Mental Mafia

Team A14

Neiro Cabrera
neirocabrera7@ucla.edu

Yamm Elnekave
yamme@ucla.edu

Andres Enriquez
andresenriquez@ucla.edu

Brandon Lo
brandonlo11@ucla.edu

Kenneth Tang
kennethtang@ucla.edu

December 8, 2024

Abstract

Mafia is a game where players are assigned an anonymous role (e.g. mafia, angel, detective, townsperson). Each round of the game, each player can perform an action corresponding to their role (e.g. a mafia can choose to kill a player, a angel can choose to prevent a player from dying to the mafia that round, a detective can probe any player and learn their role). Following this phase, everyone votes out a player. The game continues until either only the mafia remain in which the mafia wins or if no mafia remain in which the townspeople win. Typically this game requires a moderator which serves as a trusted third party (TTP) that every player can tell their decision to secretly and then announces the decisions on their behalf.

This project introduces the concept of mental mafia which seeks to answer the question of if we can still play mafia (or any mafia-like game) without a TTP. This paper introduces tools to accomplish this objective largely using primitives from secure multi-party computation (MPC) and proposes protocols for the game in two settings: honest-but-curious and malicious.

In the honest-but-curious setting, every player follows the protocol to play the game. That is every player plays the game faithful to the instructions, but can still try to figure out information that should be kept hidden such as player roles and role decisions. In the malicious setting, players can deviate from the protocol and try to sabotage the game. They can purposely not follow instructions and supply inputs to the protocol that can misguide the game and reveal information.

1 Background

Now we proceed to formalize the game of mafia in order to outline the underlying problems that need to be solved. In a game of mafia with n players, there is a public identity space P with $|P| = n$, secret/private identity space space S with $|S| = n$, and a role space R such that $|R| \leq n$.

The role assignment function $r : S \to R$ is a surjective map that takes a player's private identity $s \in S$ and outputs a corresponding role $r \in R$. This function satisfies the following properties:

- Fixed randomess: Role assignment is random, however, in a single instance of the game, a player's role does not change and computing r multiple times yields the same result, that is

$$r(s) = r(s), \forall\ s \in S$$

- Role agreement: In a single instance of the game, players agree upon the role space, for example

$$R = \{\texttt{mafia}, \texttt{angel}, \texttt{detective}, \texttt{townsperson}\}$$

The role reveal function is a mapping $m : P \to S$ that reveals a given player's private identity given their public identity. It is important to note that each player can only compute their own instance of

this function (i.e. know their own secret identity). This function represents a player's ability to reveal their role after the end of the game. Also note that this function is a permutation of the public identity space to the private identity space.

In the traditional game, the TTP knows everyone's public and private identities (i.e. TTP is able to compute the function m for any player). Thus the TTP is the party responsible for propagating authenticity of a message from some private identity. With secure multi-party computation, we create a situation where even without knowing the association of any other player's private and public identities, we can still ensure authenticity of any messages from a private identity.

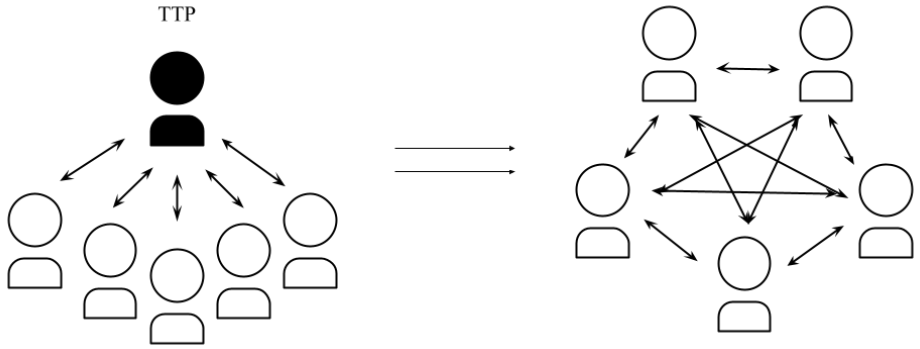

Figure 1: Using MPC to remove the need for a trusted third party

2 Primitives

There are three conceptual tasks we are trying to achieve in this game. We outline them below.

Definition 1 (Private identity generation)**.** *The notion of private identity generation allows for parties to jointly compute and distribute randomly private identities for each player that only each player themselves is aware of. Every player is assured that all private identities have been distributed and furthermore that no one has the ability to impersonate the private identity of another player.*

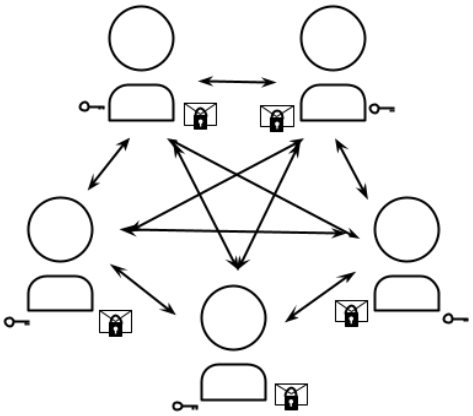

Figure 2: Private identity generation allows for everyone to receive a message (i.e. which contains their identity) only known to them.

Definition 2 (Anonymous authenticated broadcast)**.** *The notion of anonymous authenticated broadcast allows players to broadcast some message without revealing their association between their*

public and private identities, but ensuring the message is authenticated, that is it came from a certain private identity and hence a given role.

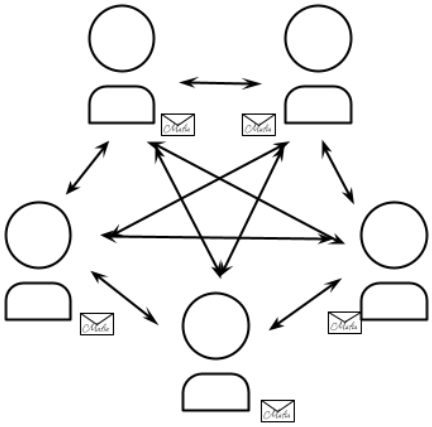

Figure 3: Anonymous authenticated broadcast sends a message to every player anonymously but signed by a role (e.g. mafia) so everyone can verify only the player with the role sent the message.

Definition 3 (Anonymous reveal)**.** *The notion of anonymous reveal allows one player to learn something else from a single other player without letting anyone else know who they chose, what they learned, or revealing their own public identity in the process.*

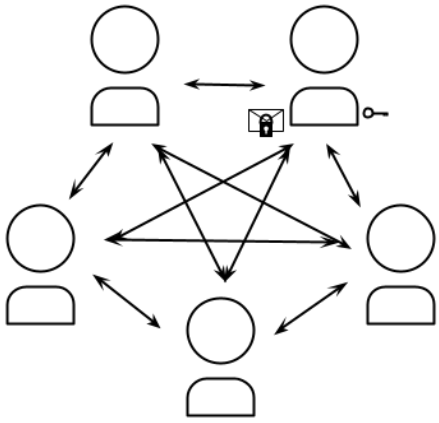

Figure 4: Anonymous reveal allows an anonymous individual to receive one message of their choosing from some player that only they can read.

3 Honest-but-Curious Setting

Definition 4 (Honest-but-curious)**.** *The honest-but-curious (or semi-honest) threat model [Lin20] encapsulates the notion that parties all behave honestly but are free to learn as much information as they can. That is, all parties follow the protocol, however while doing so, they may attempt to learn information about others.*

For our protocol in this threat model we have the following assumptions:

- Parties all follow the protocol.

- Parties do not collude with each other. That is each party may only attempt to discover information by themselves.

3.1 Private Identity Generation

In order to jointly compute and distribute roles, we adapt concepts from Mental Poker [SRA81]. The basis of this protocol is the ability to shuffle a deck of "cards" (in our case the various roles) and securely deal cards to each player without a central authority. We will use this distribution protocol for each player to be assigned one role from a deck of roles that have been agreed upon.

For example, define $P = \{1, \ldots, n\}$ and $S = \{1, \ldots, n\}$. Let us shuffle the numbers $[1, \ldots, n]$, and everyone is given one of the shuffled numbers privately. These numbers that they get are their private identities. Note this is different from the identities that they reveal publicly.

Say Bob is the first to come to the table, he will be given public identity $p = 1 \in P$. Then once the deck of private identities are distributed he is randomly given some private identity $s \xleftarrow{R} S?$, let us say $s = 4$. Then Bob can compute $r(4)$ to receive their role in the role space R.

Let us make a mapping $m : P \to S$ that shows us what secret identity a public identity has. So $m(1) = 4$. This mapping is not known to everyone, every player just knows their own entry in the map. Bob knows that as player 1, he was given the secret identity 4.

3.2 Anonymous Authenticated Broadcast

To achieve the notion of anonymous authenticated broadcast in an honest-but-curious setting, we utilize addition under secure multi-party computation. The intuition is that if every party follows the protocol and non-broadcasting parties supplies secret shares of 0 as input, the resulting computation and aggregation by all parties results in the sum of the messages from the broadcasting party. In fact, some underlying techniques from SPDZ [Dam+12] and Spectrum [NSD22] use the same principle.

Suppose player $p_i \in P$ wants to broadcast the message m. Then every player p_j such that $j \neq i$ supplies 0 as input, and the player p_i supplies their message m as their input. Then

$$\mathtt{MPC}_{add}(0, \ldots, m, \ldots, 0) = m$$

This concept adapts the idea from the dining cryptographers problem [Cha88] of sending secret shares of 0 as "cover traffic" for the message so the broadcast of the message itself is anonymized. In fact, if the parties do indeed follow the protocol under the honest-but-curious assumption, this protocol is information theoretically secure with uniform probability of guessing the party that sent the message.

Authentication comes naturally in the honest-but-curious model. In the given round of broadcast, every party that is not being publicly asked to share the message will send 0 shares. Hence, following the protocol, the message must come from the party requested (e.g. the mafia, detective, etc.).

Clearly though, if any malicious party sends a non-zero share, then the resulting message can be manipulated. Thus, further techniques need to be applied to make anonymous authenticated broadcast resilient to tampering.

3.3 Anonymous Reveal

How can some player A learn something from a different player B without letting player B know that they were chosen in an honest-but-curious setting?

This is a two-step protocol:

1. Player A needs the output of the multi-party computation to be readable only by him. Let A generate a public/private key pair and use anonymous authenticated broadcast referenced in the previous section to publish their own public key pk_A.

2. Player A then chooses a $p \in P$ player of which identity he wants to reveal, and everyone computes the following function jointly:

$$\texttt{MPC}_{reveal}([E(m(p_1), pk_A), p_1], ..., [00...00, p' \in P], ...[E(m(n), pk_A), p_n])$$
$$= \texttt{INPUT}[p'] = [E(m(p'), pk_A), p']$$

The desired player can then decrypt the message and learn the role of the player they probed by computing

$$D(sk_A, E(m(p'), pk_A))$$

thus giving player A the anonymous identity of p', $m(p')$.

Analysis (honest but curious): Every role puts in their real player id and encrypts it by the detective's public key. The detective can only get one output from this function as that is what is calculated in the MPC, and finally, the detective can see the associated role and player index.
Notice that the encryption is non-deterministic, so no player can brute-force all combinations.

4 Malicious Setting

Definition 5 (Malicious)**.** *The malicious threat model encapsulates the notion of no trust, in that anyone can be a bad actor. In this model, parties are free to deviate from the protocol, meaning they do not have to follow input constraints, required computations, communications, etc.*

For a fully malicious protocol, we need to protect against collusion. However, in our case we make the caveat that players do not collude against each other.

4.1 Private Identity Generation

> For our malicious setting to work, we require a special role distribution phase, which will give every player a new public key linked to a role, but not to them. Essentially linking a pk_{S_i} to r_i. Ie. giving them a secret identity. Additionally, they all need to come to the table with a public/private key pair for their public identity P_i

Setup phase (role distribution): suppose we are trying to assign each player a role with the requirements that no one else knows what another person's role is.

1. Each player locally generates an asymmetric key pair (sk_i, pk_i)

2. The players jointly compute the following function

$$\texttt{MPC}_{setup}(pk_1, pk_2, \ldots, pk_n) = [(pk_1, r(1)), (pk_2, r(2)), \ldots, (pk_n, r(n))]$$

We can shuffle using Fisher-Yates [FY48] and a pseudo-random number generated in MPC.

> In this setting, we show that if you are $z_i \in Z$ and you are assigned the role $r(z_i) = r(i)$, then you should be associated with PK pk_i. This does not reveal which player p_i you are in the game.

Then everyone learns all the public keys and associated roles with the public keys, although they don't know who owns the corresponding private keys (which we will use later to authenticate).

Why is this setup phase safe against malicious players? Any malicious player can deviate from the protocol and supply an input that is not their public key. Well that doesn't do them any good, because

later on they do not have a way to authenticate them to their role since any attempt to verify a message with their gibberish public key fails.

Note though that players may collude and submit the same public key for different roles, so we must ensure that public keys provided are unique (i.e. $\texttt{MPC}_{setup}$ either aborts if provided duplicate inputs or reveals the players that do so).

4.2 Anonymous Authenticated Broadcast

Now suppose a given role wants to send a message to other parties without revealing who they are but that they have some authority to send the message (i.e. they belong to a certain role)

1. The players jointly compute the following function

$$\texttt{MPC}_{verify}(0,\ldots,0,(m,S(sk_i,m)),0,\ldots,0) = [0,\ldots,0,(m,S(sk_i,m)),0,\ldots,0]$$

   where $S(sk,m)$ outputs the signature of m using secret key sk. That is only the player broadcasting the message outputs the message and a corresponding signature.

2. Every player then locally verifies the player who broadcast m has role r by searching for j such that $r(j) = r$, finding the associated public key pk_j, then performing $V(pk_j, S(sk_i,m))$ which only accepts if $i = j$ as desired.

4.3 Anonymous Reveal

Again how can a single player (a) learn something from a different player (b) without letting player b know that they were chosen?

Here we can use our public keys associated with roles to our advantage in order to achieve revealing to one player by encrypting with the desired public key.

Finally, we can ensure that only one is selected by performing an array select.

Example: Reveal roles to the detective.

$$\begin{aligned}\texttt{MPC}_{reveal}([E(m(p_1),pk_A),p_1],...,[00...00,p' \in P],...[E(m(n),pk_A),p_n]) \\ = \texttt{INPUT}[p'] = [E(m(p'),pk_A),p']\end{aligned}$$

The desired player can then decrypt the message and learn the role of the player they probed.

$$D(sk_{detective}, E(m(p'),pk_{detective})) = m(p')$$

Analysis (malicious): In order to make this safe in a malicious setting we need to prove that we are indeed the player p_j, we can do this by signing with the secret key associated with our player id $p_j \in P$ which we use to initialize the game with.

Add to p_j the signature $Sign(sk_{p_j}, p_j)$
Also, how do you ensure that they reveal the correct secret identity SID $s \in S$? In the exact same way, add to $m(p_i) = s \in S$ the signature using the associated private key from the role distribution phase. $Sign(sk_s, s)$.
Finally, how do you ensure that you get the p' from the detective, and not someone pretending to be the detective by putting all 0s? Instead of all 0s, we have to change it, in fact before the anonymous reveal we will need to perform
This can only be done by doing signature verification of 0s in the MPC. This is extremely inefficient and it is infeasible to implement.
This is why we decided against implementing a malicious protocol.

5 Applying primitives to roles

5.1 Detective

The detective should always go first, as he should not know anything about anyone killed before he does his probe.
The detective uses Anonymous Reveal to probe a player and learn their role. The detective can then use this information to try to figure out who the mafia is. Note this does not necessarily mean that people will believe them

5.2 Mafia

The mafia can choose to kill a player. We will use Anonymous Authenticated Broadcast to broadcast the player they want to kill.
Notice that the mafia kills player $p \in P$ and not any secret identity $s \in S$.
At each round, everyone should know who the kill was attempted on, but not who the mafia is.

5.3 Angel

Each round, we shouldn't know who the angel intended to save unless that person was killed by the mafia (and the angel revived them). ie. we only get a true/false on whether the angel saved someone
First, the angel must submit a commitment to a player $p \in P$ they intend to save that night. Let $c = \text{Commit}(p)$. We will accomplish this with anonymous authenticated broadcast.
Between the commitment and the calculation below, the mafia will submit its kill.
An honest-but-curious setting allows for:

$$\texttt{MPC}_{angel}(0, \dots p, \dots, 0) = (\text{mafia's kill} == p)$$

Then if it's true the angel must prove that it was its original commitment by doing:

$$AAB(\text{Open}(c))$$

Here we use a commitment, very simply it consists of two functions, commit, and open. Such that after committing no one knows what you committed to, but you can only 'open' to what you committed.
This ensures that no angel can decide on who to save after seeing the mafia's kill.

5.4 Townsperson

The townsperson has no special abilities.

6 Daytime voting

All of the above report was for how to handle the night phase. The daytime phase is much simpler.
Everybody sheds their secret roles, and simply vote on who they think the mafia is using their public identities. If there is a tie, nobody is killed. Such an MPC function already exists, and we do not need to use any of our primitives.

7 Game over logic

7.1 Honest-but-curious

Mafia win: This can only occur when one person is left, as if there is a mafia when there are 2 people alive no one can be voted out, and the mafia will kill the other person. So when there is one person left, the mafia wins.

Townsfolk win: Whenever the mafia is voted out they will elect to tell everyone that they were voted. This doesn't need to be done in MPC.

7.2 Malicious setting

Mafia wins
If only one player remains, they can reveal their role's secret key, signed by their player's private key.

$$S(sk_{p_i}, m = sk_{s_i})$$

Townspersons win
If the mafia is voted out, he doesn't need to reveal that he was voted out, but if he get's to the end of the game and attempts the Mafia wins protocol, he will be caught as this p_i is known to have been voted out and hence a townsperson victory is declared.

Thus there is no reason for the mafia won't disclose himself upon being voted out. By performing the same thing:

$$S(sk_{p_i}, m = sk_{z_i})$$

8 Implementation

When implementing MPC protocols there are a couple of problems with having an element secret shared. For instance, in the anonymous reveal, we need to get the index of the player the detective requested to reveal.
Say we were able to get a requested id from the detective. How do we find the index from the input to reveal?

```
index_target = sint(-1)
for i in range(N):
    index_target = index_target * (client_inputs_id[i] != requested_id)
                    + i * (client_inputs_id[i] == requested_id)
```

We can't know the exact index where `client_inputs_id[i]` will be the same as the requested id. As we don't know either due to the secret sharing nature. Thus we must use math, to create a secret share of the index of the target.
These are the tricks required to perform MPC. More can be found on our github but we don't want to provide a public link to the work. Please reach out to us directly if you want access to the mpc code :).

We have implemented our honest but curious primitives in MP-SPDZ [Kel20], a popular implementation of the SPDZ [Dam+12] framework for secure multi-party computation using Python-like semantics. This includes anonymous reveal, role assignment, voting, and anonymous authenticated broadcast. This will work locally but has two "hacks".

1. Our MPC protocols don't represent the whole game logic, only parts of it we deemed necessary to be done completely privately.
2. We used a concept in MP-SPDZ called a trusted Python client, where each player has a Python client that will interact with MP-SPDZ clients to perform a series of MPC protocols one after another. If one client doesn't engage in the protocol, the clients won't agree to move on, and since each protocol is secure the game is secure.

Overall, we did implement all relevant game logic, i.e. the primitives that we describes for each role in a section previously.

One thing to note is this framework incorporates various computation modes under various threat models such as honest-but-curious and malicious (e.g. MASCOT [KOS16]). We aligned our threat model with the appropriate underlying protocol threat model.

9 Interacting with other players (And getting player inputs)

We then had the problem of actually getting different player inputs across the internet to each other. MP-SPDZ [Kel20] wasn't designed to actually be run over the internet, and requires a hard-coding of IP addresses to talk to. Obviously, people are not assigned IP addresses in this day and age, they only get private IPs protected behind a NAT wall.

So we used the help of our JS team, they will propagate the inputs from all users to all the computers running the MPC protocol locally for all 5 players. This "simulates" running mpc. To make this portable, we places all this code in a Dockerfile, that can be run on any system. Since this research project was designed as a proof of concept for running this type of MPC, we found our implementation to be good enough.

To talk with the application developed by our JS team we interacted through a state file, which Kenneth and I made and populated and read from on the Python side. What was left, was getting user input from the web application, displaying to the user what is happening in the game, and sending the inputs across to all computers. This was handled by the JS team.
In order to understand how both parts will talk with eachother the architecture of our game had to be designed (Yamm Elnekave). Hopefully, this picture helps make sense of what is going on.

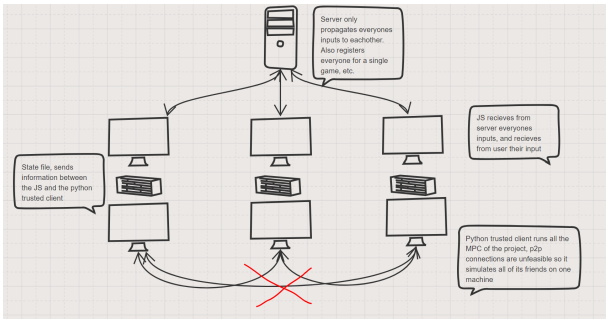

Figure 5: Architecture of the MPC integration with the Python client and JavaScript in the web application

Up until this point we described the research part of this project, which was entirely conducted by Kenneth Tang and Yamm Elnekave. We designed everything above including the protocols and implemented the MPC parts with MP-SPDZ. The following section describes the web application development side.

10 Web Application

10.1 State Management and State File

Neiro's focus and contribution to the project was developing the state.json format and the piping between the MPC and JS Frontend. The goal was to have the MPC and JS run separately and pass each other necessary game data in different phases. We had a well-defined enum that represented the phase of the game we were on and who was accessing the file. This follows a producer-consumer model that would clearly define when the JS or MPC side had access to the file. We also need to define a shared and personal state that each JSON file would contain. Some of the pieces we wanted to track were a global_enum, input, output, public-id-to-status mapping, detective-public-key, private-id, and detective-private-key. The private state was defined as the detective-private-key and the private-id. Each player would have a different private-id that defines their role (townsfolk, detective, mafia, or angel) and the detective-private-key was only assigned to a value for the detective. Everything else is a shared state that should be the same across all state files. This needed to be managed by the API we developed and we had some positive results in creating it along with some struggles. Initially, there was some confusion regarding how the MPC would format its outputs and how the JS would be able to read and utilize the

state file effectively. We eventually agreed that the frontend would read from output, write to input, and the MPC would do the opposite.

Neiro then developed JS functions for each phase of the game, including key population, detective phase, mafia/angel phase, and voting phase. They were all functional as they were tested to ensure that they wrote to the JSON file as tested by a separate test file. We then tried to resolve two main problems that could not be resolved in the state file logic. The two key problems that we needed to fix were data propagation and displaying the data on the front end.

The core issue we struggled with was that when we spawn instances of the game, we ended up having five different windows running with five different state.json files. Our state.json file includes a shared state, which every file should have the same after every phase. So, then we attempted to use the server as a crutch by distributing the final mappings and total inputs to the state.json file for the final write. This required asynchronous programming server side that our team found difficult. In our code, we have a framework of how we wanted it to work, but we were unable to complete it within the time window.

A couple of lessons that we would take away from this section of the project is that we needed to gain more clarity on the I/O expectations between the frontend and MPC. We initially spent a lot of time puzzling through how they were trying to pass data, which put us on a time crunch to resolve the asynchronous programming issues. We should have spent more time discussing agreements between the frontend and the MPC so that we collaboratively would have decided the state file. Better definitions of the abstractions we were creating would have led to better progress on the project in this phase.

10.2 Frontend/Backend

Andres focused on initially setting up the project, making sure there was a secure connection between the frontend/backend. This was done by using cors in the backend, like we learned in class. Additionally, the main routes were defined/set up which are userRoutes and GameRoutes. On the frontend side there were three main pages that needed to be set up, login, vote, and GamePlay. These were initially set up by Andres, however they were later improved upon by both Brandon/Andres. Another aspect of the frontend was that we needed to test the game logic, to do this the main problem encountered was that a localhost for 5 players doesn't work. To Bypass this a shell script was created that spawns 5 different localhost, although the backend cors() needed to be updated to accept these new ports. Additionally, on the frontend to test game logic faster, there is an Automation.js script that can be run that simulates the game, this helps with logic/testing of methods. Finally, while not fully completed(because we need to test more thoroughly before merging into main), we started adding in the state file with the frontend/using the backend to propagate data to all clients to help get away from using the server as the sole authority! A few lessons learned were that abstractions are very difficult if not clearly defined, connecting to the state file was very confusing because it wasn't clear the exact format to pass! However, this was finally figured out after a few meetings, thus before starting next time one should ensure there is a clearly defined rubric on what input/output look like! Also, Andres learned how cool MPC is. He had never heard about it before this project and learning about it/working with it was an amazing experience!

Brandon focused on the frontend and backend portion of the project. This involved the game logic, especially game stages, role specific actions, and overall game progression. This logic was created in JavaScript files Game.js and gameLogic.js. Firstly, different roles were created, such as detective. The detective is able to choose another player and reveal their role. This involved verifying a player's role as the detective and retrieving a selected player's role to the frontend using the players ID. Additionally, I implemented game stages. There are four main stages: one where the mafia kills, one where the angel, one where the detective reveals a player's role, and one where everyone votes. I enabled logic to ensure that no one can act when they shouldn't and carry out actions that they shouldn't.

In the frontend, Brandon designed and implemented dynamic UI elements to enhance gameplay and user experience. Labels were created to clearly display game stages, the end screen, and player statistics

such as kills, ensuring players remained engaged and informed throughout the game. Intuitive buttons were added to facilitate seamless player interactions and actions, while dropdown menus and additional on-screen elements were meticulously crafted to support a fully interactive and functional game interface. These enhancements adhered strictly to the rules of Mafia, delivering an authentic and immersive gameplay experience.

Later, Brandon played a role in integrating the frontend with the backend MPC logic. This integration required translating complex interactions between the two layers into JSON files, effectively bridging the gap between user inputs and the secure multi-party computation protocol. By writing and organizing much of the backend and frontend information into structured JSON files, Brandon enabled the system to accurately populate each party's individual game files. This step was crucial in ensuring proper game mechanics, smooth implementation, and alignment with the overarching protocol for secure and efficient gameplay

11 Summary

This project demonstrated the real-world application of MPC, showcasing its potential to enable decentralized systems like a game of Mafia without relying on a server. We proved it could be done, though we had to use the server to aid us in what we weren't able to do.

We successfully proved the concept, illustrating not only the feasibility of playing Mafia in a decentralized manner but also introducing a general framework for secretly distributing roles and performing actions with said roles in an anonymous authenticated manner.

This is new work, as unlike games like poker, where player abilities are transparent. When you reveal a secret you had such as a card in poker, that's it, you don't need to still hide it. In mafia, if you reveal that the mafia killed player x, but cannot show that you are the mafia, nor give someone else the capability of performing the mafia role, it becomes much more difficult.

12 Future work

Future work would include getting P2P connections to work as this will allow us to offload the work from the server. In this case the server's only job would be figuring out who wants to play together and acting as a TURN server, a server that releases other players IP addresses so that they can discover each other.

Another direction would be investigating how to make the protocol more efficient, either by optimizing MPC or finding alternate cryptographic primitives that could accomplish our goals.

References

[FY48] Ronald Aylmer Fisher and Frank Yates. *Statistical Tables for Biological, Agricultural and Medical Research*. 3rd ed., rev. and enl. London: Oliver and Boyd, 1948.

[SRA81] Adi Shamir, Ronald L. Rivest, and Leonard M. Adleman. "Mental Poker". In: *The Mathematical Gardner*. Ed. by David A. Klarner. Boston, MA: Springer US, 1981, pp. 37–43. ISBN: 978-1-4684-6686-7. DOI: 10.1007/978-1-4684-6686-7_5.

[Cha88] David Chaum. "The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability". In: *Journal of Cryptology* 1.1 (Jan. 1, 1988), pp. 65–75. ISSN: 1432-1378. DOI: 10.1007/BF00206326.

[Dam+12] Ivan Damgård et al. “Multiparty Computation from Somewhat Homomorphic Encryption”. In: *Advances in Cryptology – CRYPTO 2012*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 643–662. ISBN: 978-3-642-32009-5.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. “MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 830–842. ISBN: 9781450341394. DOI: 10.1145/2976749.2978357.

[Kel20] Marcel Keller. “MP-SPDZ: A Versatile Framework for Multi-Party Computation”. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020. DOI: 10.1145/3372297.3417872.

[Lin20] Yehuda Lindell. “Secure multiparty computation”. In: *Commun. ACM* 64.1 (Dec. 2020), pp. 86–96. ISSN: 0001-0782. DOI: 10.1145/3387108.

[NSD22] Zachary Newman, Sacha Servan-Schreiber, and Srinivas Devadas. “Spectrum: High-bandwidth Anonymous Broadcast”. In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022, pp. 229–248. ISBN: 978-1-939133-27-4.