Semester project – Applied Cryptography

Private set intersection

(Ákos Boros, Zsolt Fodor)

**1. Abstract**

Private set intersection is a secure multiparty computation cryptographic technique that allows two parties holding sets to compare encrypted versions of these sets in order to compute the intersection. In such a scheme, the output can vary, it can be a set of intersecting elements, one particular common element or even just a bit, that tells if one particular element is included in both sets or not. These are just some examples, basically any type of output is imaginable that satisfies the goal of the given protocol. The main purpose of such a scheme, that neither party should reveal anything to the counterparty except for the information that can be deduced from the output. Our goal was to implement such a protocol, that uses the elliptic curve Diffie-Hellman problem as its underlying hard task to ensure the privacy of the parties.

**2. Introduction**

Our plan for this project was to implement an application with a simple and straightforward graphical interface, that uses sockets for communication, which enables two persons to find a set of time slots in their timetable, when both of them are free, so a meeting with a predefined length could be arranged between them. However, this information exchange should be carried out in a way that ensures the privacy of the parties, which means no other information should be revealed to one of the parties about the schedule of the other person, just those time slots, when both of them are free, and the time intervals, when given party is free, and the other one is not, which can be logically deduced from the output, and from his/her own timetable. To achieve this goal, we used the elliptic curve Diffie-Hellman protocol with some modification to make it suitable for our purposes.

**3. Implementation**

**3. 1. Libraries**

We chose Python as the programming language of our implementation, because it seemed the most suitable for our purposes. Python comes with a lot of packages that gives high-level functions for cryptography purposes, and also a lot of modules are available, with which simple graphical interfaces can be built. These packages made our work much easier during the implementation, and since the speed and careful memory management were not something that we wanted to prioritize, Python seemed the superior choice over C or C++. In the following paragraphs I introduce the most important packages that we used and give a short description about them, and also mention at what part of our implementation we made use of them.

We used the tinyec library, which is a little library to perform arithmetic operations on elliptic curves in pure python. In this module, there are two main classes, one for the curves and one for the points. We can choose what elliptic curve we would like to use, there are some predefined ones that comes with the library, and also arbitrary curves can be defined, however it seemed wiser to choose one that has been used by real protocols in practice, so in our implementation we used one of those ones, which is called *secp192r1.*

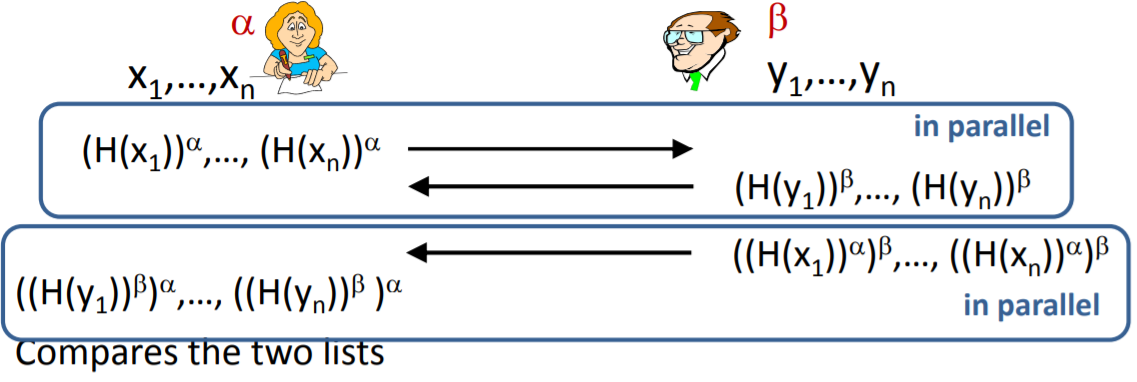
The one package that we used to build up our user interfaces is PyQt5. PyQt5 is a comprehensive set of Python bindings for Qt v5. Qt is set of cross-platform C++ libraries that implement high-level APIs for accessing many aspects of modern desktop and mobile systems. It can be used to create many kind of graphical applications, and the use of buttons and text fields is well supported in this library so it seemed optimal for our purposes.

For the communication between the parties we used the socket package, which is part of the standard library of Python. It provides functions for listening on and sending data to specified addresses and ports, so client and server processes can both be implemented. It supports TCP and UDP as well, but for our goal TCP seemed the better choice, so we used that in our implementation.

**3. 2. The protocol**

As I previously mentioned the whole protocol is based on the Diffie-Hellman key agreement protocol. The basic idea is that we create a new private key from every free time slot of both of the parties then check if there are any private key that has been created twice, that would mean that the time slot it is created from is free for both of the parties. To make it more specific let’s go through the whole protocol step-by-step. Our two parties are Alice and Bob. We denote Alice’s available time slots with x1, x2, …, xn, and Bob’s free time slots with y1, y2, …, ym.

1. First Alice and Bob agree on a hash function H, and computes H(x1), H(x2), … H(xn), and H(y1), H(y2), … H(ym) respectively.
2. Alice and Bob picks one-one private input a, and b, then compute H(x1)a, H(x2)a, … H(xn)a, and H(y1)b, H(y2)b, … H(ym)b respectively, and both of them sends their values to the other party.
3. Both of them computes H(y1)ba, H(y2)ba, … H(ym)ba and H(x1)ab, H(x2)ab, … H(xn)ab respectively and Bob sends his values to Alice.
4. Alice checks the values she got from Bob, and if there is any of them is the same as one of the values she computed in the previous step, then Alice checks from which time slot this value was computed, and saves this time slot. After Alice has found all the matching time slots she proposes one of them to Bob.



1. figure The basic protocol

However, to achieve advanced security we chose to implement the protocol using the elliptic curve Diffie-Hellman problem instead. This modification made the protocol slightly more complicated, but the main idea behind the schemes stayed the same. With using the previous notations, except for Bob time slots where we use z1, z2, .., zm this time to avoid confusion, with the y coordinate of points, the modified protocol follows these steps:

1. First Alice and Bob agree on a hash function H and an elliptic curve E, then they compute H(x1), H(x2), … H(xn), and H(z1), H(z2), … H(zm) respectively.
2. For every i = 1 .. n Alice computes yai, such that the point PAi := (H(xi), yai) is on the curve E. Bob does the same for every of his time slots, so for every j = 1 .. m Bob computes ybi, such that the point PBi := (H(zi), ybi) is on the curve E.
3. Alice and Bob picks a private input a, and b, computes a \* PA1, a \* PA2, …, a \* PAn and b \* PB1, b \* PB2, …, b \* PBm respectively, and send these points to the other party.
4. Both of them computes a \* b \* PB1, a \* b \* PB2, …, a \* b \* PBm and b \* a \* PA1, b \* a \* PA2, …, b \* a \* PAn respectively and Bob sends his points to Alice.
5. Alice checks the points she got from Bob, and if there is any of them is the same as one of the points she computed in the previous step, then Alice checks from which time slot this point was computed, and saves this time slot. After Alice has found all the matching time slots she proposes one of them to Bob.

To increase the security of the protocol even more, we also add some fake slots to the ones that the first party chooses, and the points corresponding to these slots will be sent as well to the other party. These fake values will be filtered out in the end of course, before the first party makes the decision about the proposed time slot for the meeting. This solution makes the privacy of the two parties a bit unbalanced, because the one, that initiates the protocol has higher security guarantees, however it would be problematic to provide the same for the other person, given the decision in the end is always made by the first party.

**3. 3. Key parts in the implementation**

The greatest issue that we encountered in the implementation process of the steps of the protocol was to find an y coordinate ta given x value, such that a point with coordinates (x, y) is on the curve. The mathematically hard part of this task is that the modulo square root of a given value has to be computed in order to find a good corresponding y coordinate for a value that is used as the x coordinate. This is however, when a large prime is used in the generation of the elliptic curve can be a hard task. For the first time, we tried the naïve algorithm for this problem, so we checked for every value in the given field, if it is a square root of a given number or not. This method is really slow, and took way too much time, so we needed to find a better solution so the protocol could finish in reasonable time. For the second try, we implemented the Shanks-Tonelli algorithm, that provides a faster way to compute the modulo square root of a given number. This method, however was still not fast enough, if the prime that is used in the generation of the elliptic curve was big, however if we wanted to work on a publicly accepted secure curve, then we should have not decreased the value of this prime. The final solution was to use the thesis that says if for a prime p it is true that p % 4 = 3, then for any number n if it has a square root modulo p, then it should be n((p + 1 )/ 4) or - n((p + 1 )/ 4). With this method, if we choose an appropriate curve, then we can get the square root really fast, so it seemed ideal for our protocol. The prime that generates the curve *secp192r1* satisfies the previous property, mostly that was the reason we chose this elliptic curve for our protocol.