591 B2 Secure Multiparty Computation

Mobile Application Interface

Linshan Jiang, Wenjun Shen, Tianqi Xu

I. Introduction

Secret Multiparty Computation is an incredibly useful tool in computer science. It computes the aggregated data from multiple parties without revealing private data of each individual party. Even Though MPC has been studied for many years with great development, the accessibility of MPC to the majority of people and the easiness to set up the current MPC frameworks are significantly underdeveloped. To broadcast MPC to a more popular level and benefit people by MPC related softwares, we developed a convenient mobile-app-based MPC API for smartphone users all over the world. Our application API has 3 notable improvements comparing to all the other MPC frameworks:

- A ready-made API. Other current MPC frameworks all exist in some sort of "shapeless" language or system to be accessed by normal user. They may give up trying to use these MPC models simply because the cost of installation is too much. Our API solved this problem just well - a programmer can integrate this API into his/her program without having to set up.
- Less malicious case. Our app is for normal group of users who are friends or colleagues, which means that normally the users among a single group know each other well. It is

often not necessary to support malicious user among a group of friends. Thus our API assume semi-honest party.

- Little maintenance cost for central server. Our MPC models do computation in each individual smartphone which reduces lots of requirements for the central server. The central server, is actually a router.

II. Implementation

Secret Multiparty Addition

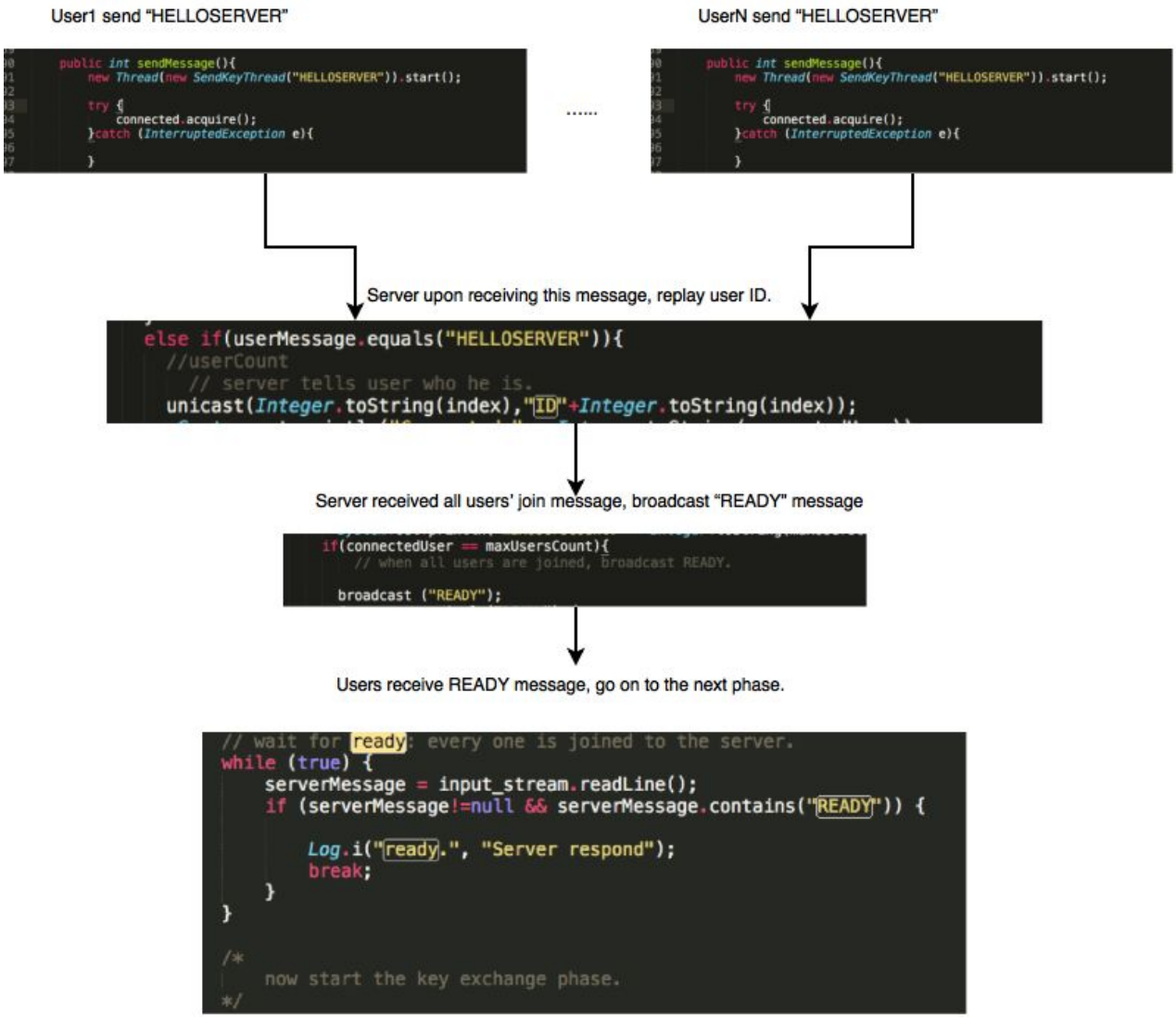
We have implemented a secret multiparty addition API. It supports secret addition for any n parties (Although this is true, obviously, for n less than 3, the protocol is not secure. Because each party can simply subtract his/her secret from the sum obtained and acquire the secret of another party). Below are the complete phases that our API processes for outputting the result of a addition operation.

1. **Encryption Phase**

Each user generate a public, private RSA key pair. The key size is setted to 1024 bits. Input for P_i is $x_i \in \mathbb{Z}$p, where p is a fixed prime agreed upon in advance. Each P_i split his secret into n shares and each share is chosen uniformly at random in $\mathbb{Z}$p. Each P_i encrypts all shares with public key that corresponding to other users.

2. Pre-Transfer Phase

P_i connect to server by sending server "HELLOSERVER" message and server echoes back the player with id i. The server wait until specified number of user connect to the server, and then send confirmation message "READY" to all users indicating that all users are online and ready to transfer messages. Upon receiving this message, all users enter next phase.



Flow Diagram for Pre-Transfer Phase

3. Key Exchange Phase

P_i broadcasts public key with header "PUB" to all users through the server. P_i receives other user's public key, and store them in an array.

```
/*
    now start the key exchange phase.
*/

// first send our public key to let server broadcast it.
out.println("PUB" + Integer.toString(myID) + " " + publicKey);
// if we have the ID, we can now reveive other users' public key.
while (numPubKeyReceived < numParties) {
    serverMessage = input_stream.readLine();
    if (serverMessage != null && serverMessage.contains("PUB")) {
        //returnText.setText(serverMessage);
        Log.d(tag, serverMessage);

        //parse public key
        int indexofspace = serverMessage.indexOf(" ");
        int parti = Integer.parseInt(serverMessage.substring(3, indexofspace));
        String pubKey = serverMessage.substring(indexofspace + 1);
        partyPublicKeys[parti] = pubKey;
        numPubKeyReceived++;
        Log.i("public key", "public key of party " + Integer.toString(parti) + " is " + pubKey);


    }
}
```

← Sending my public key.

← Receiving others' public key.

Image for Key Exchange Phase

4. Transfer Phase

Each P sends encrypted ith share to P_i using P_i 's public key. (Note that P will send the share to himself. In this case, we store the share directly instead of sending it.)

```
Addition a = new Addition (secret,10000457,numParties);
int[] sharedSecret = a.shares();

// share ith share to party i.
for(int i = 0; i < numParties; i++) {
    String ithshare = Integer.toString(sharedSecret[i]);
    Log.i("sent share",ithshare);
    // encrypt this share using party i's public key.
    String encryptedMes = RSA.encryptWithKey(partyPublicKeys[i], ithshare);
    // better for rounting - styip out new lines. Thus the key would not be seperated into many m
    encryptedMes = Crypto.stripPrivateKeyHeaders(encryptedMes);
    //text3.setText(encryptedMes);

    new Thread(new SendKeyThread("@" + Integer.toString(i)+ " " + encryptedMes)).start();
}
```

Image for Transfer Phase

5. Decryption Phase

P_i decrypts each incoming shares with his/her private key. Each P_i adds corresponding shares of all shares of secrets and obtains the combined sum.

```
publicKeyReceived.release();

while (numShareReceived < numParties) {
    serverMessage = input_stream.readLine();
    if (serverMessage != null && serverMessage.contains("FROM")) {
        int indexofspace = serverMessage.indexOf(" ");
        int parti = Integer.parseInt(serverMessage.substring(4, indexofspace));
        String share = serverMessage.substring(indexofspace + 1);
        try {
            String decrypted = RSA.decryptFromBase64(Crypto.getRSAPrivateKeyFromString(strippedKey), share);
            shareReveived[parti] = decrypted;

            Log.i("share", "FROM" + Integer.toString(parti) + " is " + decrypted);
            numShareReceived++;
        } catch (Exception e) {
            throw new RuntimeException(e);
        }


    }
}
```

Image for Decryption Phase

6. Broadcast Summed Shares phase

Last phase is broadcasting the summed shares obtained in last phase to all parties (encrypted using their public keys). Each party receiving all the summed shares, decrypt them and add all up to obtain the final output.

```
shareReceivedFlag.release();


while (combinedShareReceived < numParties) {
    serverMessage = input_stream.readLine();
    if (serverMessage != null && serverMessage.contains("FROM")) {
        int indexofspace = serverMessage.indexOf(" ");
        int parti = Integer.parseInt(serverMessage.substring(4, indexofspace));
        String share = serverMessage.substring(indexofspace + 1);
        try {
            String decrypted = RSA.decryptFromBase64(Crypto.getRSAPrivateKeyFromString(strippedKey), share);
            combineReceived[parti] = decrypted;

            Log.i("Combine Share", "FROM" + Integer.toString(parti) + " is " + decrypted);
            combinedShareReceived++;
        } catch (Exception e) {
            throw new RuntimeException(e);
        }

    }
}
```

7. Server Implementation

The server and users communicate by using socket. While user connects to the server, it initializes a user thread that listens to user. The functionalities of server include:

- broadcast messages to all users. This function is used in the key-exchange phase, and summed shares phase.
- Send message to specific user. It is used in transfer phase where users send ith share to ith users.

For test purpose, we deploy the server on csa2 with port number 8000. Number of parties are specified after the port number. Below is the command line to start server:

lindsay$ java Server_MPC 8000 2

III. APIs

1. APIs Referenced From an Open-Source Github Repository:

- **RSA**

This class imports java security API to perform RSA key generation, encryption, and decryption. Generate() function returns a pair of public key and private key with key size 1024 bits. encryptWithKey(String key, String text) takes a public key and plain text as parameters, and return the ciphertext. decryptFromBase64(Key key, String cyphredText) takes in private key and ciphertext as parameter, and return plain text.

- **Crypto**

This class implements helper functions that convert key to string, and convert string back to key. This is for easier routing purpose. Since the RSA key generated involves new line

characters and headers, it's better for us to eliminate these characters and offer a pure one-line key that would cause less trouble when routing - our server listens and fetches user inputs line by line. Thus if the keys are split as multiple lines, our server may shuffle the sequence of key pieces.

2. **APIs we implemented:**
    - **Addition:**

This class takes in a secret integer value, prime, number of participants as parameters. It includes two helper functions: shares() and combine(String[ ] shares). shares() returns an arraylist that stores the splitted shares into indexes that corresponding to user's ids. combine(String[ ] shares) takes an arraylist of all shares that collects from all users, and return the final output. Example of use:

```
Addition a = new Addition (secret,10000457,numParties);
int[] sharedSecret = a.shares();
```

- **MPCSecretAddition:**

Using the APIs above, we further build secret addition API. This class takes server port number, server IP address, party number, and secret message as parameters. Below is an example of initializing a MPCSecretAddition instance.

```
MPCSecretAddition mpc = new MPCSecretAddition(SERVERPORT, SERVER_IP, numParties, Integer.parseInt(input) );
int result = mpc.sendMessage();
```

Once a new instance is created, call sendMessage() function to send message, and all the works from encryption phase to broadcast summed shares phase will be done on the background.

IV. Application

- Secret addition

Secret addition can be applied to calculate summation, average, which are the most common operations people try to do with their personal data. For example, salary is one of the personal information that people do not want to reveal to public. This secret addition can simply calculate the sum of all salaries and divided by number of participants. Vote is an alternative function that can be accomplished by addition. It's an addition of 0s and 1s. Vote can be applied to decision making in various occasions from deciding to watch a movie or not to executives decide whether to acquire a company. It can be done on a simple mobile application.

- About Vector addition

Suppose a group of people want to find a coffee shop to meet as close as possible but they do not want to expose their locations. Assume each group member computes a distance vector, which stores the distance from his location to local coffee shop. And all the vectors are canonical. We can perform a vector addition to sum up all the vectors and find the minimum value all under MPC. Using secret addition we already built, we can compute summed vectors. However, since we revealed this sum to all parties, it may leak some user privacy. We try to solve this problem by a different approach. Instead of finding a closest coffee shop, we find the best "preferred" shop to all parties. For example, there is a distance vector to shop A,B,C: [10,5,7]. We convert this distance vector to preference vector[3,1,2]. Notice that the closest one is the most preferable.

By converting to preference vector, we can hide the real distance vector. After that, we perform a vector addition on the preference vectors. Each party knows the sum of preference vectors, and compute minimum value using non-MPC method. In this way, we may or may not find the true minimum distance, but is guaranteed to find the most “preferred” place. Moreover, this alternative approach should still be user for party number greater than 2, just like secret addition. Since with two party, one can subtract his/her preference vector from the summed preference vector and get the other’s preference vector. Knowing this piece of information could still reveal some user privacy such as relatively how far he is from each shop. However, the privacy leak will decrease with increasing number of parties.

V. Future Work

To date, our API fully supports secret multiparty addition. The vector addition API is still unstable with limitation of time. But we will definitely keep working on it. In the future, we are going to extend the API to support more operations, such as comparison and minimum. We hope to implement our “finding closed coffee shop” problem by truly finding the minimum of summed vectors. We hope our project lights a way for popularizing secure multiparty computation by lowering the cost for developing, and thus bridges the gap between theory and implementation. Hopefully in this way, we attract more intelligent computer scientists to dedicate to this area and allow ordinary citizens to realize the benefits of MPC.

Citation

https://web.archive.org/web/20130622140256/http://www.daimi.au.dk/~ivan/MPCbook.pdf

http://blog.brainattica.com/working-with-rsa-in-android-3/