of the input and its negation (using ElGamal encryption and randomness generated by rand() function in C++) and sends the encryption to Bob in form of a string.

Bob then receives the whole string and populates its data structures Enc and Add as per the values it receives. Then it goes ahead and evaluates the circuits gate-by-gate. At the input and Output of every gate, we shall have values stored in form of the Enc data structure. Whenever it encounters a NOT gate, it calls the negation method which generates another Enc structure which represents the negative of the original one (by calling itself recursively until the ciphertext level is reached). Similarly, whenever it encounters an OR gate, it calls the OR method which generates another Enc structure which represents the OR of the two input structures (by calling itself recursively until the ciphertext level is reached making fresh encryptions of that level for constants 0 and 1 as required).

Once the evaluation is complete, Bob has a level-k ciphertext in hand in an Enc structure. It then calls the randomize method on it. This method calls itself recursively on all its children and does appropriate randomization for each Enc (by permuting the entries) and Add (by picking randomly from the two tuples which represent the same Add value (x, y) and $(NOT(x), NOT(y))$). At the base level of ciphertext, it does the ElGamal re-randomization. It then sends this to Alice.

Alice calls decrypt on this level-k ciphertext using its secret key which calls

itself recusively and uses the ElGamal secret key for decrypting base level ciphertexts. It decodes the output into 0 (from 1) and 1 (from 4) and prints it.

4 Testing and experimentation

We have tested our code for the following circuits and have received the correct output.

Figure 1: AND gate

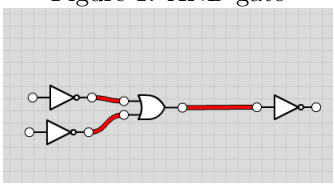


Figure 2: 2-layer OR gate

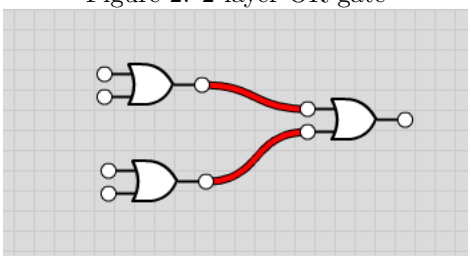
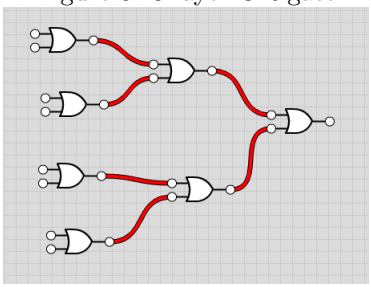


Figure 3: 3-layer OR gate



Additionally, we have run our code on circuits of varying OR-depths and have plotted the time taken for evaluating them as per our protocol.

The plot obtained is as above. All the gates we've considered have fan-in of 2. So, we cannot make large shallow circuits to test for our implementation.

Figure 4: The deep OR circuit used for the graph

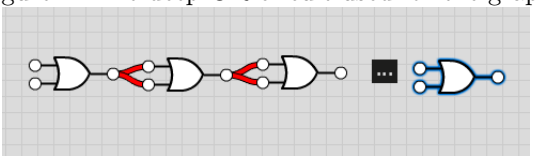
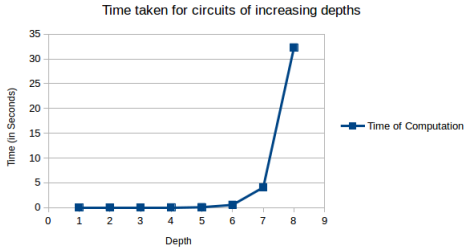


Figure 5: Depth vs time



5 How to run the code

- Compile the bob.cpp by running g++ -std=c++11 -o server bob.cpp <port_no>
- Compile alice.cpp by running g++ -std=c++11 -o client alice.cpp <port_no>
- Run the server by ./server 8000 < <text file for circuit input>
- Run the client by ./client <host_ip> <port_no>

The circuit representation is such that first you give the number of layers defined by OR levels. Then number of input gates, then number of gates excluding the output and input gates and then you define those gates by <layer_no, gate_type, node1, node2> if gate_type is OR else <layer_no, gate_type, node1>. Then the number of output gates and then the description of the gates accordingly. Currently the system works for a single output gate because multiple outputs is just multiple runs of single output circuit.

The example circuits are provided in the code folder.

References

[SYY99] Tomas Sander, Adam Young, and Moti Yung. Non-interactive cryptocomputing for nc1. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 554–, Washington, DC, USA, 1999. IEEE Computer Society.