B.Sc.Eng. Thesis
Bachelor of Software Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

Security and anti-cheating system in a turn based game

Major League Wizardry

Kristoffer la Cour (s113406)

Kongens Lyngby 2014

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Matematiktorvet, building 303B,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

This thesis looks at security and anti-cheating measures, that can be implemented in an advanced turn based game, where players have secrets to keep from each other, and the result of the game has to be reported to a third party that can verify the outcome.

This thesis have been written along side the development and implementation of the turn based digital trading card game Major League Wizardry, and is therefore focused on designing solutions that would increase security in Major League Wizardry.

Preface

This Software Technology thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfilment of the requirements for acquiring a bachelors degree in Software Engineering.

Kongens Lyngby, February 28, 2014

K. la Cour

Kristoffer la Cour (s113406)

Acknowledgements

I would like to acknowledge the following people for helping me throughout this project.

Ekkart Kindler, my supervisor, for making this project possible, for helping me throughout the project with guidance, and suggesting technologies of possible relevance, as well as proofreading my material.

Joseph Kiniry, my secondary supervisor, also for making this project possible, for initially setting me up with Ekkart, and for suggesting reading material and technologies of possible relevance for my thesis.

Kasper Lauersen, for helping with the general formatting and structure of my thesis, as well as proof reading it with me before hand-in.

Lotte Hauge, for helping me reflect on the basic parts of my design that utilize homomorphic encryption, and what properties would be needed of a homomorphic system to achieve my goal.

Contents

Summary i

Preface iii

Acknowledgements v

Contents vii

1 Introduction 1
1.1 What is Major League Wizardry . . . 1

2 Problem analysis 3
2.1 How Major League Wizardry works . . . 3
2.2 Possible attacks . . . 6
2.3 Game types . . . 7
2.4 Requirements . . . 7
2.5 Summary . . . 9

3 Existing techniques 11
3.1 Basic techniques . . . 11
3.2 Homomorphic encryption . . . 11
3.3 Fingerprinting . . . 13
3.4 Mental poker . . . 13
3.5 Bit-commitment and fair coin flips . . . 14
3.6 Timestamping . . . 16
3.7 Security by obscurity . . . 18
3.8 Summary . . . 20

4 Live game 21
4.1 Environment preliminaries . . . 21
4.2 Server controlled . . . 21
4.3 User controlled . . . 23
4.4 Summary . . . 30

5 Play by email game 31
5.1 Game creation . . . 31

5.2 Playing the game 32
5.3 Ending a game 33
5.4 Summary 34

6 Single player game 35
6.1 Server seeded 35
6.2 Distributed verification 36
6.3 Summary 37

7 Discussion 39
7.1 Reflection on homomorphic encryption 39
7.2 Resource priorities 39
7.3 Bugs and errors 40

8 Conclusion 41

Bibliography 43

CHAPTER 1

Introduction

This thesis will be looking at how to implement anti-cheating measures in the turn based trading card game Major League Wizardry [7]. The focus is on enforcing the rules of the game, in a way that makes it practically impossible for a user to circumvent them, or in other ways gain an unfair advantage.

The thesis will be looking at what kinds of different methods can be used to cheat, and what countermeasures could be implemented, that would make such methods futile.

The game itself (Major League Wizardry) have been developed by the company Game Made Studio [1] alongside this thesis.

1.1 What is Major League Wizardry

Major League Wizardry, (MLW), is a digital trading card game, originally based on a turn based play-by-email (PBeM) [9], concept. For a full understanding of the game and its rules, it is recommended, but not necessary, to read the analysis section in the report of our prototype application implementation [2].

The game also has a live game implementation, where both clients are connected to the server throughout the entire game, and a single player mode where the player plays against an artificial intelligence, (AI).

One of the core elements of MLW, are the cards. There are three kinds of cards in the current version of the game, spell cards, monster cards, and trap cards.

Each game is played between two players, where each have a wizard that represents them in that particular game. A wizard have a specific amount of health, referred to as health points. A player, or wizard, owns a deck, which is a list of cards that the player has defined before the game starts. When the game starts each player knows the content of his own deck, but not the order.

At the beginning of each turn, the active player (player1) draws a certain amount of cards from his deck, where the amount is defined by the game rules. These cards becomes visible to player1, but not the opponent (player2), as illustrated in Figure 1.1.

A player is able to play cards through paying mana, which is points that the player gains at the beginning of each turn, and through other actions in the game. The mana can be used to play cards from a players hand, which will put the cards into play. The amount of mana a player has, is also known by the opponent, but not how much mana is used each round.

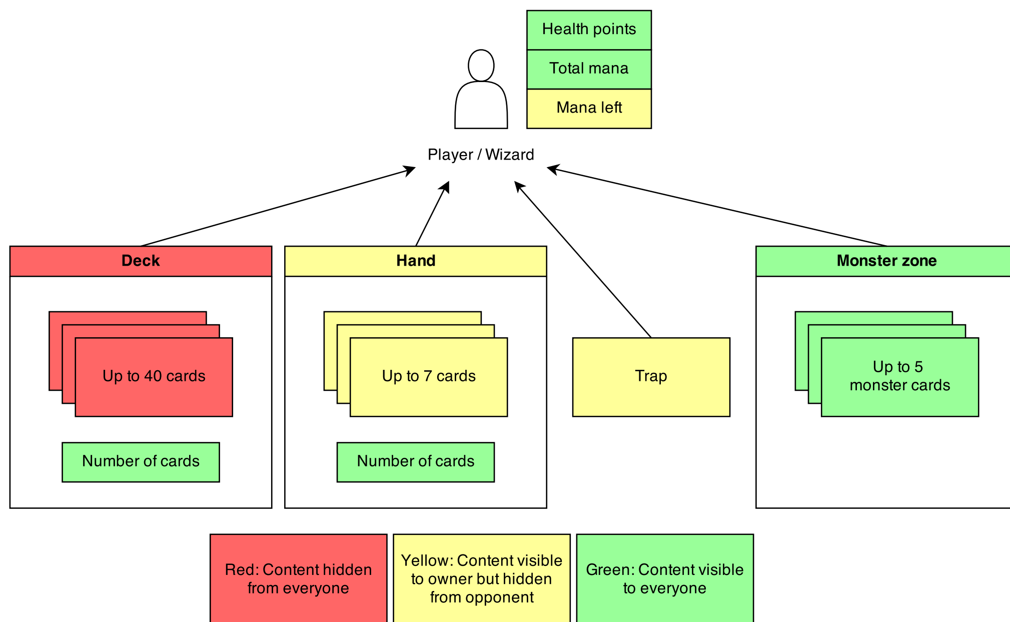

Figure 1.1: A player and the data connected to him during a game, color coded to represent the secrecy level of the data.

If a player plays a card from his hand, the card is revealed to the players opponent, unless it is a trap card, at which point the card should stay secret, but its effects should still be put into play.

A smart player would be able to know how much mana is used on monsters and spells, which is played openly, but not how much was used on the trap, and thereby implicitly know how much the opponent has left.

The trap card can be triggered on different events, depending on the trap, but what event the trap is triggered by is not revealed to the opponent before the trap is actually triggered.

For an example: Player2 plays a trap, that is triggered next time player1 play a spell, but does not show you the trap. In the next turn player1 try to play a spell, and player2 will then tell player1 that he activated the trap card, and show player1 the card and its effects.

A spell card is simply played, and its effects is executed immediately. A monster card is put into the monster zone, where it can defend the wizard, or attack the opponents wizard, or the opponents monsters.

All of these rules are what needs to be protected from cheating or manipulation, while playing in one of the three game types, PBeM, live, and single player, which is what this thesis will focus on.

CHAPTER 2

Problem analysis

This chapter describes the technical aspects of MLW, how the game works, the different elements of the game, how each element should be treated. It will look at what kind of threats to defend against, what the anti-cheating system will have to focus on, and what requirements there are.

2.1 How Major League Wizardry works

Based on what was investigated in the development of the prototype application [2], it is already known how some parts of the game is going to work. This section is meant to give an understanding of the technical aspects of MLW.

Server and client communication

When a game is initially created, the server matches two users, the player and his opponent, and creates a game state to send down to the users with use of serialization[11]. One user is then selected to take the first turn, and is expected to send his turn to the server, as shown in Figure 2.1.

Turns and moves

A turn is build of data describing each move a user have made. For an example, a move can describe the action of playing a card from the hand, where the data in the move would then be the action (playing a card) and the card effected by the action.

The turns and moves is executed in a linear order, the first turn in the list should be executed first, and the first move in a turn should be executed before the others. The two users continues like this until the game is over.

Ensuring the integrity and legality of these turns and moves is a requirement, since they are the building stones of a games current state, and altering them would mean altering the games state, which is considered cheating.

Randomness

Some of the cards in the game have a random chance to do one thing or an other, but when the game is run on one client, it needs to be executed the same way on an other, with the same values, which means that the randomness needs to be consistent.

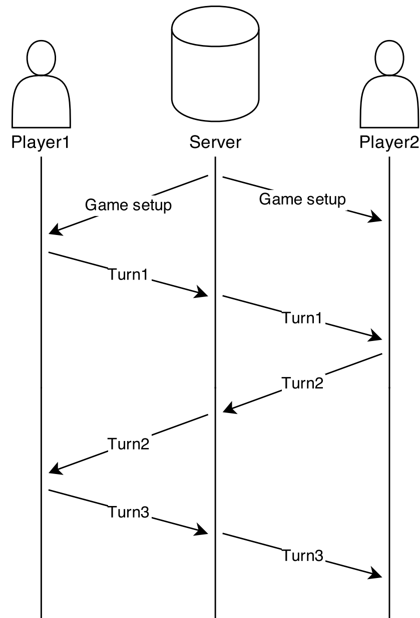

Figure 2.1: Time sequence diagram of how the server and users interact when playing a game.

A player must not to be able to predict or interfere with a random step, as this would be advantageous to him, or directly cheating.

Scripted cards

All cards have different abilities, which requires different code for every card. In order to gain flexibility in terms of editing cards or adding new ones to the application, all cards are scripted, and the client receives these scripts from the server, and executes them during a game when needed.

That cards are scripted means that they execute their code based on a string of text, and not machine compiled code like the rest of the application. This allows them to be edited without having to re-compile and re-distribute the entire application.

While this provides flexibility, it also provides a point of attack for a potential attacker. Changing the script of a card allows unwanted or incorrect code to be executed.

Game elements

The elements of the game itself has some rules that should be enforced, as described in Chapter 1. There are two players opposing each other in every game, where each

player owns a deck which is a collection of cards, a hand which is the cards a player has available to put into play, a "monster zone" where all active monster cards are represented, and a trap which is a card that is in play but hidden to your opponent. Each of these have different rules, and different valid actions that can be preformed on or with them.

The optimal solutions for each of these game elements would enforce the following rules.

A players mana

The amount of mana a player has can be described in different ways. There is the max possible mana a player can have in a given turn, and the available or unused mana a player has. The max mana available is public knowledge to all players in the game, but how much man a player have used or how much he have left is only for that player to know. However he should also be able to prove that he has mana needed available to preform a certain action.

The deck

The cards in the deck are selected by the user from his collection of cards, before the game starts, but shuffled so that the user does not know the order of them. The opponent should not know the cards in the deck or the order of them during the game. The cards in the deck are revealed to the user when he draws them into his hand. The amount of cards left in a deck is public knowledge.

The hand

The cards in the users hand is known to him, but not to his opponent. The amount of cards in a users hand is not secret to opponent. The mana cost of both spells and monsters played from the hand is published whenever the card is played.

The monster zone

Every card in the monster zone is public knowledge to all players in the game. The mana cost of a monster in the monster zone is public knowledge as well.

The trap card

The trap card should be hidden from the opponent, even when played, until the trap is triggered by an action preformed either by the player or the opponent. The action that would trigger the trap, should also be a secret to the opponent, until he or the user preforms this action. The mana cost of the trap should also be a secret hidden from the opponent.

Anatomy of a game

In order to be able to discuss the aspects of the game, a technical anatomy of a game and its life-cycle, as well as the actors involved in a game, is described in this section.

Game creation phase

Initially two users decide they want to play a game against each other, either by challenging each other directly, or by being introduced to each other by the server in a random match. These users will be referred to as *Alice* and *Bob*, and whenever a specific user in being discussed, they will be referred to as the *User* and his *Opponent*.

When the users have been introduced, the game needs to be created, and both users need to choose which deck they will be using. This can either be done in a way where a user does not know what deck his opponent chooses, or where one user openly chooses his deck and before challenging the other. The deck will be referred to as *Deck*, while a card will be referred to either as *Card* if it does not matter which type the card is, or by the cards type, *Spell*, *Trap* or *Monster*.

When both decks have been chosen, they need to be shuffled, and a starting player needs to be chosen. When this is done, the game is ready to begin, meaning that the game creation phase is over.

Game phase

Now that the game has begun the game phase starts, which is build out of turns and moves. One player only gets one turn at a time, and each turn consists of the moves he made that turn.

Every move have the potential to trigger a new script in a card that is in active play, such as a trap or a monster in the monster field, or a spell that is still in play.

End phase

The game phase continues until the game ends, either by a user running out of cards in his deck or loosing all the health points his wizard has.

When the end is reached, a winner have to be declared, and reported to the server so it can record the game results, and give a reward to the winner.

2.2 Possible attacks

Before attempting to protect the game against cheating, it is important to understand what to protect it against, and what to avoid.

In theory, a skilled hacker can access everything, and do what ever he wants on the client side, he can even write a custom client in order to have total control of everything that happens.

However, there are some techniques an attacker is more likely to use, such as accessing the memory [3, p. 118] of a game, in order to try and change a value in the game that would benefit him, for an example how much damage a monster does.

An other angle of attack is tampering the communication between server and client, e.g. to change the values used in a move, such that he may make a move he was not otherwise allowed to.

One of the most powerful attacks, is decompiling the code, in order to change what it does, and then re-compile a modified client.

A basic assumption is that he client cannot be trusted, and there should therefore be limited possibilities in terms of sending invalid information to other parties.

2.3 Game types

As mentioned there are some different kind of game modes, live, PBeM and single player. A protocol design for these game modes will be discussed individually, where an attempt will be made to determine what is needed to make the game safe, and what can be done in order to detect cheating.

The live game

is what offers the most to work with, since all players are available during the entire game. The design of the live game will at first focus on how to make the game secure, then on how you can make the game secure without having the server involved, and still be able to report a winner to the server after the game is over, in a way that no one can deny the true winner of the game.

The play-by-email game

or the PBeM game, refers to playing a game, where one the moves of one turn is send from player1 to player2, player2 then reads it and replies with his moves. PBeM is a bit more challenging, especially because a player preferably, during the game, should not contact the server at other times than when he receives a move from the opponent, or when he sends a move him self.

The single player game

can be done in a couple of different ways, the game could either be played at the clients side, or the server could initialize the game and confirm the game in the end, or the player could even play against the server the entire game.

2.4 Requirements

The following is an overview of the general requirements for the design of the three different game types.

Secrets

As mentioned, there are multiple game elements and values that are secret to different players at different times. A big part of the design should make sure these secrets are not revealed to the players before they should be. This involves the content of the decks, the mana a player has left, and the trap card.

Randomness

A lot of things in MLW depend on randomness, the most apparent one is shuffling the decks, but some cards use random values as well. It is important that the randomness is not manipulated or foreseeable by any of the players in the game.

Cheat detection

If a player were to make an illegal move or try and manipulate any other aspects in the game, it has to be detected and even further proved, so that one player cannot unrighteously accuse the other of cheating.

Server load

The server load should be kept at a minimum, which means that the users clients should do most of the computational work, and the server should do as little as possible.

Response times

Some things have a time limit. For an example in a live game a player should only have 90 seconds to makes his moves each turn. If the 90 seconds pass, it should be the other players turn. The design should be able to enforce these time limits.

Proof of game result

When two players have played a game, it has to be possible to prove to the server who the winner is, even if the looser tries to sabotage the game.

Flexibility

Another requirement of the game is that it should be flexible. Every card in the game has different abilities, which on the physical version of the cards is described on the respective card, and can in theory circumvent *any* of the other rules defined for the game. For the sake of having at least some boundaries to work with, this is mostly minded on changing the state of a card in play, or forcing your opponent to reveal a secret or dispose of a card, e.g. a card may say that it should reveal the opponents trap card. This, combined with the play-by-email protocol, can make it very hard to

keep any secrets in the game, since a card in theory can grant you permission to read that secret.

2.5 Summary

This chapter described how a game of Major League Wizardry works in a more technical aspect, and which aspects of MLW are important to focus on in terms of anti-cheating. What a player can do in order to attack the application was briefly covered, as well as the different game types a protocol should be designed for.

In the end the requirements were listed, in order to give an overview of the focus areas this thesis will be covering.

CHAPTER 3

Existing techniques

This chapter looks at different technologies and methods, that have been suggested by supervisors, or found by research, in the process of coming up with a secure design for the implementation of MLW.

Strengths and weaknesses of each technology is found, and discussed whether or not they can potentially be used in our implementation.

Before reading about the more complicated techniques in this chapter, there are a few basic cryptologic and general techniques the reader should know about, which will be covered briefly in the following section.

3.1 Basic techniques

Hash functions or one-way functions, are functions that take a piece of data and computes a value based on the data. If you change some of the data and run it through the hash function again, you will get a different value. This is practical if you want to check weather it is the same data used to produce a specific hash value, or just ensure that the data have not changed.

Public-Private key cryptography you generate a key-pair, a private and a public key. If you then encrypt something with the public key, you would need the private key to decrypt the message. Some public-private key algorithms are also able to preform a task called signing, if you have the private key. Signing is basically that you encrypt something with the private key, that can only be decrypted by the public key generated along with it.

Random seeds There exist some functions that take a string of data as input, and uses this to generate seemingly random data. The string is called the seed, and if the same seed is used with the same function, the same sequence of pseudo-random numbers can be generated. This is useful if you want a consistent stream of pseudo-random values.

3.2 Homomorphic encryption

Whenever you encrypt something, you get a piece of cipher-text, that can basically not be used for anything but representing the data, until it is decrypted.

Homomorphic encryption refers to a cryptographic system that would allow computation of certain functions with encrypted data, without the decryption key (e.g. multiplication or addition of two encrypted values).

Recently scientists have begun attempting to design fully homomorphic encryption schemes, where the basic requirements are that you should be able to both multiply and add two encrypted numbers with each other, getting the resulting encrypted number, without knowing the decryption key, or the numbers for that matter.

The first fully homomorphic encryption scheme was announced in Craig Gentry's thesis in 2009[4]. In his thesis, Craig Gentry describes a system that uses a private and public key system, where the public key is used by the functions that is used for performing calculations on encrypted data, and the private key is used to decrypt and encrypt values. Using such a system it would be possible for a server to preform calculations or queries for a user, while keeping the values involved in the calculations private, and without the server even knowing what it would be sending back to the user.

Paillier cryptosystem

Certain well known encryption schemes have some homomorphic properties, one of which is called the Paillier cryptosystem, that has additive homomorphic properties, along with other properties. [5, Sec. 8]

The Paillier system is based on the private and public key concept, where the public key is used for homomorphic manipulation. The public key consists of two numbers, n and g, where n is the product of two large primes, and g is an integer chosen at random, such that $g \in \mathbb{Z}^*_{n^2}$.

As described in [5], one of the homomorphic properties of the system allows you to take two chiphertexts, and cumpute a new chiphertext, that decrypts to the sum of the two:

$$\mathrm{ADD}(E(m_1), E(m_2)) \to D(E(m_1) \cdot E(m_2) \bmod n^2) = m_1 + m_2 \bmod n \tag{3.1}$$

Another property that makes this particular cryptosystem usefull, is that every time a message m is encrypted, a new random value r is used to create the chipertext, which means that if you encrypt the same value twice, the chiphertext would not be the same. A chiphertext is generated as follows:

$$c = g^m \cdot r^n \bmod n^2 \tag{3.2}$$

As can also be seen from (3.2), the only thing needed to create a chiphertext that is compatible with the key pair, is the public key. This means that the party that does computations on the values can introduce new values, without having to get them from a party with knowledge of the private key.

Multiplication of a chiphertext is also possible, though not with another chiphertext, but instead with a normal unencrypted integer.

$$\text{MULT}(E(m), k) \to D(E(m)^k \bmod n^2) = k \cdot m \bmod n \quad (3.3)$$

Feature set of the Paillier cryptosystem

- Allows computation on encrypted values.
- Different chiphertexts for the same message.
- Allows introduction of new values without the private key, using the public key.
- Anyone can introduce new values using only the public key.

3.3 Fingerprinting

Fingerprinting refers to taking a piece of data e.g. a file, and computing a value based on it, that would look different if any of the data was changed. This is basically running the data through a hash function, though some fingerprinting algorithms does exist, the fingerprint algorithms are focused on being low-performance, but are typically easier to cheat than a hash function.

The simplest way to make a fingerprint is simply to run whatever data you want a fingerprint of through a hash function $Fingerprint = Hash(Data)$.

Feature set of fingerprinting

- Enables representation of data without revealing it.
- The fingerprint is mostly much smaller than the data represented.
- Requires the data holder to present the data if people need to verify that a piece of data is connected to a specific fingerprint.

3.4 Mental poker

Mental poker [6, p. 92] refers to a cryptographic protocol that allows players to deal cards amongst them without anyone being able to read the other players cards, and without the dealer knowing what cards he is dealing.

The original algorithm from is build for multiple people, where some of the core steps in short works like this:

- Each person generates a public and private key.
- The first party, Alice, then takes the list of cards, encrypts each of them, along with a random string for each card, with her public key, shuffles the order of the deck, and hands them over to the second party, Bob.
- Bob then picks picks five cards he encrypts with his public key, and sends them back to Alice.
- Alice then decrypts the messages with her private key. Still unable to read them, since they were also encrypted with Bob's public key, Alice sends the cards back to Bob, so that he can now decrypt them, and know which cards he chose.

This leaves Bob with a fairly dealt hand, and the rest of the deck. If Alice wants cards, Bob can simply deal them to her by sending them back to her without encrypting them, so that she may know her hand.

If a third person were to join, Bob can either deal them cards, or send the deck to them in order for them to draw cards them self. They would then use the same scheme as Bob, and encrypt them with their own public key, then send them to Alice, in order to have her decrypt them.

At the end of the game everyone reveals their hands and private keys, so that everyone can verify that no one has cheated.

Feature set of mental poker

- Allows people to shuffle or randomly distribute values between them, without revealing, to anyone but the people possessing the values, who possesses which values.
- Does not require interaction with a third party.
- Reveals the total number of values being shuffled or distributed.
- Requires contact to Alice and Bob every time a new value is distributed.

3.5 Bit-commitment and fair coin flips

Bit-commitment

Bit-commitment [6, p. 86] is a cryptographic protocol where a person commits to choosing a piece of information, without revealing what piece of information that person is committed to.

The way this is often done is by using one-way hash functions to commit to a message, that you will then later reveal to the other party.

For an example if Alice and Bob are watching a re-run of a football game together, Alice claims to know the outcome, but does not want to ruin the game for Bob. In real life, Alice could simply write it down and put the answer in an envelope, that Bob could then open later, to confirm that Alice indeed knew the outcome of the game.

If the same is to be achieved in the digital world, Alice can use a *bit-commitment blob*. A blob is a string that Alice can send to Bob, and then later open, by sending Bob a bit more information. Bob can however not find out what value Alice is committed to, and Alice has no way of changing the value.

An example of creating a blob with an one-way hash function:

1. Alice writes down the result of the football game $Result$, and generates two random strings R_1 and R_2.
2. She generates a blob value by combining the result and the two random strings $Hash(Result, R_1, R_2)$.
3. She sends the blob along with R_1 to Bob, and is now committed.
4. When the game is over, she sends the last random string, along with the result to bob, so he can verify the hash value.

Fair coin flips

Another protocol that has sprung from the bit-commitment protocol is called fair coin flips [6, p. 89], which is used to allow two parties to fairly flip a coin between the them.

This simply works by Alice and Bob both generating a random bit, which they use to generate the outcome of the coin flip. The bit-commitment is used in order to prevent any of the parties, to generate their bit based on the other party's choice, in order to generate a desired outcome.

One party commits to a choice, the other party openly makes his choice, they then compare choices in order to determine a result.

PBeM example An example of how you can use bit-commitment along with fair coin flips, in the PBeM format, could be, that Alice and Bob are playing rock-paper-scissor over email. Usually this would be difficult because, neither party can tell the other what they would choose, without the other party having the information of the their opponents choice available, when they get to reply with their choice.

This is where the bit-commitment scheme comes into play:

1. Alice makes her choice and chooses paper C_{paper}.
2. She then generates two random strings R_1, R_2 .
3. She then computes the hash value $H(C_{paper}, R_1, R_2)$.

4. She sends the hash value along with R_1 to Bob $H(C_{paper}, R_1, R_2), R_1$.

Alice is now committed to the choice, since she have given Bob the hash value, as well as one of the two random strings, she can not alter her choice without breaking the hash function used.

Bob can now make his choice, and tell Alice directly, without the need for any encryption or hashing, and then require of Alice to reveal her choice.

5. Alice then simply sends the second random string R_2 along with her choice C_{paper} to Bob.
6. Bob then computes the hash value him self, using the random strings and Alice's choice, and compares it to the value Alice used to commit to her choice with. If the values match, the choice Alice sent along in the end can be deemed valid.

Other bit-commitment schemes exist, but this scheme has been chosen because it has the advantage of not requiring Bob to do anything, except validating the message in the end, and allows Alice to commit to the message in one single e-mail, and reveal the message in an other.

Bit-commitment can be used for hiding a value from a user, while still letting the user know the value exists, until the value at some point comes into play. While fair coin flips can be used to fairly make a random choice, and without anyone being able to predict the outcome.

Feature set of bit-commitment

- Forces a person to stay with a choice, without revealing it.
- Fair coin flip: Allows two persons to choose two dependent values independently.

3.6 Timestamping

Timestamping [6, p. 75] refers to recording a date a given document were in a given state, which is used, for an example, to prove that a contract has not been altered since that time.

Actually timestamping a document usually required a trusted timestamping service to record the current time and state of the document. This is because it is too easy for Alice to just make up her own time, and then claim that the document was stamped at that given time.

A simple way to timestamp a documet is to make the trusted timestamping service, who will be referred to as Trent, store a copy of the document along with the time. Then if Bob wants a proof of the timestamp, he can ask Trent when he received

the document. This does however have some drawbacks, Trent have to store the document, and if Alice wants a large document signed the computing, storage, and networking resources required to maintain the timestamp grows.

A far more resource friendly method for documenting a timestamp is to use one-way hashing and digital signatures to record a timestamp.

- Alice computes a hash value of the document $H(Doc)$ and sends it to Trent.
- Trent appends the time and date to the hash value $Sig_{Trent}(H(Doc), time)$, signs it and sends it back to Alice.

Now if someone wants a proof of the timestamp, Alice can simply present them with Trents signature of the data. This way Trent does not even have to store the record of the document, and Bob can get a proof from Alice directly, even without Alice revealing the content of the document, since the hash value serves as proof of the documents state.

Linked timestamping protocol

A different approach, that is meant to increase trust in the timestamp, is to use a protocol where the timestamps Trent provides are linked to the other timestamps he makes, meaning that Alice's timestamp are depend on the timestamp Trent issued before Alice made her request.

For an example if Alice makes a request for signing a hash of the n'th document $H(Doc_n)$. Trent will then timestamp $H(Doc_n)$, and sign it along with the timestamp made right before Alice's request, as well as the identity of the person that requested that timestamp, and a hash value L. L is based on the previous timestamps info, including the L hash value used in the previous timestamp, and serves as the link back to the previous timestamp.

This way every timestamp linked to the previous one, and when Trent timestamps the next document, Alice will be sent the identity of the next person in the chain, and the next timestamp will be linked to Alice's information.

If the timestamp Alice has for her document is questioned, she can contact the persons before and after her, and ask them to present theirs timestamp, which are linked to hers. The information in theirs timestamp will then correspond to Alice's timestamp information, and at minimum prove that Alice's timestamp comes after the person before her, and before the person after her.

Feature set of timestamping

- Provides proof of a piece of data's existence at a given time.
- Makes it practically impossible to alter an entry after the a timestamp have been issued.

- Linked protocol: Forces entries to be linked in continuation of each other.
- Requires a trusted service to issue timestamps
- Linked protocol: Extra validation requires contact to the linked parties, who might not be reachable at the given time, if Trent's honesty is questions.

3.7 Security by obscurity

Some other more commonly used non-cryptographic technologies exist, but are often referred to as bad practice by security specialists, since they are usually used to try and hide errors in a fragile design.

They can however act as a minor barrier for inexperienced or impatient attackers, which means that even though these solutions does not secure your system, they do in practice help keep some people from tampering with it.

Code obfuscation

Code obfuscation refers to making your code hard to read for humans, while still maintaining the same functionality when the computer interprets it. Usually this is done by a program, that is designed to obfuscate your code for you.

Listing 3.1 shows an example of a JavaScript function before obfuscation, and listing 3.2 shows the same function after it has been run through an obfuscation program.

```
function updateHealthPointsBy(monster, wizard, game, damage, source) {
  if (damage > 0) {
    if (monster.eventBeforeMonsterHealed(damage, source)) {
      monster.currentHealth = damage + monster.currentHealth;
      monster.eventOnMonsterHealed(damage, source);
    }
  } else if (damage < 0) {
    if (monster.eventBeforeMonsterDamaged(damage, source)) {
      monster.currentHealth = damage + monster.currentHealth;
      monster.eventOnMonsterDamaged(damage, source);
    }
  }
  monster.deathCheck();
}
```

Listing 3.1: Un-obfuscated JavaScript function.

Obfuscation is often used in an attempt to hide code from users of a program, either for copyright reasons, or in order to make it harder for a hacker to manipulate the program.

```
var _0x9cf3=["\x65\x76\x65\x6E\x74\x42\x65\x66\x6F\x72\x65\x4D\x6F\x6E\x73\
    x74\x65\x72\x48\x65\x61\x6C\x65\x64","\x63\x75\x72\x72\x65\x6E\x74\x48\
    x65\x61\x6C\x74\x68","\x65\x76\x65\x6E\x74\x4F\x6E\x4D\x6F\x6E\x73\x74\
    x65\x72\x48\x65\x61\x6C\x65\x64","\x65\x76\x65\x6E\x74\x42\x65\x66\x6F\
    x72\x65\x4D\x6F\x6E\x73\x74\x65\x72\x44\x61\x6D\x61\x67\x65\x64","\x65\
    x76\x65\x6E\x74\x4F\x6E\x4D\x6F\x6E\x73\x74\x65\x72\x44\x61\x6D\x61\x67\
    x65\x64","\x64\x65\x61\x74\x68\x43\x68\x65\x63\x6B"];function
    updateHealthPointsBy(_0x644bx2,_0x644bx3,_0x644bx4,_0x644bx5,_0x644bx6){
    if(_0x644bx5>0){if(_0x644bx2[_0x9cf3[0]](_0x644bx5,_0x644bx6)){_0x644bx2
    [_0x9cf3[1]]=_0x644bx5+_0x644bx2[_0x9cf3[1]];_0x644bx2[_0x9cf3[2]](
    _0x644bx5,_0x644bx6);} ;} else {if(_0x644bx5<0){if(_0x644bx2[_0x9cf3
    [3]](_0x644bx5,_0x644bx6)){_0x644bx2[_0x9cf3[1]]=_0x644bx5+_0x644bx2[
    _0x9cf3[1]];_0x644bx2[_0x9cf3[4]](_0x644bx5,_0x644bx6);} ;} ;} ;
    _0x644bx2[_0x9cf3[5]]();} ;
```

Listing 3.2: Obfuscated JavaScript function.

However, the only thing this achieves is making it harder, it is not a real solution to the problem of people being able to find out what your program does, since the code is still the same, just harder to read for a human. It will act as a barrier for potential hackers, but not an unbreakable one.

Obfuscation would not be an acceptable solution for banking software, but for something as simple as a harmless game, it can be a viable solution to low-priority problems.

Memory encryption

An other method that resembles obfuscation, is trying to hide important values in your program, by encrypting them. This method is used because usually when someone wants to find a value in memory, they look for a place in memory where that exact value appears.

However, you still need to be able to decrypt them when your program has to use them, and would have to be able to get the decryption key.

You can either let the user input the decryption key whenever the values needs to be accessed, or you can store the key somewhere in the system.

If you just store the key somewhere, it only means that an attacker would have to find the decryption key first, and then the values he was looking for, which means that it does not solve the problem at hand, but instead only makes it more difficult to find what you are looking for.

If you let the user input the key, you protect against other people trying to access information in the users system, but if the goal is to prevent Bob form accessing something in his own system, having him hold the key is no use.

Feature set of security by obscurity

- Can make it harder for an attacker to locate a vulnerability in a system.
- Is not an actual solution to the problem it attempts to solve.

3.8 Summary

This chapter covered how the use of basic cryptologic techniques allows you to set up protocols, that can allow you to hide secrets, create fair randomness, prove the integrity of your data or the time of which you were able to present a piece of information. It explained how existing cryptosystems makes it possible to preform calculations on data you do not have cleartext representations of. It also discussed obfuscation and memory encryption, and why these widely used techniques are not perfect solutions.

CHAPTER 4

Live game

This chapter describes the different problems involved with playing a live game of MLW, proposes different solutions to the problems, and discuss the advantages and drawbacks of each solution.

4.1 Environment preliminaries

Before continuing to the design of the game protocol, the environment should be set up, so that there is a firm ground to build the design upon.

The actors of the system is Alice and Bob, that are both users playing a game against each other, and each can attempt to chat.

The design assume that all connections between parties are secured through standardized protocols like TSL [8], or use of private and public keys previously shared between parties.

Then there is a standard registration and log-in system, based on email registration and password authentication.

An other assumption is that all users know the servers public key, which can be coded directly into the game application, which is distributed through trusted providers, or shared through trusted public key registration authorities in a public key infrastructure[10].

In order for users to communicate with each other, each will assigned a private and public key pair, where the private key will be known to the user it is assigned to. Alice generates her own private and public key, then registers the public key with the server through normal password authentication, and gets a server signed version of the key along with Alice's ID $Sig_{Server}(ID_{Alice}, PubKey_{Alice})$. She can use this to share her public key with Bob, if she wants to start communicating with him, and since the server has signed it, Bob can trust that the key belongs to Alice's ID.

4.2 Server controlled

An easy way to prevent cheating, is to let the server do everything. Choosing the decks and pointing out a beginning player is the only thing that needs to be done, from there on the entire game could be played out on the server. The server only sends the information a player need, whenever a player makes a move he reports it to the server, the server then tells the player what happens by creating an event chain it

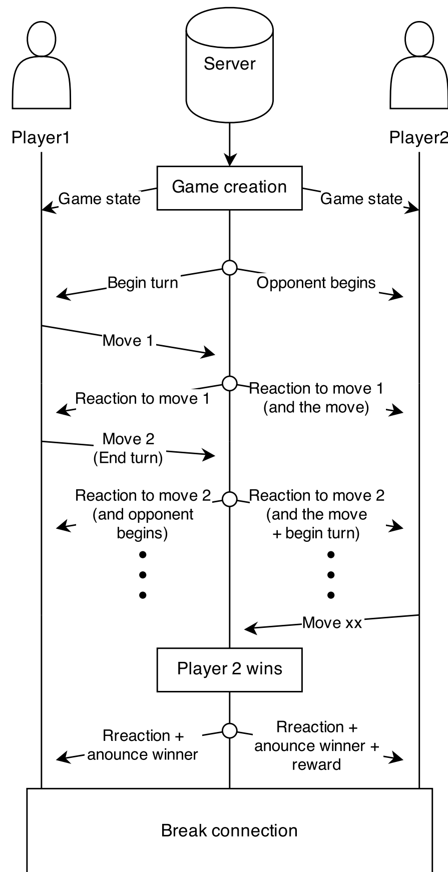

Figure 4.1: The process of a server controlling the entire game illustrated by a time sequence diagram.

sends back, along with whatever values that would be needed to complete the move. Whenever a winner is found, the server announces it, and rewards him, as shown in Figure 4.1.

This allow us to make sure everything is ran as it should be, since the server is a trusted party and handles everything. The only thing it needs to do is validating that it receives input from the correct users. However, it is a huge load on the server to actively execute every command and report every move in every game played, and would quickly exhaust the servers resources, thus breaking our priority of a low server

load.

4.3 User controlled

If the protocol is designed a little different, it is possible to allow two users to play a game together, with very minimal contact to the server, while still keeping everything secure, and even reporting a winner to the server in the end, along with a proof that the game took place in a fair manor. The goal here is to take as much work as possible away from the server.

Game creation

The first thing that needs to take place is the game creation phase. Assume that Alice and Bob have made contact to each other, and now want to start a game together. The first thing they need to do, is choose which decks they want to use, then shuffle them, and choose a beginning player.

Choice of deck

The design needs to sure that Alice and Bob actually own the cards in their deck. The server is keeping track of who owns which cards, as well as what decks the users have build. Say Alice wants to use her fire deck, she can get a timestamp of the deck from the server, that includes her ID.

$$Timestamp_{(Deck,\ A)} = Sig_{srv}(ID_A,\ hash(Deck),\ Time) \tag{4.1}$$

She can then send the timestamp to Bob, without having to reveal which cards it contains, and if the timestamp of a deck is valid only for a specific amount of time after it is issued, it would be ensured that Alice did atleast own everything in the deck within that time frame.

If Alice and Bob dont want to reveal their choice of deck before the other has chosen as well, they can use bit-commitment to commit to the timestmap from the server, and then wait for their opponent to choose a deck as well before they reveal their choice.

Shuffling the decks

Now that the decks have been chosen, they need to be shuffled. neither Alice nor Bob can be allowed to know order of either his or her deck, or which cards their opponents deck contain.

The mental poker protocol 3.4 could be used to shuffle the decks, then let the players deal cards to each other during the game. This would require that each player generates a private and public key pair to use for each game specifically, since mental poker requires that you share your private key after the game is over.

With a tweak to the mental poker algorithm, that use bit-commitment instead of public key encryption when sending the cards to Bob to be shuffled, the opponent can still shuffle the deck, and send us back cards when needed, or really just the index of the card he chooses next.

This way Bob still have a representation of the individual card, and Alice can reveal cards easily during the game, without Bob having to verify the moves in the end of the game. The order of actions would look like this:

1. Alice shuffles the deck on her side.
2. She then commits to each card $Card_n$ individually by generating two random strings and calculating a hash value $Hash(Card_n,\ R1_n,\ R2_n)$. A card commitment ($Cc$) then consists of the hash value, and the first random string, while a card commitment proof (Ccp) consists of the card and second random string.
$$Cc_n = \{Hash(Card_n,\ R1_n,\ R2_n), R1_n\} \tag{4.2}$$
$$Ccp_n = \{Card_n,\ R2_n\} \tag{4.3}$$
3. She then sends Bob a commitment deck, which consists of the card commitments in shuffled order.
$$CDeck = Cc_1,\ Cc_2,\ ...,\ Cc_n \tag{4.4}$$
4. Bob can now randomly pick a commitment for Alice to use next time she needs a card.
5. When Alice then needs to reveal the card to Bob, she then simply sends him the proof for that specific card, and Bob can verify that it was the right card.

This way Alice can use the commitment to represent the card whenever she makes a move with it, and Bob still have a representation of the card on his end, which will be useful when they start playing the game.

An other technique that could be used, would be to use the fair-coin-flip technique to build a random string, that Bob would use as a seed in a random algorithm to pick the order of Alice's cards. This would leave Bob with a bit less control of the order of the deck, and could even make it obsolete for Alice to shuffle the deck at all, since her participation to the shuffling of the deck would be to provide one half of the deck seed.

Accounting for mana

As mentioned in **??**, one last thing that needs to be hidden from Bob, is how much mana Alice has left. This can easily be done by simply not telling Bob. However, if Alice does not tell how much mana she has left, what keeps her from just pretending she has infinite mana?

Bob could simply check everything after the game has been played, and make sure that Alice did not use more mana than she is allowed to, but this is a lot of work, and if for some reason Alice and Bob does not agree on the result of a game, the server would have to validate the game, which means that the server have to work through the entire game just to check that neither Alice or Bob have cheated with their mana use.

A way to solve this would be through the use of a homomorphic encryption scheme, that would allow for Bob to keep track of Alice's mana throughout the game, and then send the record back to the server at the end of the game, in order to get a verification that Alice did not cheat.

This can be achieved by using the Paillier cryptosystem 3.2. The mana a player has in the beginning of his turn, can be calculated by anyone from other values that are not secret, so it is essentially her mana usage each turn Bob wants to keep track of. This can be achieved using the following scheme.

1. The server holds a private key, and the public key is known to both Alice and Bob.

2. When the server signs a deck for Alice, it makes sure to include a negative card cost chiphertext $c_{negCost_i} = E(-cost_i)$ for each card, that will be used in the card commitments as well.

3. At the beginning of Alice's turn t, Bob creates a chiphertext of her initial mana $cm^t_{initial} = E(initial_mana^t)$.

4. Whenever Alice plays a card, $card_i$, she sends $c_{negCost_i}$ to Bob along with the move.

5. Every time the mana changes during a turn, Bob calculates a new chiphertext of Alice's current mana $cm^{(t,j)} = E(mana_left)$ for that step j, either by use of $c_{negCost_i}$ or if it is publicly known values that affect the mana, he can create and add the values him self.

6. Alice creates a chiphertext for the negative of the mana she has left every time it changes $cm^{(t,j)}_{neg} = E(-mana_left)$, and sends it to bob.

7. Bob can then add the values for each step to a check value $ch_{t,j}$, so that if Bob and Alice agree, the result should be a chiphertext that decrypts to the number zero $D(ch_{t,j}) = 0$.

$$ch_{t,j} = ADD(cm^{(t,j)}_{neg}, cm^{(t,j)}) \tag{4.5}$$

8. Bob then multiplies each check value, using equation 3.3, with a random number r_j each, that he chooses, which should result in an other chiphertext for zero, and then adds it all together in one value for that turn.

$$ch_t = \sum_j \text{MULT}(ch_{(t,j)},\ r_j) \tag{4.6}$$

9. At the end of the game he adds all the checksums of each turn into one final checksum $ch = \sum_t ch_t$, and sends it to the server.

10. The server then decrypts ch, and can tells Bob if the value is zero or not.

The reason for the server to tell Bob if the value was zero or not, is that Bob can easily send a value that is not zero, and just claim Alice cheated, but if it is left up to Bob to decide if he wants a thorough verification of the game, he is less likely to cry wolf.

A way to give Alice a chance to verify that Bob does his calculations correctly, is to use a random seed for the random values, that is used whenever a chiphertext is created. This way Alice could preform the exact same calculations as Bob, and they could verify the checksums in the end.

This scheme takes a lot of work off the server, since most of the time the only thing it has to do is decrypt one value for Bob.

It would be possible for the checksums to collide if Alice cheats twice, and one checksum ends up being the negative value of the other, but with a large enough range of random values to choose from on Bob's end, this becomes extremely improbable.

Starting the game

Now that the decks have been chosen and shuffled, the game needs to get started. A starting player has to be selected, which can either be done by who challenged who, with a fair-coin-flip, or which player has the better record.

Before continuing the protocol, both players encrypts all their secrets with the servers public key, and sends their encrypted secrets to each other. This way Alice can always go to the server for help if Bob either disconnected and lost his data, or for validation if Bob is cheating.

Both Alice and Bob have to agree on the starting state of the game $GState_{start}$, just before the first player makes his first move. This allows for saving a snapshot of the starting state $Snapshot(GState_{start})$, and use it to re-create the game if something should go wrong.

An easy way to achieve this is to have both Alice and Bob sign fingerprint of the starting state with their private keys, that can serve as proof that they have accepted the starting state of the game.

- Alice and Bob creates the starting state of the game together.
- Alice and Bob individually computes a fingerprint $Hash(Snapshot(GState_{start}))$.
- Alice signs her fingerprint and sends it to Bob.

- Bob verifies that the hash value matches the one he calculated.
- Bob sends his signed fingerprint back to Alice.

Only information that are available to both parties should be included in the snapshot, since it is important for the hash value to be the same for both Alice and Bob, so in order to include the decks, only the card commitments will be included in the snapshot. Any value that have importance for how the game will be played out should be included, so that it can not be changed without being detected.

Both parties now have a proof from their opponent, and the game can begin.

Playing the game

Lets say Alice has been chosen as the beginning player. As covered in Chapter 2, the game is played with turns and moves, and Alice has the first turn.

Making a move

The first thing Alice will need to do is to draw her cards, and this is done by sending a move to Bob that tells him that she wishes to begin her turn. Bob then sends back the index of the cards he picks for her, based on how that move should be executed, as dictated by the game rules.

Alice can then proceed to play a card, which as anything else, is done by sending a move to Bob. When Alice plays a card from her hand, she should reveal the card by sending the card commitment proof along with the move, unless she plays a trap card.

Most of the other moves that can be made during a turn are not complicated, since the simply execute commands based on data already known to both players. The last move that should be sent in a turn is the end turn move.

Documenting game progress In case one player looses connection to the other, or if Bob simply leaves the game because he is about to loose, it should be possible to re-create the game to the point where Bob left, however, Alice should not be able to tamper with the game just because Bob lost connection.

All that is needed to recreate a game is the starting state of the game, and the moves made by each player up until the current state.

Both players have already signed a fingerprint of the starting state, which serves as proof that the stating sate have been validated by both players, but something that proves that both players approve of the moves is needed as well.

An easy solution is to keep signing a fingerprint of the game state after each move, but making a fingerprint of the entire game state after each move is a lot of work, so it could be contained to just fingerprinting after each turn, and then have both players sign the fingerprint.

An other way to do it, is to document the moves them self. Taking some inspiration from how the linking in the Section 3.6 works, linking the moves together, starting

with the first link being the fingerprint of the starting state, it is possible to make sure that the moves are presented in proper order, while reducing the work we need to do each move to simply make a new link based off of the previous link.

$$Link_n = Hash(Move_n,\ Link_{n-1}) \tag{4.7}$$

The only thing we need now is for both parties to sign the link, so that there is a proof of agreement. A weakness of only documenting the moves, is that you are not sure that both parties agree on the rest of the state of the game, but this should be ensured by if everyone executes every move as it should be according to the game rules, both Alice and Bob's game state should be the same if they execute the moves in the correct order.

Trap card

Since the trap card has to stay hidden from your opponent, it should not be revealed on a play card move, but as soon as a trap card is in play, it has the ability to react to any possible move that can be made in the game. Bob does not want to tell Alice what his trap reacts to, which means Alice has to ask Bob if his trap card triggers whenever she makes a move.

When Bob finally reveals his trap card, it is also important that Alice checks that it should not have had been triggered before Bob claims it should.

If Bob did reveal his trap to late, Alice would need to be able to proof that he cheated. This proof can be made by requiring that Bob signs the link of the moves sent to him, along with a bit indicating if his trap triggers on that move or not. This way he cannot deny at which point he claims his trap should trigger.

Reporting to the server

At the end of the game, a winner has to be declared and proven to the server, so it can record it an reward the winner.

If Alice wins, and Bob is not a sore looser, Alice could get Bob to sign a fingerprint of the final state of the game, so that Alice can present the server with the final state of the game, which would show that Alice had won the game, or Bob can simply sign the game along with a message saying that Alice won.

But if Alice wins and Bob does not want to declare Alice the winner, Alice would have to prove to the server that she won fairly. If the game progress is documented by signing fingerprints of each move or turn, she would only have to send the moves needed to win the game along with the last signed game state, and the server could just execute the last few moves in order to confirm her victory. If only the linked moves were signed, the server would have to run everything from the beginning, in order to truly know if Alice did in fact win, since there are no later game state approved by both Alice and Bob, than the starting state.

Once Alice have reported her self as a winner of the game, it should not be possible for her to send the same game again and get double credit for it, so the server stores

the fingerprint of the starting state, until some time after the timestamps for the decks used in the game is no longer valid plus the time limit for a game. If Alice then submits a new game, the server checks that it does not have the same starting state fingerprint recorded already.

Time limits and connection issues

A thing that have not yet been discussed is time. A player should not be able to stall a game he is about to loose by simply not replying, in order for his opponent to loose patience and leave or forfeit the game. It would therefore be nice to have a time limit on taking a turn.

The problem is, that neither Alice nor Bob can be trusted to provide a valid timestamp, the only trusted party that can provide a trusted timestamp in our system is the server.

Alice and Bob could contact the server when the game is started, and provide it with the singed fingerprint of the starting game state, in order for the server to note the starting time of the game. This is simple enough, but it gets tricky when the game begins.

As mentioned the players sign a link of every move, which Alice could use to report to the server whenever she wants to document that she have made the move within the time limit, but what if Bob delays this process, either to try and make Alice go over the time limit, or because of a bad connection, Alice could just send the link and move to the server in order to prove that she has made her mind about the move, even though Bob is not responding. But what then if Alice never actually send the move to Bob, or simply ignored his reply, and is just trying to call him a cheater?

The server could take contact to Bob whenever Alice claims he is not responding, and show him the moves Alice sent. Bob could then reply to the server, and it could send the reply to Alice and back and forth.

The problem is that there is no proof that either Bob or Alice is causing the dispute, which forces them to resort to just trying to confront their opponent with their own last message and get a response from them.

It becomes clear that trying to keep track of the time in a game would end up requiring a lot from the server, and the idea of a game played without the server would quickly be lost if these time limits were to be completely enforced while people are trying to cheat.

Despite a lot of effort it have not been possible to find a solution to this problem, though a simple way to solve it would be to have the player, who claims that either contact have been lost or a time limit have been violated, send along the entire starting state and moves history when he contacts the server, and ask to transfer the game to a PBeM style game. This way Bob can not leave the game just to avoid defeat, since Alice can document the game progress, and has an encrypted representation of the secrets Bob need in order to continue the game, Bob would be forced to continue the game.

4.4 Summary

This section covered the design of a live game played out between two players, with minimum requirements to the servers involvement. A lot of problems were discussed and solved with the help of the techniques learned from Chapter 3.

The design manages to shuffle decks by using bit-commitment, keep track of the mana usage of the opponent without revealing the amount he has left, documenting the game progress with signed fingerprints, and reporting it to the server in the end.

CHAPTER 5
Play by email game

Now that the live game have been discussed, it is time look at what is possible when using the PBeM game protocol.

The main things that changes when when moving from a live game to the PBeM concept, is that all communication now goes through the server, and that Bob is not necessarily available when Alice makes her moves in her turn.

If the design should stay completely true to the concept of PBeM, the only contact Alice would have with the server is when she begins her turn, fetching the latest data and moves made by Bob, and when she ends her turn, sending the moves she made back. It will however be necessary to contact the server more often than that, if the game is to be kept completely secure.

There are some of techniques used in the design of the live game without a server, that can be re-uses for the design of the PBeM game, but there are also a lot of things that will have to be changed.

5.1 Game creation

Creating the game is a lot easier this time, since the creation of the game can take place on the server. The decks does not have to be verified since the server is the one keeping record of the decks, and the server can shuffle them as soon as the players have agreed on stating a game.

The game creation process quickly narrows down to very little communication between the parties when the server is responsible for creating the game.

1. Alice contacts the server, saying she wants to challenge Bob, and what deck she wants to use in the game.
2. The server lets Bob know that Alice has challenged him, but not necessarily which deck she wants to use.
3. Bob then accepts the challenge and chooses his deck.
4. The server then shuffles both decks, and let either Bob or Alice begin the game.

The players could submit part of a random seed, that would be used to generate the random values needed in the game, in order to let them have a provably fair game, but since the server is trusted, letting the server handle the random values is much easier for now.

5.2 Playing the game

When the game actually starts, if the design have to stay true to the PBeM concept, there will be some things that get complicated very quickly. The players need to be able to draw cards, use random values, and trigger traps. These three main issues can be hard to solve, if to contact the server is not allowed until the end of a players turn.

Without true PBeM

If the design is able to take contact to the server whenever it is needed, designing a system that would ensure that the players can not cheat is easy.

If Alice contact the server for each move she makes, the server could send a random seed back for each move, that would be used for any random events that would occur during that move, as well as for drawing a card from her deck. This way Alice can draw a card from her deck if needed, and she can not predict the card or any random values used in the game.

If the opponent have a trap in play, Alice can then execute the move, only to record every event that the opponents trap could trigger on, and send the events to the server. The server checks if the trap triggers, and reveals it to Alice if needed. Alice then executes the move fully, so that the traps effect are included.

Staying true to PBeM

This is where it gets hard, if Alice is supposed to be able to draw cards, use random values, and trigger her opponents trap, she needs to be able to access the information if needed.

Random values

Using random values becomes a problem, the server could send down a random seed for each turn, but Alice would be able to predict what happens if she makes a specific move, and even make an other move that uses random values, in order to get a different outcome for an other moves random values. The server could give a random seed to each card, that would be used whenever that specific card needs to use a random value, that way she can at least not change the outcome of a specific cards move that easily, but she would still be able to predict it, there is no way to prevent this without somehow committing to a move that requires a random value, before receiving the random value, which requires contact to the server.

Drawing a card

If Alice needs to draw a new card on her own, it needs to be ensured that she can at least not choose which card to draw her self. This can again be done with a random

seed that would be used to draw cards from her deck. There are two ways to do this, and each way has a downside to it.

You could use the same random seed for Alice to draw cards with throughout the entire game, which would allow her to know what cards she draws next turn as well. The other way is to use a new seed for drawing cards each turn, but if she for an example has a card on her hand that lets her draw three new cards, she can see which ones she would get this turn, and can decide to wait till next turn in order to get a different selection of cards. Again there is no way to solve this without increased server contact.

Triggering a trap

It should not be possible for Alice to know what the trap card is, as well as the script that dictates the traps abilities, but she need it to be revealed to her if she makes a move that triggers it. Alice can not be trusted, and does not report to anyone who can. It is essentially a paradox, Alice should not posses this information know, but she needs to.

There is no way for us to completely hide the trap from Alice, memory encryption could be utilized, but she would need to know when she triggers the trap, and be able to find the decryption key at some point if she ends up triggering the trap.

This is one of the cases, where the only solution is to attempt to make it hard for her to get to the information. The application can be obfuscated, the trap can be encrypted and the key hidden somewhere in the code or in memory, but the application would still show her the trap if she triggers it, and she can simply backup the state of the application, or her entire computer, before she triggers it. Then after she has triggered it, she restores the previous state, and makes a different move instead.

Obfuscation of the code would also help making it harder to find the random seeds used for drawing cards and other things, but it is not an actual solution as much as a minor annoyance for the potential cheaters.

5.3 Ending a game

Since the server already knows everything about the game, reporting a winner is easy. Alice makes the final moves that makes her win the game and reports it to the server. The server can verify that the moves did indeed result in Alice winning the game, and record her as the winner immediately, while sending Bob the sad news.

However if the server is to verify the winner through the moves, it has to run through every move made in the game in order to get to the final state. This can be solved by having the users sign fingerprints of the entire game state whenever the end a turn, as discussed in Section 4.3, then Alice can send the game state that matches the last fingerprint Bob singed to the server, along with the moves she claims leads her to victory. This way the server only need to run the moves of the last turn in order to verify Alice's claim of victory.

5.4 Summary

This chapter discusses why some anti-cheating features, specifically keeping secrets and making randomness unforeseeable, becomes harder to implement when the contact to the server or the opponent limited.

CHAPTER 6
Single player game

In a single player game, Alice plays without an actual living opponent, meaning she plays against an AI, an algorithm designed to complete tasks that normally require human interaction, like making decisions, or playing a game.

Again, the easy way to ensure that Alice can not chat, is by handing the game over to the server. She can play by the protocol of either a live or PBeM game, and the AI would be controlled by the server, making the moves needed. A lot of the other security measures could even be cut off the protocol, since Alice's opponent is now a trusted party. But again, that is a lot of work for the server, so it would be desirable to find a better way.

Alice can not be trusted to create the entire game by her self, as she could easily create a starting state that put her at an advantage. Furthermore letting her control the AI is risky, since she could have it do whatever she wants it to, unless we have a way to prove that she have acted by the rules of the AI.

6.1 Server seeded

A way to take some of the control away from Alice, is by contacting the server in order to get the initial state of the game. The server then notes the time of when Alice requested a single player game, and sends her the AI's deck, as well as random seeds needed to shuffle decks and preform other random actions in the game.

The major flaw here is that the same problem exists here as with the trap card in the PBeM game, Alice needs to know how to play the rest of the game, but that means she can predict random events, as well as see what cards the AI has, and predict what moves it is going to make.

Anything that can be done in order to decrease the amount things Alice can predict and control, will increase the contact to the server. A solution could be to do as in the PBeM design, and contact the server after each turn, getting random seeds needed for the game. If the AI were designed with some randomness in order to be less predictable, a seed could be given for its turn as well. The process would look something like this.

1. Alice gets the starting state for the server, along with the random seeds needed for the first turn.
2. Alice then contacts the server after each turn, reporting her moves so far, along with the AI's moves.

3. The server sends Alice random seeds for the AI's turn, and Alice's next turn. (The same seed could be used for both.)

4. It keeps going until the game is over, and Alice reports that it is over.

Alice might loose, and not turn in the result of the game to the server, but a time limit for how long the server wants to wait for Alice could simply be introduced, if Alice does not report back within the time limit, the AI would win the game.

When the game is over, Alice would have to report it to the server in order for the game to be verified, and for Alice to be rewarded if she won the game. But if the server has to verify the game, it has to run through everything, which would result in the same amount of work as just playing against the server.

6.2 Distributed verification

Running the entire game through from the beginning is a lot of work, specially because the scripts used to control the cards in the game execute a bit slowly, compared to pre-compiled code.

The system can be designed a bit smarter here, by asking the users to verify each others games instead, by comparing the ending state fingerprint, as well as the games winner. Whenever Alice sends in the result of her game, the server sends back two to five games for her to verify, which she does, and sends the fingerprints and winners she finds back, or an error message if the game is not over.

The server then sends Alice's game to the next people that turns in a game, and compare their results to Alice's. When Alice's game have been verified, the game is recorded, and she gets her reward.

If someone does not get the same result on a game as the other people, the server can send it to a few more people, until at least three people agrees on a result, or if people keeps getting different results, the server can decide to just run the game through by it self, and use the servers own result for the game.

This requires a bit more of the network connection, but much less of the servers computing powers.

Distributed cheating

It is possible for a group of people to cheat a distributed verification system, like the one described here, if they get together and decide to contact the server to get a game, then after a few moves declare them self as winner, even if the game have not yet ended. They share with each other what information they sent to the server, and if they are asked to verify each others games, they would simply send the same information.

The faster they could send in fake games, the more likely are they to get each others games to verify, since they flood the system with their games.

Other users would some times be asked to verify one of their games, and send an error message to the server, but if two other cheaters get to verify the game, they would overrule that users error, and get the game marked as verified anyway.

Taking samples

A way to attempt to detect distributed cheating, would be to have the server take samples, by picking games at random and verifying them itself. If the server manages to to catch a game that have been falsely verified, it could punish the players that verified the game, and put their other games and verifications into question.

This can be done at runtime, or if the games are archived, they could be looked through later, e.g. by copying the archive to a developers computer, and have the developer check the games. This means that you would not have to take samples until you actually suspect cheating in the system.

6.3 Summary

This chapter covered the further complications of handing over more control to the user, but proposed a system where the server seeds the initial game, and makes the users verify each others games, in order to take the workload off the server.

CHAPTER 7

Discussion

Throughout the thesis a lot of different options in terms of securing the game have been presented. Some solutions have benefits or drawbacks that might make them a better choice than others to use in an actual implementation.

This chapter will discuss what is actually practical to use in real life, as well as what choices have actually been made during the implementation of MLW, and why.

7.1 Reflection on homomorphic encryption

The work done on mana in Section 4.3 with homomorphic encryption is fairly simple, compared to what could be achieved with a fully homomorphic encryption system. With further work and investigation it might be possible to solve other problems, like hiding the trap in a PBeM game, or the logics of the AI in a single player game.

It might also be possible to utilize other cryptosystems than the Paillier cryptosystem, that has other properties, like the ability to preform XOR operations, in order to solve other problems like hiding random values.

A property that would have been desirable in the Paillier cryptosystem, would be if the chiphertext had a function, that would indicate if the message was a negative number, that way Bob would be able to verify cheating him self, and only need to contact the server in order to prove it.

Further work could be focused on homomorphic encryption, and the possible opportunities for application of these in order to enable further security and cheat detection in games and other applications.

7.2 Resource priorities

Some of the methods used in the design for making sure that players can not cheat are quite complicated, and would take some time to implement in an actual system.

Some things, like setting up a distributed cheat verification system, can take a lot of development time, and the cost of the development can quickly exceed the cost of simply paying for hardware that can handle the verification process, depending on how many requests the system gets over time.

Likewise if you have a small development team, and are trying to meet a deadline, the time it takes to implement and test a method that handles a certain issue, compared to the severity and how likely it is for an attacker to find and exploit the

issue, can force you to postpone the implementation of the security features, in order to spend the time on implementing other features of higher priority.

The resource priorities of the MLW project have been focused heavily on actually getting the game and the surrounding features up and running, which means that much of the current implementation lacks some security, but due to this thesis a solution for most imaginable security issues is now available, if they should become an issue.

7.3 Bugs and errors

Even though the design of an application may be secure in theory, the implementation of an application usually has a few bugs or errors, that breaks the security of the application. This is often cause for cheating or exploitation in both games as well as other applications.

This is something that can be hard to protect against, and will often have to be handled as part of maintaining the application, which makes it a good idea to make sure that you can force the users to use the newest version of the application.

Of cause, you can set up software tests, preform static analysis, or use other methods in attempt to secure your application, but this does not secure you against all bugs and errors.

CHAPTER 8

Conclusion

This thesis has described methods, that can help secure a system against cheating, allow people to keep secrets from each other, and help secure a system against predictability.

A design for securing three different game types has been proposed, with the use of fingerprinting, bit-commitment, public-private key encryption, homomorphic encryption, hash chaining and time stamping.

In the design of the live game a proposal is set up for a protocol that would allow two players to play a game of Major League Wizardry without a server, while still enforcing every rule of the game and documenting everything in a way that allows for reporting of a cheater to the server, except for the requirements for response time.

In the design of the play-by-email game the problems of keeping secrets from the user without server contact were outlined, and a few proposals on how to overcome these problems were made, though not many of these proposals led to a satisfying solution.

In the single player game a design was proposed where server seeds the initial game state, letting the user play the game on his own device, then submitting it for validation. The design also suggested a distributed validation system, that was meant to minimize the server workload of validating the games.

The result of this thesis serves as a description, of what problems can be encountered when attempting to prevent cheating in a turn based game with secrets, and how these problems can be handled. Even though the resources and priorities involved in creating something like a game do not usually allow for implementation of such designs, the methods used could have applications elsewhere.

Bibliography

[1] Game Made Studio ApS. game made studio. `http://www.gamemadestudio.com/`, 2014.

[2] Kristoffer la Cour Casper Sloth Paulsen, Phillip Barth. Major league wizardry - software technology project, 2013.

[3] Erickson, Jon. *Hacking: The art of exploitation*. William Pollock, 2008.

[4] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[5] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in cryptology—EUROCRYPT'99*, pages 223–238. Springer, 1999.

[6] Schneier, Bruce. *Applied Cryptography*. Wiley, 1996.

[7] Game Made Studio. Major league wizardry. `http://www.mlwcards.com/`, 2014.

[8] S. Turner and T. Polk. Prohibiting Secure Sockets Layer (SSL) Version 2.0. RFC 6176 (Proposed Standard), March 2011.

[9] Wikipedia. Play-by-mail game — wikipedia, the free encyclopedia, 2014. [Online; accessed 26-February-2014].

[10] Wikipedia. Public-key infrastructure — wikipedia, the free encyclopedia, 2014. [Online; accessed 26-February-2014].

[11] Wikipedia. Serialization — wikipedia, the free encyclopedia, 2014. [Online; accessed 26-February-2014].