Puppy Love: Cryptographically secure anonymous couple matching

Saksham Sharma

Computer Science and Technology
Indian Institute of Technology Kanpur
sakshams@cse.iitk.ac.in

Abstract

Anonymous and secure dating algorithms are not only hard to define, but apparently tough to realize as well. Tinder et all have succeeded in making this a mainstream application, and yet, there is scope left in improving the anonymity of such platforms to provide a guarantee to the users that their choices shall not be made public, nor known to the administrators.

We designed an algorithm for this problem which provides strong guarantees about privacy of users, and implemented it inside our campus, catering to almost 2000 users. The algorithm is designed to be easily scalable, and was a success in the university setting.

1. Introduction

The queerly named platform has been running in IIT Kanpur since 2014, meant to help shy nerds meet their crush. The platform opens one week prior to the Valentine's Day every year, and lets people choose up to 4 of their crushes. At the stroke of the midnight hour on 14th February, people who happen to like each other are informed about the same. If your *love* was unrequited (the other person didn't like you), you will not get to know. More importantly, if you did not like the other person, you would not know if that person liked you or not. In addition, people can see the count of people who have selected their names without finding out who they are.

Since a campus is often a close knit circle, users are apprehensive about revealing their crushes to others, including the administrators (who are students themselves), lest it leak out. That makes it necessary to design a technique to ensure that users are guaranteed that obtaining their choices is either impossible or computationally infeasible for anyone but them.

We developed a new version of the platform which provides strong privacy guarantees without compromising on performance or features.

2. Requirements from the algorithm

The following discussion refers to $Alice$ and Bob. All arguments apply to each of them.
The security goals are the following:

1. $Alice$ should not find out the choice of person Bob if $Alice$ did not like Bob.
2. It should not be possible for $Alice$ to cheat and obtain whether $Alice$ matched with the Bob, without this information being made available for Bob. In other words, the algorithm should be fair, and ideally symmetric.
3. The administrator should not be able to obtain the choices of a person, or brute force the data in any way to deduce the likes of a person.
4. The administrator should not find out which people matched, but knowing the count of matches is reasonable.
5. The user should be able to verify that the frontend is not sending any such information which could reveal their choices. Thus, the privacy guarantees should hold whatever code is running on the backend.
6. The counting technique should function independently without revealing the identities of the users.
7. Security features should be practical to implement in a web browser and should be completely transparent to the user.
8. It can be assumed that the server does not collude with another involved party. Thus, the server is assumed to be honest-but-curious and will not disrupt the service intentionally.

Some other practical requirements:

1. The algorithm should have no false positives or false negatives.
2. It should ideally involve only $O(1)$ computation by each involved party.
3. There should ideally be no to-and-fro communication between the involved parties.

3. Related work

A similar problem was proposed and tackled in [1] where the proposed various algorithms, including one which does present a secure protocol. The drawback of such an algorithm is that it requires all involved parties to take part in $O(n)$ computation, where n is the number of other parties.

Our algorithm is completely different from this work, but it is interesting to look at the mentioned paper nonetheless.

4. Initial approaches

We went through a lot of approaches for this problem, before finally arriving at the presented approach, which, although extremely simple, is not obvious at first.

We initially modelled the problem as computing the AND of $O(n^2)$ bits, disregarding the requirement of $O(1)$ computations per person.

4.1 Homomorphic computation

Our initial attempts at homomorphic were foiled by the fact that completely homomorphic systems are impractical.

In contrast, partially homomorphic functions, although practical, do not provide any function which can be used for this purpose. The available computations are XOR, multiply, addition over 1 bit et all. Composing any two of these functions provides enough information to both parties to violate privacy concerns.

4.2 Yao's two party computation

Yao presented a secure protocol to compute any arbitrary function (which can be run using logic gates) across two parties in a shared manner. The protocol was not originally intended to be fair though, and one party could cheat in some respect.

We describe the protocol here:

- Alice party creates a truth table for an AND gate, with columns A (her choice), B (Bob's choice) and C (result).
- She selects 6 random values $a_0, a_1, b_0, b_1, c_0, c_1$. She then shuffles the table, and encrypts $((A = i)\&(B = j)) = Ck$ with the value $a_i - b_j$. She then shares this table with Bob.
- If Alice's boolean choice is $choice$, she sends a_{choice} to Bob as well.
- Bob obtains $b_{bob'schoice}$ from Alice using Oblivious transfer, such that Alice will not know which bit value was obtained by Bob.
- Bob tries decrypting the table with the key $a_{alice} - b_{bob}$ and finds out what value he obtained. He can then look up whether they matched or not.

We attempted to make the protocol fair by using the central server, who would then match both parties and declare their result to them.

A clever hack around this was found on noticing the asymmetry in the algorithm. Alice could forge the truth table to return a $mismatch$ if Bob likes Alice, and a $match$ otherwise. A false positive can be borne easily, but if Bob did like Alice, he would not know that Alice has cheated and has found out about his liking. This is certainly unacceptable.

4.3 Public key encrypted messages

This involves the following step:

- All parties get to send 4 values to the server.
- If Alice likes Bob, she will send a message (string containing their roll numbers) encrypted with Bob's public key to the server.
- Bob will encrypt the same message with his own public key and store it on the server.
- The person to use the other's public key is chosen using lexicographic comparison.
- The server will detect duplicates and declare matches.

This is a very simple algorithm, but fails on a computationally feasible brute force attack. The server can easily try out all possible messages which Alice could have sent, and stop when it produces a value which is the same as the one sent by Alice.

4.4 An O(n) algorithm using duplicates

We observed that instead of trying to compute an AND, it may be interesting to instead use the duplicate detection in another way.

An algorithm for this is quite simple:

- Both parties establish an encrypted channel using their public keys.
- They both exchange a random token over this channel. Let the tokens be A, B.
- Both of them declare a value $hash(A + B)$ to the server, who verifies that the values are the same.
- Each party gets to send 4 new values to the server.
- If Alice likes Bob, she will send the pre-decided A to the server. Otherwise she will send a random value.
- Server computes $hash(A_{received} - B_{received})$ and matches it with the values received earlier. If they are the same, it declares a match.

Although this algorithm has no obvious security concerns, it involves a lot of computation, infact $O(n)$ computation per person. Our tests showed a huge performance bottleneck on modern browsers while using Stanford's $sjcl$ library for crypto-protocol implementations.

The platform originally launched with this algorithm, but the performance issues on scaling to $O(1000)$ people made it practically impossible to run.

5. The Puppy Love algorithm

The final algorithm was inspired by the previous $O(n)$ algorithm, wherein it was identified that it is essential to come up with a message which only the involved parties can create, and no one else. This is the only step which was causing a $O(n)$ bottleneck.

Apart from that, some other requirements (counting the number of people who like you, stopping the server from knowing the matched people) required some additions to this algorithm which are described later.

5.1 Matching

- There is a known generator field g.
- Let the public key of Alice be g^a and her private key be a. The same is true for Bob's g^b and b.
- The value g^{ab} cannot be computed by anyone other than Alice and Bob, since it is the Discrete Logarithm problem, which is known to not lie in PTime.
- Both parties get to send 4 values to the server.
- If Alice likes Bob, she will send g^{ab} to the server. The same applies to Bob.
- The server declares a match if it detects a duplicate value.

Some remarks:

1. The algorithm itself is quite simple, and does not present any apparent privacy concerns. The only issue (to be resolved later) is that the server will know which two people have matched.
2. It is extremely fast to implement, and running times are ideal for browsers, since it involves only 4 maximum mathematical computations.
3. Values received by the server appear completely random to it, and cannot be brute forced.
4. The value of g^{ab} is very easily obtained if the Public-Private keypairs are ElGamal, as was in our case. This value is simply the Diffie Hellman value computed in Elliptic Curve Cryptography, and Stanford's $sjcl$ library provides a direct function for this computation.

5.2 Implementing the counting of people who like you

We opted for a very simple solution to this, albeit it has a minor scope for inconsistencies.

When Alice sends her preferences, her client also encrypts a random string with the public key of Bob (if Bob is among her

likes). This value is stored on the server in a list ordered by timestamps. Other clients retrieve all new entries in this list everytime they log back in, and try decrypting all received values. The client can detect that some of the messages were encrypted with the user's public key, but cannot detect the sender of the message.

5.3 Ensuring that the server does not know the matches

This was not implemented during 2017's deployment of the platform, but should work fine regardless.

- Before sending the final chosen values to the server, the client obtains the blind signatures of the values it plans to send.
- The blind signatures, as well as the actual values which are being sent will not be known to the server.
- The client now switches IP (in our case, sends the packet through the NAT of the university, re-entering the university network using a reverse proxy, thus completely anonymizing the identity of the sender), and sends its 4 tokens (with signature) to the server without attaching its login cookie with them. The server can only verify that the received values are correctly signed.
- The server declares all the matched tokens in the end, and clients can check if their token is among this list of matched tokens.

5.4 Miscellaneous security concerns

There are some other concerns which are discussed here:

- IP switching is unreliable. The user may not be expected to undertake this arduous task of accessing the platform behind a VPN/Tor for the step of sending the tokens. Thankfully, our university setup had a network which is known to be not in the control of the students, and thus the packets may use the NAT to ensure their identities are safe.
- It is possible for the server to display fake public keys for users. This surely is a concern, but can be easily verified. The server does not ask for cookies while serving public keys, and thus, it can be easily verified (by a concerned user) whether a different public key is being server to some people. If the university infrastructure allowed for reliable hosting of public keys, this task could be eased.
- The server may serve fake code for some users which reveals more information. But in such a case, the hashes of the code received and the code expected will not match, and a concerned user can verify the same, or run a local copy of the code (which she has manually verified to be safe).

6. Deployment

The platform was well received, with more than 1800 students registering on the website, and as many as 45 matched couples. There were many people who had multiple matches, both among boys and girls.

Although girls are outnumbered on the campus $1:10$, among the registrants, the ratio was close to $1:5$.

The backend was implemented in Golang, using the $iris$ web framework. The frontend was written using Angular2 and TypeScript, communicating with the backend using REST API of the backend. The server resource usage was monitored and was observed to be very low even during peak times, due to Golang's lightweight web frameworks. MongoDB was used for data storage, and Redis was used to ensure persistent sessions.

References

[1] Senpai: Solving the dating problem SIGTBD 2017.