Cheat Proof P2P Multiplayer Games

Contents

Introduction

The Problem

Many online multiplayer games have encountered problems with cheaters. These cheats can occur in a multitude of different manners, such as the client reporting a different state to their actual state (for example, saying the player in is one location when they are in another), or looking at additional information they are not supposed to have, such as packet sniffing techniques which can reveal where other players are located.

Ultimately, all these problems occur when a game involves hidden information, whether that is a set of cards in a player’s hand or their location on a map in a shooter. The problem is that a player both wants to hide their own information and verify that the other player is not lying about their information. Therefore, a trusted third party is required to check each player’s hidden information.

This is generally achieved through the use of a server, where players connect to the server, and the server validates each player’s status while revealing only the information each player actually needs. There are some drawbacks to this approach, one being that the players must still trust this server, but the biggest issue is that when the servers are shut down, the game is left unplayable.

This is where P2P protocols have an advantage. With a P2P protocol, players connect directly to each other without the need for a server, and so will be able to continue playing the game, potentially forever. However, without that trusted third party it becomes much easier for a player to lie about their hidden information. Traditional approaches to stop cheaters involve using anti-cheat software (which typically involves scanning a player’s computer for known cheating software) and closed source client code (a.k.a. security through obscurity[1]). Neither of these methods are very effective, and there are frequent reports of cheaters cracking these methods.

Project Aim

For this project I will explore the use of cryptographic techniques, with a particular interest in secure multi-party computation (MPC), as a method of securing a P2P multiplayer game. Given the lack of real security provided by having a program closed source, I will assume that the client software is open source and completely hackable. Therefore, the role of MPC will be used to replace the role of a server in other games.

To achieve this, I will be using SPDZ[17], which is a Python based framework for creating MPC applications. This library will make it easy for players to securely input numbers using Shamir shares and to perform basic arithmetic and comparisons, and then reveal the values to specific players or all participating players.

I will start this project by implementing a secure variation of the Mastermind board game[2]. This will give a relatively simple game to implement as it does not need to track state between rounds, or need any kind of randomness (e.g. dice rolls, card shuffling). These latter features will be explored with additional games, and towards the end of the project I will discuss the suitability of MPC for real time games, such as shooters.

Background

In this section I'll take a more technical look at how certain cryptographic techniques are implemented. I'll give a summary at the end that summarises the key points for each technique and why they are useful, so feel free to skip to the summary if you are not interested in the implementation details and just want to know why I've chosen to use these techniques.

Hash

A cryptographic hash function will convert data into a deterministic value, called a hash. A very simple naive implementation would be:

```
H(x) = x mod 10
```

This works well as a one-way/non-reversible function as there is no way to get back to the original value (e.g. 37 mod 10 = 7). However, it is not collision resistant as it is trivial to find another value which would produce the same output. Next, we'll take an overview of a more secure implementation. Many of these involve some kind of block based cipher. For example, if we look at the SHA-1 algorithm, the data is split into chunks and then each chunk is processed through the block cipher.

The process for this involves 5 values which are combined with a piece of the current chunk through a series of bitwise operations. This step is called a round and the data is processed repeatedly through 80 rounds, each round changing the value a little more. This works as the output of each round is used as the input in the next round while adding in each additional piece of the message chunk, continually shifting and modifying the hash.

At the end of the 80 rounds, the output is combined with the previous chunk's output and then the process is repeated on the next chunk of data, until the final hash is formed.

Commitment Schemes

A commitment scheme is a tool used to allow one user to commit to a value. It should be infeasible for the committer to change the value after the commitment has been sent and also infeasible for the verifier to discover the value that was committed to.

A simple commitment scheme is to use a hash function like H(R||m), where R is a randomly generated string prepended to the message. The resulting hash can then be sent to the other person as a commitment. If the value is later revealed, the verifier can prove that the values produce the same hash.

Zero Knowledge Proof

A zero knowledge proof is a cryptographic proof that the prover holds some piece of knowledge (for example, a password or answer to a problem) without revealing anything about that knowledge to the verifier.

The basic zero knowledge proof is often explained through a short story[9]. This involves a cave with 2 entrances that lead to dead ends, but there is a secret passage between them that can be opened with a password. To prove they know the password, the prover enters the cave unseen through one of the entrances, then the verifier flips a coin to randomly choose a side and calls for them to come out of that entrance. If they come out of the wrong entrance, this proves they do not know the password, while if they come out the correct entrance this gives a 50% chance that they do know the password. By repeating the experiment several times in a row, you can reach a very high level of confidence, for example repeating 10x in a row would give a cheater <0.1% chance of succeeding. For an alternative explanation involving a graph colouring problem, see [18].

The problem with this is the interactive nature of verifying this proof. In a game that needs to run quickly, requiring many roundtrips over the network between the players can prove to be too slow, especially if multiple proofs are needed. This is where non-interactive zero-knowledge proofs (NIZK) come in.

More advanced NIZK involve using commitments to hide the values. These more advanced methods can be used to provide proofs for any program that can be converted into a boolean circuit, thus making it generally possible to compute NIZK for a vast array of problems. These require a significant CPU overhead in order to generate the proof, but once sent to the verifier it can be proved in milliseconds.

NIZK is a little too complex to explain the implementation here, but there are some attempts at explaining simpler implementations. For a very simple example of a NIZK see [10], and for a relatively simple NIZK implemented as a boolean circuit using logic gates, see [11].

MPC

Secure Multiparty Computation (MPC) is designed to allow computing a program between multiple parties without involving a trusted 3rd party. The problem frequently referred to as an example is the millionaire’s problem, where a group of millionaires want to know who is the richest without revealing how much each of them is worth. Thus, they must compute the maximum of each of their inputs, without revealing the original inputs.

The implementation again resolves around boolean circuitry. But, there must be a protocol between all the parties to evaluate each gate together. Player inputs are then used through secret shares which split the value of an input across all the parties. This means that basic arithmetic like addition and multiplication can be performed locally (as all the shares will add up to the secret value).

Further information about MPC implementations can be found in [12], [13] and [14].

Summary

This is a summary of the key security points for the above cryptographic tools and how they can be useful for this research.

Hash

A cryptographic hash function is a one way function to transform a piece of data.

- One way: The output of a hash function cannot be reversed to discover the input.
- Deterministic: The hash function will always produce the same output when given the same input.
- Size: The output will always be of the same length, regardless of the input.

- Unpredictable: Any change in the input (even a single bit) will vastly change the output in an unpredictable manner.
- Uniform: Inputs have an even chance of being mapped to different values, in order to minimise collisions (where 2 different inputs produce the same output).
- Collision Resistant: Primarily as a result of being unpredictable and uniform (and with a sufficiently large output length), the hash function should make it computationally infeasible to find a collision with another input.

There are a number of ways we can make use of a hash function for games, such as verifying an input is the same value, without revealing what the input is.

Commitment Scheme

Commitment schemes allow a user to commit to a value, and another user to verify that commitment. For games, this could be used for players to commit to a choice without immediately revealing their choice. For example, in a card game, where the players need to select their cards, but don't want to reveal those cards until everyone has selected, we can have each player provide a commitment for their cards and only reveal the cards after everyone has provided a commitment.

Zero Knowledge Proofs

Non-interactive zero knowledge proofs (NIZK) allow one person to prove they have some piece of information without revealing anything about that information.

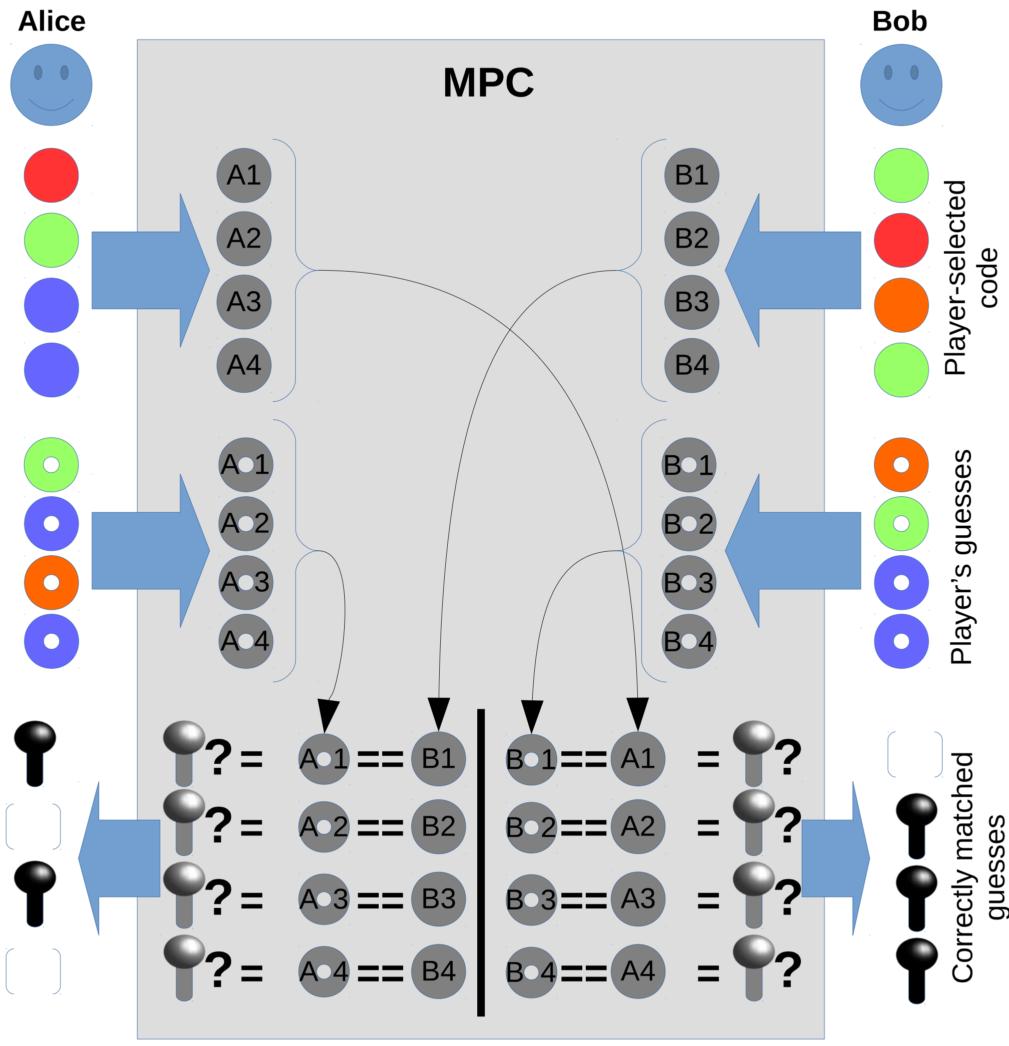
A non-interactive zero knowledge proof is possible, if a program can be written that can verify that a set of inputs are valid. Such a program can be converted into the required boolean circuitry needed to create a zero knowledge proof. This can be used for anything that can easily be verified by a program, but is difficult to solve with a program, for example it is easy to write a program to verify the answer to a large sudoku puzzle, but it is an NP-complete problem to write a program to solve such a puzzle.

This can be used, for example, for one player to efficiently prove that their inputs for a game are valid without revealing what those inputs are. Or, to prove they have a solution (such as the Sudoku example), without revealing what that solution is.

Secure Multiparty Computation

MPC allows multiple people to execute instructions as if there was a trusted 3rd party involved. For example, if you wanted to know who was the richest without revealing how much each of you were worth, you could each tell your worth to a trusted 3rd party and then that person would check which person gave them the highest value and announce to everyone which person that was. MPC allows you to achieve this without involving an actual 3rd party.

MPC can be thought of as a black box, into which each player can input their own secret values. Once in the box, nobody can see what those values are without explicitly agreeing to reveal them. These secret values can have arithmetic and basic logical comparisons performed on them. This allows complex programs to be built using these secret values, which then only reveal the result without leaking any information during the computation.



The diagram to the left shows how this might work. The players start by entering their code, and a guess at the other's code. Once entered into the shared MPC, these are secret values that the other player is unable to view.

The players then agree to do a comparison between each players' guess and the other player's code. The result is represented by the grey pegs, which are still secret values.

Finally, they agree to reveal the comparisons relating to each player's guess (represented by the black pegs). These results can only be seen by that player and are still secret to the other player.

One thing that makes writing these MPC programs tricky is that you can't write branching logic using the secret values. For example, imagine some code such as below, that you might write in a typical program:

```
        if a_secret == 3:
             b = 5
        else:
             b = 8
```

The problem with this code, is that if a_secret is a secret value, then we can't evaluate it in an if statement, as it's current value will be seemingly random, and not related to what it's secret value would be. Alternatively, if we wrote:

```
        a_public = a_secret.reveal()
        if a_public == 3:
             b = 5
        else:
             b = 8
```

This code would work, but in doing so, we've revealed to all participants what the value of a_secret was, and thus also what the value of b now is. If the goal is to keep these values secret (perhaps this is just the start of a calculation) then this code has failed our security requirements. Therefore, it is necessary to write the code like this:

```
r = (a_secret == 3)
b = (5 * r) + (8 * (1-r))
```

Here we first perform the comparison, which returns a new secret value which will evaluate to 1 or 0. Then we do 5*r which gives us 5 if r is true (1), or 0 if r is false (0). We add that to 8*(1-r), which does the inverse, resulting in 8 if r is false and 0 if r is true. This does the same thing as the previous code, but as no secret values are revealed, we have not revealed what the value of a_secret was, nor can we tell what value b now is after the operation, unless we later reveal its value.

Due to this restriction, it is trickier to write the programs needed to enforce the rules of a game compared to writing singleplayer or server side code.

Security Assumptions

I have assumed that a normal player is running an unmodified version of the software and that their computer has not been compromised. If either of these things have happened, then there can be no assumption of security, as someone may have changed the program behaviour, installed key loggers etc. Securing a person’s PC is outside the scope of this research.

An attacker is assumed to be a competing player who can freely modify their own version of the software, and may attempt to lie about their own data while trying to find secret information about the other player’s data. This is the primary focus of the research.

An attacker could be an outside party attempting to interfere with the game, but due to the fact that all communication is encrypted to the other parties, and secret values are shared in a way that only reveals the values to agreed players, it is unlikely there would be any information for an attacker to view or modify from outside.

A MITM (man-in-the-middle) attack would also be ineffective. The attacker would essentially just be playing 2 games, and because a player’s inputs are not revealed directly to the other player, they would either need to forward all messages unchanged in order to be valid, in which case they are nothing more than a relay, and once again are outside the game unable to see or modify anything. Or, they would have to make up their own inputs to each player, which is functionally the same as just playing 2 different games, which is not considered cheating. I have not aimed to provide any proof of identity when connecting to an initial player (which might be important for ranked matches or tournaments), again this is a separate subject which is outside the scope of this research.

Another assumption that comes from a limitation in the implementation of MPC, is that there must be >=50% honest players in a game. If more than half of the players are attackers collaborating together, then they can undermine the security of the MPC protocol.

Mastermind

The first game I implemented was a version of the Mastermind board game. This is a code breaking game, where one player picks a code of 4 colours, and the other player must make a guess at what the code is. For each correct colour in the correct location, the player is given a black peg, and for each additional correct colour that is in the wrong location, they are given a white peg. Using this information they need to correctly match the other player's code within 20 turns.

I slightly modified the rules, to have a more head-to-head approach, so both players pick a code and both must try and guess the other's code each round. Rather than limiting the game to 20 rounds, it is simply the first player to guess correctly wins. These 2 tweaks to the rules of the game mean that we don't need to store any changing state between rounds of the game, thus giving us a nice minimal set of requirements for the first project.

The code for this program is in the appendices.

The first thing we do after collecting the player's inputs is to perform some validation on those inputs (lines 11-18). This simply checks if any of the player's code values are <0 or >4, as we only allow 5 colours which are encoded as the numbers 0-4. If a player has entered an invalid value, the program will output a message here (which in a more complete game could be used to drop the connection from the cheater).

We then have a find_matches() function which implements the majority of our program. This function takes one player's guess and the other player's code. We run through each of the guess colours and count how many matches that colour has. If it matches at the same location in the code, we get an exact match (adding 1 to the number of black pegs) and any other positions that also match the colour are added up. If the number of previous matches (number of times we've already recorded a match for this colour) is less than the total number of matches, then (if it's not a black peg) we add 1 to the number of white pegs.

After running through all of the guesses, the total number of black and white pegs are returned.

Finally the program uses this function for both player's guesses and then reveals the counts to each individual player for their own guess.

Future Work

The missing part of this program, is that a player could change their code inputs between rounds. As can be seen in the placeholder in the code, we would want to compute a hash of each player's input as part of the program computation and then reveal that hash. This way, each player may check the hash is the same each round, to be sure the other player has not changed their code. But, the exact value we hash must be considered carefully.

If we were to hash only the code, of which there are only 5**4 (625) possibilities, then a player could easily hash each of those possibilities (likely taking little more than 1 second on a typical PC) and then match up the hash in order to find out what the other player's guess is.

The general security answer to this problem is to add a salt, which is a random known value concatenated to the beginning of the secret. We could have each player generate a random value and use this as a salt. This would ensure that the other player is unable to find a matching hash, as they would now have to guess the random salt (which can be any arbitrarily large value) in addition to the code.

However, this opens us up to a birthday attack.

> The birthday problem[3] asks the question: how many people must be in a room for it to be likely (>50% chance) that there will be 2 people in the room with the same birthday? An intuitive response considers that for someone with a particular birthday, they would need 183 people (365/2) for it to be likely that they would have the same birthday as someone else. However, because we are looking for *any* 2 people to have the same birthday, rather than for 1 specific person, this actually results in only needing 23 people to have >50% chance.
>
> The birthday attack[4] takes this idea to search for collisions (where 2 different inputs produce the same output) in hashes. So, we could search for random salt values with all the different code combinations, looking for any 2 that produce the same hash values. Because we are not looking for a collision with a specific value (which might have a 1 in 2^{128} chance), we have drastically reduced the difficulty of finding a collision (to around 1 in 2^{64} chance).

Because the player chooses their own salt, they could spend any amount of time pre-computing the birthday attack before they joined a game. Which makes this a difficult, but (depending on the choice of hash function) feasible attack which may be reused in as many games as they like once they've found a single collision.

If instead we used a salt from the other player, then we are back with the first vulnerability, as they know the salt, they know there are only 625 different hashes to check against.

The better solution is to take the salt from both players, concatenate them together and use this new salt for the hashes. Because each player only knows half of the salt, the previous 2 attacks are no longer possible.

Battleships

The other game I implemented was Battleships, based off the ruleset of the 1990 Milton Bradley boardgame[5]. The game consists of a 10x10 grid onto which each player places 5 ships, and then each round the player's announce a target and are told if it is a hit or a miss. The goal being to sink all 5 of the opponent's ships before they manage to sink yours.

The code for this program is a little more complex than the previous. The first step is to produce a list of coordinates for all the ships from the inputs which are given in the form of a starting coordinate and a horizontal/vertical flag.

We then have a function to validate the ship positions. The ships must be within the grid boundaries and not crossing each other. We start by checking each ship start position is within the valid positions (0-9), then calculating the end position of each ship and validating that. Then we check that no ships are crossing. Because we have already converted ships into a list of coordinates, we can simply check this list for duplicates. I use the knowledge that it is impossible for a ship to cross with itself in order to reduce the number of comparisons needed.

Next we start checking for hits. We first test each of the ships to see if the other player's guess matches any of the coordinates. Then we add any possible hit to the list of previous hits, so we can keep track of how many hits we have. Given that we can't know if we have a hit or what coordinate the hit is at, this is a little more complicated. Essentially we run through each value of the array and test if the value is a duplicate of the hit or if it is still a null value. We then update the value such that it will only be changed if this is the first null value we’ve seen and we do indeed have a hit and we’ve not seen a duplicate value.

Having updated our list of previous hits, we continue to test if any new ship has been sunk. To achieve this we check each ship coordinate against the list of previous hits. Then we test if the number of hits against that ship is equal to the length of the ship, which indicates a sunk ship. Finally we reveal whether the ship has been sunk or not and display a message if it has. We can choose to reveal this publicly or only to one player, hiding the progress from the other player.

Finally we reveal the values to see if each player has successfully made a hit or won the game by sinking all the ships.

Future Work

The one thing missing here is securing the list of previous hits each round. At the moment, a player could change the list of previous hits in a way to give them an advantage (essentially getting free turns).

Depending on exactly what security we want, there are a couple of solutions to this. If we are happy for a player to know all their previous hits, we can just do something similar to the Mastermind game and hash all the previous hits together (again with a salt). We can produce a hash at the end of each round from the new list, and then produce another hash at the beginning of each round and verify that the hashes are the same. To avoid leaking information about when a player has made a hit, we would also want to change the salt each round to ensure the hash is different even if the list of hits hasn’t changed.

Other variations of the game may involve not telling a player when they’ve got a hit, only telling them when they’ve sunk a ship, or alternatively allowing them to take multiple guesses but only telling them that they hit, but not which guesses hit. For this, we could get random values from the players as before, but using them as a password to encrypt the list of hits. Each round the players would then re-enter the encrypted list of previous hits and their half of the password, and it can then be decrypted, updated, and re-encrypted within the MPC computation without either of the players ever being able to see the contents. Again we would want to change the password or add a random value to the contents each round to ensure the encrypted output always changes, even if the list is the same.

Optimisations

One concern with this is the time to run these games. While they run without delay locally on a single system, I suspect further testing over a slow network may find these programs start to become a little slow, in particular with multiple round trips over the network being required. If this turned out to be true, there are some things we could do to improve the performance.

For example, in the Battleships game, the most complex and involved part of the program is the validation of inputs at the beginning of the program. However, we could potentially remove this from the MPC code and run it as a separate step without all the network roundtrips. To achieve this, we could make use of non-interactive zero-knowledge proofs (NIZK).

We could for example write a program that takes the player’s inputs and returns true only if they are valid inputs for the game. We can then create a program which takes the inputs, runs them through the preceeding program and returns the result of the program along with a hash of the (salted) inputs. This program can then be converted into a NIZK program, which can be run offline by the players to create a proof to be sent to the other player. They can then easily verify each other’s proof to be sure that hash was created from a set of valid inputs. This only requires a single message to be sent to each other containing the hash and proof. Then the validation in the MPC program only needs to generate the hash for each player’s inputs, so they can be sure the other player is still entering the same inputs that were used for the proof.

Evaluation of SPDZ

Writing basic MPC programs in SPDZ was relatively easy, once understanding what can and can’t be done with secret values and how to work with those limitations. One particularly useful thing that is missing though, is some built-in tools that would make it easy to perform basic cryptographic functions on secret values, e.g. hashes and encrypting/decrypting. There is an AES implementation given as one of the example programs, but copying that code into another program and trying to make use of the encrypt/decrypt features was too complex for me to work out in a short space of time.

The biggest issue though, is with integrating an MPC program with a larger application. In terms of creating a full game, we would want to restrict the MPC program to only handle exactly what is necessary to enforce the rules of the game, due to the complexity and inefficiency of the MPC program. We can then have this MPC program running as a component of the full application, where the rest of the application would handle rendering the game, handling input and any other features not strictly needed for enforcing the rules. For example, in the Battleship game, the player would expect their misses to be tracked as white pegs, but because previous misses don’t affect the rules of the game there is no reason to track them in the MPC program as they can’t be used to cheat at the game. The application surrounding the MPC program would be responsible for tracking these misses and displaying them to the user.

The problem with SPDZ is that there is no easy way to integrate a program. There is no API to run a program as a component, there are only applications and scripts to run a standalone program. This means to run it as part of an application, we would likely need to make a bunch of system calls and hope that everything is set up correctly to execute the program. Further complications come with input/output handling: inputs must be listed in a file and then a script exists to convert that into the correct format, after which that converted file must be copied into the correct location to load the player’s input. Similarly, for outputs which are revealed to specific players (rather than publicly) they are output to a file and must be converted again to be human-readable.

This additional complexity of needing to read/write and convert files for every invocation of the program makes it a lot less appealing to integrate into an application.

Ideally for future usage, SPDZ would be able to run as a Python module that can be imported and used to execute an MPC program and handle inputs/outputs directly. It would also have some builtin functions for hashing and encrypting/decrypting.

Ideas for Expansion

In terms of creating more complex games, one thing we can look is random number generation (RNG). There are methods of securely generating random numbers between multiple parties, e.g. by one user creating a commitment to a random value and sending it to the other user, who then shares their own random value, after which the first user can open their commitment and both can then use the 2 shared values as a seed value for a RNG.[6]

There are also variations that can be used to only reveal the random number to one particular player. With something like this, we could even input the secret random values into an MPC program and then use the random value without revealing it to any of the players.

Another interesting problem to look at would be shuffling. This could be shuffling a pack of cards between players or randomising the order of anything that might be used in a game. Generally this involves encrypting all the different cards, and then revealing keys to decrypt each card individually, see mental poker[8] and secure multiparty shuffling[15][16]. Again, we could use the keys in the MPC program to decrypt and use the cards without actually revealing the values. This may even allow all the cards to be encrypted with the same key, if the decryption will occur within the MPC program without revealing those keys.

Real-time games would be the most complex challenge to attempt. Some slower paced games could work, as the underlying rules of the game implemented in an MPC program could be implemented in an essentially turn-based manner, where the game state is updated every second or so; the game UI would make use of animations etc. to make the game feel like it is running in real-time. Done correctly, this would be indistinguishable from a truly real-time game. I believe this approach would work for most RTS (real time strategy) games.

For faster paced games such as FPS (first person shooter) games, this becomes much trickier. For example, we have to be aware of the amount of time elapsed between updates and verify if a player will have been able to move as far as they claimed to move. This could be really tricky to do, as you would need to account for network delays and random lag, while trying to stop somebody taking advantage by deliberately adding a delay to their responses.

Another thing that would be difficult to account for with multiple players is temporary disconnection/delays. If the game is getting updated 10x each second, and then one player has a temporary delay for half a second, how does the game cope with this? One approach might be to have a timeout, and then any players that have not entered their inputs are not included in that round of updates. This possibly has the advantage that a player who deliberately adds a delay would not be able to see any updates about the game state until they reconnected with their updated positions, thus removing the biggest advantage to cheating in this regard. Further research into MPC methods would be needed to fully evaluate exactly how feasible this approach is.

Summary

Through this research I’ve evaluated the feasibility of creating cryptographically secure peer-to-peer multiplayer games. I’ve shown that it is currently possible to create turn-based games that meet this criteria, although, currently, the tools are not easy to integrate into a project and would likely need some work to make this easier to use.

I’ve achieved this mostly utilising an implementation of MPC with some minor additional use of other crypto primitives such as commitments and encryption. I’ve also considered the use of NIZK as an optimisation to reduce the amount of networking required to evaluate an MPC program.

I’ve also taken a brief look at real-time games and some of the additional challenges this entails. It may be possible to create secure real-time games, but more research will need to be done to fully evaluate this.

References

[1] - https://www.schneier.com/crypto-gram/archives/2002/0515.html#1

[2] - https://en.wikipedia.org/wiki/Mastermind_(board_game)

[3] - http://mathworld.wolfram.com/BirthdayProblem.html

[4] - http://x5.net/faqs/crypto/q95.html

[5] - https://www.hasbro.com/common/instruct/Battleship.PDF

[6] - https://crypto.stackexchange.com/questions/12775/trustless-multiparty-random-number-generation#12778

[7] - https://crypto.stackexchange.com/questions/12742/shared-secret-generating-random-permutation#18993

[8] - https://en.wikipedia.org/wiki/Mental_poker

[9] - http://pages.cs.wisc.edu/~mkowalcz/628.pdf

[10] - https://blog.cryptographyengineering.com/2017/01/21/zero-knowledge-proofs-an-illustrated-primer-part-2/

[11] - https://people.xiph.org/~greg/simple_verifyable_execution.txt

[12] - http://research.cs.wisc.edu/areas/sec/yao1982-ocr.pdf

[13] - https://eprint.iacr.org/2008/068.pdf

[14] - http://u.cs.biu.ac.il/~orlandi/icassp-draft.pdf

[15] - https://crypto.stackexchange.com/questions/12742/shared-secret-generating-random-permutation#18993

[16] - https://eprint.iacr.org/2015/664.pdf

[17] - https://github.com/bristolcrypto/SPDZ-2

[18] - https://blog.cryptographyengineering.com/2014/11/27/zero-knowledge-proofs-illustrated-primer/

Mastermind Code

```
# Get inputs
p0_pegs = [sint.get_raw_input_from(0) for _ in range(4)]
p1_pegs = [sint.get_raw_input_from(1) for _ in range(4)]

p0_guess = [sint.get_raw_input_from(0) for _ in range(4)]
p1_guess = [sint.get_raw_input_from(1) for _ in range(4)]

p0_pw = sint.get_raw_input_from(0)
p1_pw = sint.get_raw_input_from(1)

# Check valid inputs
invalid_inputs = lambda pegs: sum((x < 0) + (x > 4) for x in pegs)
if_then(invalid_inputs(p0_pegs).reveal())
print_ln("P0 is cheating.")
end_if()
if_then(invalid_inputs(p1_pegs).reveal())
print_ln("P1 is cheating.")
end_if()


def find_matches(pegs, guess):
    # Black pegs are exact matches, same colour, same location.
    matches_black = sint(0)
    # White pegs are matching the colour, but in a wrong location.
    matches_white = sint(0)

    for i in range(len(guess)):
        # Get number of previous guesses of same colour.
        num_previous_same_guesses = sint(0)
        for j in range(0, i):
            num_previous_same_guesses += (guess[j] == guess[i])

        # Count the number of matches exact and non-exact
        num_matches = sint(0)
        for j in range(len(pegs)):
            if j == i:
                exact_match = (guess[i] == pegs[j])
            else:
                num_matches += (guess[i] == pegs[j])

        not_black = 1 - exact_match
        # If previous guesses is equal or higher than number of other
matches,
        # then we have already counted the match.
        has_match = (num_matches - num_previous_same_guesses) >= 1
        matches_black += exact_match
        matches_white += (not_black * has_match)

    return matches_black, matches_white

p0 = find_matches(p1_pegs, p0_guess)
p1 = find_matches(p0_pegs, p1_guess)

print_ln("P0 black %s", p0[0].reveal())
print_ln("P0 white %s", p0[1].reveal())
print_ln("P1 black %s", p1[0].reveal())
print_ln("P1 white %s", p1[1].reveal())

# TODO: Hash p0_pw + p1_pw + p1_pegs and vice versa. This can be used to
verify
# players have not changed their inputs between rounds.
```

Battleships Code

```
# Ships are:
#   0: Length 5
#   1: Length 4
#   2: Length 3
#   3: Length 3
#   4: Length 2

SHIPS_LEN = (5, 4, 3, 3, 2)
SHIP_NAMES = ("Carrier", "Battleship", "Cruiser", "Submarine",
"Destroyer")

# Get inputs
p0_ships = [sint.get_raw_input_from(0) for _ in range(15)]
# Start coordinates for each ship
p0_ships_x = p0_ships[0::3]
p0_ships_y = p0_ships[1::3]
# Direction of each ship:
#   1 - Horizontal (to the right)
#   0 - Vertical (downwards)
p0_ships_dir = p0_ships[2::3]
p1_ships = [sint.get_raw_input_from(1) for _ in range(15)]
p1_ships_x = p1_ships[0::3]
p1_ships_y = p1_ships[1::3]
p1_ships_dir = p1_ships[2::3]

p0_target = [sint.get_raw_input_from(0) for _ in range(2)]
p1_target = [sint.get_raw_input_from(1) for _ in range(2)]

p0_prev_hits = [sint.get_raw_input_from(0) for _ in range(34)]
p0_prev_hits = zip(p0_prev_hits[0::2], p0_prev_hits[1::2])
p1_prev_hits = [sint.get_raw_input_from(1) for _ in range(34)]
p1_prev_hits = zip(p1_prev_hits[0::2], p1_prev_hits[1::2])

def ship_coords(x, y, d):
    return [
        [(x[i] + (d[i] * offset), y[i] + ((1 - d[i]) * offset))
         for offset in range(SHIPS_LEN[i])]
        for i in range(len(x))]

p0_ships = ship_coords(p0_ships_x, p0_ships_y, p0_ships_dir)
p1_ships = ship_coords(p1_ships_x, p1_ships_y, p1_ships_dir)

def invalid_ships(ships_x, ships_y, ships_dir, ships):
    # Check start position of each ship in on the board.
    invalid = sum((ships_x[i] < 0) + (ships_x[i] > 9)
                  + (ships_y[i] < 0) + (ships_y[i] > 9)
                  for i in range(len(ships_x)))

    # Check ship end position is on the board.
    for i, l in enumerate(SHIPS_LEN):
        invalid += ships_dir[i] * ((ships_x[i] + l-1) > 9)
        invalid += (1 - ships_dir[i]) * ((ships_y[i] + l-1) > 9)

    # Check directions are 0 or 1.
    invalid += sum((d < 0) + (d > 1) for d in ships_dir)

    # Check ships don't cross each other.
    for i in range(1, len(ships)):
        invalid += sum(((a[0] == b[0]) + (a[1] == b[1])) == 2
                       for a in sum(ships[:i], []) for b in ships[i])
    return invalid

# Check valid inputs
print_ln("VALIDATING")
if_then(invalid_ships(p0_ships_x, p0_ships_y, p0_ships_dir,
p0_ships).reveal())
print_ln("P0 is cheating.")
end_if()
if_then(invalid_ships(p1_ships_x, p1_ships_y, p1_ships_dir,
p1_ships).reveal())
print_ln("P1 is cheating.")
end_if()

def append_to_prev_hits(prev_hits, target, hit):
    matches = sint(0)
    # Find the first null in prev_hits, and update it, but only if we had
a hit.
    for i in range(len(prev_hits)):
        # This avoids duplicates being stored
        matches_duplicate = ((prev_hits[i][0] == target[0])
                             + (prev_hits[i][1] == target[1])) == 2
        matches_null = prev_hits[i][0] == -1
        matches += (matches_duplicate + matches_null) > 0
        # matches will be 1 only the first time we encounter null (or
duplicate).
        do_update = hit + (matches == 1) == 2
        prev_hits[i] = ((prev_hits[i][0] * (1 - do_update))
                        + (target[0] * do_update),
                        (prev_hits[i][1] * (1 - do_update))
                        + (target[1] * do_update))

def check_hits(ships, target, prev_hits):
    # Record which ship is hit, if any.
    ships_hit = [sum(((c[0] == target[0]) + (c[1] == target[1])) == 2
                     for c in ship)
                 for ship in ships]

    ship_was_hit = sum(ships_hit) > 0
    append_to_prev_hits(prev_hits, target, ship_was_hit)

    # Check for newly sunk ship
    sunk_ships = [sint(0) for i in range(len(ships))]
    for i, ship in enumerate(ships):
        num_hits = sint(0)
        for cx, cy in ship:
            is_hit = sint(0)
            for px, py in prev_hits:
                is_hit += ((px == cx) + (py == cy)) == 2

            num_hits += is_hit > 0

        sunk_ships[i] = num_hits == len(ship)
        sunk_new_ship = (sunk_ships[i] + ships_hit[i]) == 2

        if_then(sunk_new_ship.reveal())
        print_ln("Sunk %s", SHIP_NAMES[i])
        end_if()

    return ship_was_hit, (sum(sunk_ships) == len(ships))

hit, won = check_hits(p1_ships, p0_target, p0_prev_hits)
if_then(hit.reveal())
print_ln("P0: Successful Hit")
end_if()
```

```
if_then(won.reveal())
print_ln("P0 has won.")
end_if()

hit, won = check_hits(p0_ships, p1_target, p0_prev_hits)
if_then(hit.reveal())
print_ln("P1: Successful Hit")
end_if()
if_then(won.reveal())
print_ln("P1 has won.")
end_if()

# TODO: Encrypt prev_hits and output. Then read in encrypted prev_hits to
avoid
# revealing any information and stop tampering.
```