PPML Project Report

Nikhil Pappu and Raja Rakshit

May 2020

Abstract

This report provides implementation details of our semi honest 4 party MPC (4PC) protocol tolerating Q^2 adversary strucures.

Introduction

Secure Multiparty Computation (MPC) allows a set of parties to securely compute a function of their inputs such that an adversary corrupting a subset of parties does not learn anything about the inputs of the honest parties.

MPC for small sets of parties is a particularly interesting area due to its practical significance in many real world problems. Examples include statistical data analysis, email filtering, distributed credential encryption etc. It has therefore been studied extensively in the honest majortiy (where the number of corrupted parties $t < n/2$) and the dishonest majority (where $t < n$) settings.

However, the threshold setting does not fully capture all possible adversary structures. This was generalized in [HM00] by considering general adversaries characterized by an adversary structure $Z = \{Z_1, ..., Z_n\}$. It was shown that MPC with guaranteed output delivery can be achieved in the cryptographic setting if and only if $Q^2(P, Z)$ (no two adversary sets in Z cover the entire player set P) is satisfied.

Efficient protocols for small sets of parties tolerating general adversaries have not been shown previously. We consider the setting of 4 parties (4PC). In the threshold setting, honest majority implies that only a single party can be corrupted. In the dishonest majority setting, any number of parties can be corrupted but the involved protocols are slower as they require the use of public key primitives. Also, it is impossible to achieve fairness or guaranteed output delivery with dishonest majority.

However, in the 4PC setting, guaranteed output delivery can be achieved by considering Q^2 adversary structures which are stronger than the threshold structure where only a single party can be corrupted. In addition, we can perform efficient computations only using symmetric key primitives similar to the honest majority setting.

We have implemented a semi honest 4PC protocol tolerating Q^2 structures using only symmetric key primitives. Note that without loss of generality, we

consider the following Q^2 adverary structure:

$$\{\{P_0\}, \{P_1, P_2\}, \{P_2, P_3\}, \{P_3, P_1\}\}$$

We provide the details of our implementation in the following section.

Implementation Details

Tools Used

Our implementation is written in C++. We used the cyptoTools library for performing networking and cryptographic (AES) operations. We also used the Eigen library to perform matrix computations. Our protocol does not directly depend on the code of ladnir/aby3 but shares similarities with it.

Setting Up Network Channels

We first set up communication channels (network sockets) to the other parties. We made use of the cryptoTools library for this.

The following example shows a two way communication channel set up between party P_0 and P_1. The IPs are same here so the code can be tested on the PC.

```
// CommPkg is a pair of channels (network sockets) to the other parties.
CommPkg comm;
if(partyIdx == 0)
    comm.mNext = Session(ios, "127.0.0.1:1313", SessionMode::Server, "01").addChannel();
else if(partyIdx == 1)
    comm.mPrev = Session(ios, "127.0.0.1:1313", SessionMode::Client, "01").addChannel();
```

Setting Up AES keys

Once the communication channels were set up, we had to establish some common randomness to be used in the protocols. We implemented the functionality in the init function of the class Sh4Encryptor which contains code for sharing and reconstruction.

```
Sh4Encryptor enc;
enc.init(partyIdx, comm, sysRandomSeed());
```

We passed the same random seed to pairs of parties who require common randomness as part of the protocols. The random seed is used as the key for generating random values using AES.

The Share Data Type

Each party's share in our sharing scheme consists of two field elements. We therefore defined a data type for the share of a party and overrided the basic operators to compute using these shares.

```
struct si64
{
    using value_type = i64;
    std::array<value_type, 2> mData;

    si64& operator=(const si64& copy);
    si64 operator+(const si64& rhs) const;
    si64 operator-(const si64& rhs) const;
    si64 operator*(const int i) const;

    value_type& operator[](u64 i) { return mData[i]; }
    const value_type& operator[](u64 i) const { return mData[i]; }

};
```

Sharing Functions

To perform the actual sharing of values, we make use of two Sh4Encryptor member functions localInt and remoteInt. The localInt function is executed by the party that wants to share its input. All the other parties run remoteInt which involves sending some information to the party that wants to share as well as receiving the shares.

```
si64 sh;
// localInt() takes the channels and the input
// and returns a share type si64.
if(partyIdx == 0)
    sh = enc.localInt(comm, 60);

// The other parties execute remoteInt.
// remoteInt() takes the channels and the sharing party's index.
else
    sh = enc.remoteInt(comm, 0);
```

The implementation of the localInt and remoteInt functions includes sending and receiving values using the communication channels.

```
comm.mPrev.asyncSendCopy(ret[0]);
comm.mPrev.recv(ret[1]);
```

This function essentially returns the random ciphertext obtained from AES using the common random seed. The AES function is called everytime the previously generated randomness gets used up over the course of multiple sharings.

```
if (mShareIdx + sizeof(i64) > mShareBuff[0].size() * sizeof(block)){
    mShareGen[0].ecbEncCounterMode(mShareGenIdx, mShareBuff[0].size(),
    mShareBuff[0].data());
    mShareGenIdx += mShareBuff[0].size();
    mShareIdx = 0;
}
mShareIdx += sizeof(i64);
```

Multiplication and Reconstruction

The multiplication and reconstruction functions are implemented in a similar fashion. The multiplication function takes the communication channels, the shares of a and the shares of b and returns the shares of c. The reveal function takes the communication channels and the shares as parameters and returns the reconstructed integer value.

```
si64 mulSh;
//eval is an object of the Sh4Evaluator
//class which deals with multiplication.
mulSh = eval.asyncMul(comm, shA, shB);

i64 recVal;
recVal = enc.reveal(comm, mulSh);
```

Matrix Computations

For implementing ML algorithms, we need to deal with matrices. The above protocols for sharing, rec and mul were modified using a sharing matrix data type. The matrix computations are done using the Eigen library so that they are optimized. Computing a dot product requires only one multiplication operation as the parties compute shares after multiplying and adding up all the individual values.

Asynchronous Programming

Asynchronous programming increases the throughput by utilizing multithreading. Consider this problem. If multiple sharing functions are called one after the other and if each of these functions has to wait to receive a value from an other party, then these functions will essentially block the computation and will execute only after the previous one finishes execution even though the computations are independent. We can use asynchronous programming for this.

```
// The receive part will now be run in a different thread.
auto fu = comm.mPrev.asyncRecv(dest[1]);

//At a later point, we can call
fu.get();
// This waits for the child thread to complete and obtains its returned value.
```

The advantage here is that if we use asyncRecv and call the .get() function for all of the sharing functions at a later point, then the rest of the computations in the functions can run faster as the bottleneck is removed.

Conclusion and Future Work

Finally, we tested the working of our code for small manually coded circuits. We used separate threads for all 4 parties as we needed all of them to run in parallel. Note that this is not a secure implementation and is only meant for demonstrating the working or for benchmarking of protocols. The code can be found at: https://github.com/NikhilPappu/ppmlCode.

Future work includes implementing floating point operations and a truncation protocol which are immediate extensions and allow us to perform linear regression. Further extensions would involve making the involved protocols maliciously secure and boolean sharings and arithmetic boolean switching to implement logistic regression.