Secure Distributed Poker using MPC

Christian Bobach, 20104256

Master's Thesis, Computer Science
April 2017
Advisor: Claudio Orlandi

AARHUS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

TODO: Abstract in english . . .

Resumé

TODO: resume in Danish…

Acknowledgments

TODO: Roberto for the idea
TODO: Cludio for sparing
TODO: Gert for fisher-yates
TODO: Kasper for fisher-yates and sorting network idea
TODO: Ni for help with compiler

Christian Bobach,
Aarhus, April 26, 2017.

Contents

Chapter 1

Introduction

In this thesis I have made a practical study of the application of Multi Party Computation(MPC) protocol. To show what and how MPC's can be used a poker game has been developed as a show case.

It is easy to think of how one could be cheated when playing an online poker game. It is hard for one player to know if the dealer and one of the other players has an agreement such that the dealer deals better cards to one player than the others. The idea of using a MPC protocol here is that as a player of online poker you would like to have a guarantee that the card are dealt fairly.

To ensure that the card are dealt fairly I will use a MPC protocol to take care of the shuffling of the cards. In this study I will use a two party computation(2PC) protocol. Therefore only a two party heads up poker game will be possible. The study is a showcase of the possibilities of MPC protocols and what can be achieved by them. It should be possible to easy extend the work done in this thesis to work in cases with more that only two parties using a protocol designed for that purpose.

For the poker game to work I have studied various fields both inside of computer science and outside. I have read up on different types of poker games to figure out which one was best suited for a two party setting. I have studied the underlying MPC protocol to understand how it works and to ensure that it for fills the right properties needed for an application as a poker game. I have studied different permutation algorithms and implemented them to compare them and see what effects they have on the underlying protocol.

TODO: introduce chapter 1: introduction to project

In chapter 2 on shuffle algorithms I introduce the different algorithms studied during the project. I argue for the ideas behind the algorithm and why they work in the application of a poker game. Some optimizations that can be done to the algorithm to reduce their size are perposed. At last the algorithms will be compared on a teoretical level to see different benefits.

TODO: introduce chapter 2: shuffle algorithms

TODO: introduce chapter 3: protocol and security

TODO: introduce chapter 4: implementation specif details

TODO: introduce chapter 5: conclusion and proposals of futher studies

In the next section the variant of poker game that will be used in this thesis will be introduced and others will be mentioned to give an idea of their differences.

1.1 The Poker Game

A poker game is a card game played in various rounds where the player draw cards and place bets. The bets are won according to a predefined list where the card constellation with the lowest probability wins the round. There exists many different variants of poker. The variant chosen to use in this thesis is the five card draw variant and the game is played between two parties. In this variant five cards are dealt to each player. Then the a betting round occurs. After the betting round the players have the possibility to chose between zero to five cards to change to try to improve their hand. Then the last betting round is performed before the cards is revealed and a winner is found.

Five card draw poker is played with a deck of 52 cards where at most 20 cards is used per round. This poses some requirements for our shuffling algorithms. Since there are 52 cards in the deck which yields 52! different permutations. We require an algorithm that can produce all these permutations to represent all the possible full permutations of the card deck. Because only the first 20 card of the deck is needed it is enough for the algorithm to produce a complete shuffle of these card and not the remaining 32 cards. This implies that the algorithm used to shuffle the cards needs to produce

$$\frac{52!}{(52-20)!}$$

different permutations.

Other variants of poker will have different optimizations. For one example if tree players was used instead of two, 30 cards of the complete deck would be needed. An other example could be the Texas Hold'em variant which is played by dealing two cards to each player and placing tree cards face upwards on the table. These cards are the used as a part of each of the players hand. After this a betting round which is continued by another card dealt facing upwards on the table. This is done twice before the final revelation phase where the winner is found. If the game involves two players then 4 card is dealt to the players and 5 to the table resulting in a total of 9 cards is dealt. This implies an algorithm producing

$$\frac{52!}{(52-9)!}$$

different permutations of the card deck is needed.

From here on and on wards when talking about a poker game the five card draw poker will be the reference otherwise it will be specifically mentioned. This is especially interesting when looking for optimizations on the shuffle algorithms and when they are compared. When coming to the protocol used by the protocol the way the cards are dealt will be the interesting part.

Chapter 2

Shuffling Algorithms

In this chapter I will introduce the different shuffling algorithms studied during this project. I will introduce the ideas behind each algorithm studied and what makes it special. I will introduce the idea behind why these were chosen. I will explain how they were optimized to fit better to the specific needs for an application as a poker game. Lastly I will compare the algorithms to see the different benefits.

All the permutation algorithms studied are with the purpose of shuffling card decks. It is important to chose an algorithm that ensures that the correct amount of permutations is reached.

The first algorithm studied is the Fisher-Yates algorithm introduced in [B3]. It may also be known as Knuth shuffle and was introduced to computer science by R. Durstenfeld in [A1] as algorithm 235. This algorithm uses an in place permutation approach and gives a perfect uniform random permutation.

The second algorithm proposed uses ideas from shuffling networks which is introduced in [A2] and the well known *bubble-sort* algorithm. The idea is simple and use conditional swaps which swaps two inputs based on some condition. Tjis also yields a perfect uniform permutation. An other type of shuffling networks called Bitonic shuffle network will be introduced and discussed. But no implementation of such a shuffle network was done.

All these shuffle algorithms is optimized to fit to the poker game introduced in the section on poker in chapter 1. This is done such that it only shuffles the first cards that are used during a game and not the whole deck.

2.1 Fisher-Yates

$Fisher\text{-}Yates$ is a well known in place permutation algorithm that given two arrays as input one of some values that should be shuffled and another such that the first array is shuffle accordingly to the values of the second array. These swap values of the second array indicate where each of the original values should go in the swapp. When the algorithm runs through the first array which is supposed to be permuted it swaps the value at an given index whit the value specified by the swap value of the second array. Think of the input to be shuffled as a card deck then you take the top card of the deck and swap it with another card at

Algorithm 1 ***Fisher-Yates***

deck is initialised to hold n cards c.

seeds is initialised to hold n random r values where $r_i \in [i, n]$ for $i \in [1, n]$.

```
function Swap(card1, card2)
    tmp = card1
    card1 = card2
    card2 = tmp
end function

function Shuffle(deck, seeds)
    for i=1 to n do
        r = seeds[i]
        Swap(deck[i], deck[r])
    end for
end function
```

a position predefined by the swap value.

This implies that the algorithm takes two inputs of the same size where the one is holding the values to be permuted, denoted *deck* with n c_i values, for $i = 0, \ldots, n$. The other holding the values for which the different c_i in the *deck* is to be swapped, denoted the *seeds* with n r_i values. If the swap values r_i from the *seeds* are not given in the correct interval the probability for the different permutations is not equally likely. Therefore it is important that the r_i values are chosen accordingly to the algorithm. The algorithm states that r_i is chosen from an interval starting with its own index i to the size of the *deck* which is n. This gives exactly the number of permutations required as the first card of *deck* denoted c_1 has exactly n possible places to go. c_2 has one less possible places to go an so forth until the algorithm reaches the last card c_n which has no other place to go. Since $r_i \in [i, n]$ we have $n!$ because i runs from 1 to n which should be the case as described in chapter 1.

If the values contained in *seeds* is not chosen for the right interval but instead chosen on all r_i from 0 to n we would end up having a skew on probability of the different permutations. As r_i in this case has n possible places to go yields n^n distinct possible sequences of swaps. This introduces an error into the algorithm as there are only $n!$ and n^n cannot be divisible by $n!$ for $n > 2$. Resulting in a non uniform probability for the different permutations. The same is the problem if r_i is not chosen from $[i, n]$ but instead $]i, n]$ such that the own index is not in the interval. By introducing this error to the algorithm the empty shuffle is not possible. In other words it is not possible to get the same output as the input. Which does not give the desired uniform distribution of permutations.

As described in the section on poker in chapter 1 we only need a permutation of the first 20 cards. Which means that we only need the $\frac{52!}{32!}$ specific permutations out of the total of 52! different permutations. Doing a m out of n permutation using the *Fisher-Yates* algorithm is straight forward. Instead

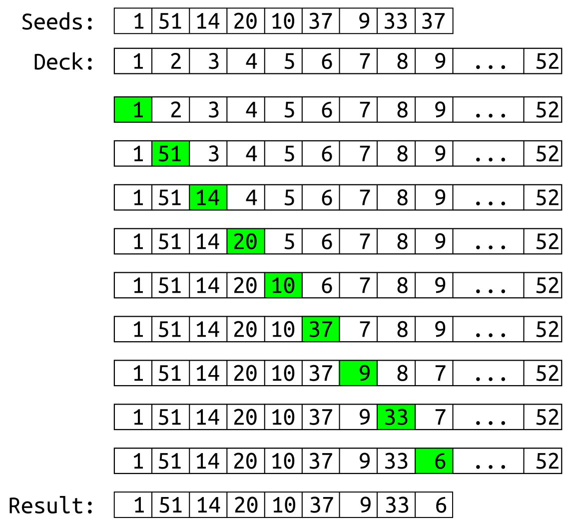

Figure 2.1: Fisher-Yates algorithm in action: In this figure a 9 out of 52 shuffle has been completed to ilustrade how the algorithm works. First 1 is swpaed with 1. Then 2 is swaped with 51. 3 with 14. 4 with 20 and so on until the first 9 numbers has completed a full permutation. Resulting in 1, 51, 14, 20, 10, 37, 9, 33, 6.

---

of running through n swaps indicated by the size of *seeds* it is enough to run through m swaps. Resulting in the input *seeds* only need to have size 20 and therefore a for-loop running fewer rounds. Those giving us a full permutation on the first m indexes of *deck*.

2.2 Shuffle Networks

Shuffling networks or permutation networks has a lot of resemblance to sorting networks. The idea behind this type of networks is that they consist of a number of input wires and equally many output wires. These wires go straight through the entire network. On each pair of wires there is placed a conditional swap gate. Such that if the condition of the gate is satisfied the input on the two wires are swapped. By placing these swap gates correctly on the input wires it is possible to get a complete uniform random permutation of the input on the output wires.

Applying such a shuffle network in the setting of a poker game is simple. The input to the shuffle algorithm is the *deck* that we want to shuffle and the output is the shuffled *deck*. The more interesting part is how to place the swap gates to ensure that the right number of possible permutations is satisfied. There are many different shuffle algorithms that can be implemented using shuffle networks. The first and only I have looked into and implemeted builds on ideas from [A2] where they introduces conditional swap. The algorithm is a combination of the well known *bubble-sort* algorithm and the conditional swap.

Algorithm 2 ***Conditional swap***

deck is initialised to hold n cards c.

seeds is initialised to hold $\frac{n^2}{2}$ random *bit* values where $bit_i \in [0, 1]$ for $i \in [1, \frac{n^2}{2}]$.

```
function ConditionalSwap(bit, card1, card2)
    if bit equal 1 then
        tmp = card1
        card1 = card2
        card2 = tmp
    end if
end function

function Shuffle(deck, seeds)
    index = 0
    for i=1 to n do
        for j=n-1 to i do
            index = index + 1
            bit = seeds[index]
            ConditionalSwap(bit, deck[j], deck[j + 1])
        end for
    end for
end function
```

Conditional Swap: The conditional swap algorithm takes ones again two inputs where the first input is an array, here a *deck* of n cards c_i for $i = 1, \ldots, n$. The second input is differint from the other algorithm. This time an array *seeds* of size $l = \frac{n^2}{2}$ bits b_j where $j = 1, \ldots, l$. The algorithm creates $n - 1$ layers of conditional swap gates where each layer is decreasing by one in size. Each layer is constructed such that each swap gate overlap with one input wire. Then each layer is made such that the first layer contains $n - 1$ swap gates. The second layer $n - 2$ and so on to the last layer containing only one gate. The layers are stacked in a way such that the first input wire is only represented in the first layer. This resembles what is done in the $Fisher$-$Yates$ algorithm, where output c_1 is determined by the first round in the for-loop. It can be seen in such a way that the first input c_1 has n places to go. The second layer determines which output c_2 will have. Continuing this way until reaching the last layer and the two last outputs c_{n-1} and c_n is determined. Resulting in a shuffle algorithm with a perfect shuffle and $n!$ different permutations as wanted.

This algorithm requires much more randomness as input compared to the $Fisher$-$Yates$ algorithm. But this algorithm does not have the same problems of how the randomness should be chosen. This algorithm uses one bit of randomness at a time and therefore do not suffer from the problem of choosing randomness outside of the correct interval. Therefore this algorithm is more robust in terms of the input seeds. But if the inner for-loop is not running fewer and fewer rounds it will suffer the same problems as encountered by $Fisher$-$Yates$ because it will produce n^n distinct possible sequences of swaps which is not compatible with the $n!$ possible permutations. This resulting in a

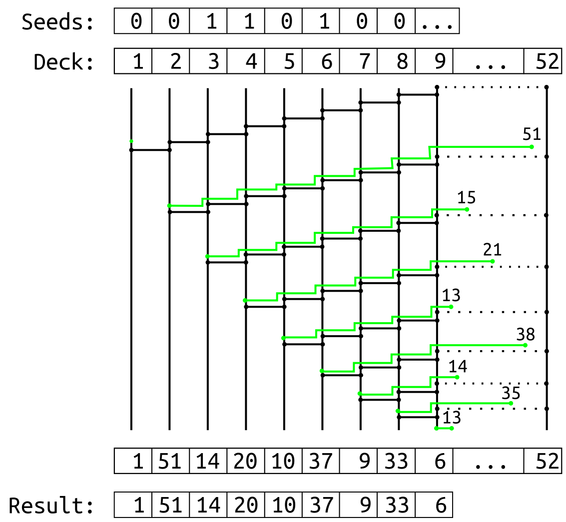

Figure 2.2: Conditional swap algorithm in action: In this figure a 9 out of 52 shuffle has been completed to ilustrade how the algorithm works. Each bit in the *seeds* indicate if a gate should be swapped. Since the size of *seeds* is so big I have tried to ilustrate which wire each value is located at before moved in a layer resulting in 1, 51, 14, 20, 10, 37, 9, 33, 6.

---

skew of the probability of the different permutations such that this is no longer uniform.

Once again it is possible to do some optimization to the algorithm since we do not need a complete permutation of the n inputs, but it is enough to only have m out of n. This can be done as in the case for the $Fisher\text{-}Yates$ where we let the outer loop run for m iterations and then we are done. This yields n possible values for c_1, $n-1$ possible values for c_2 and so one until $n-m$ values for c_{n-m}. This is exactly the amount of permutation we require for our algorithm as this gives us $\frac{n!}{(n-m)!}$.

2.3 Algorithm comparison

TODO: describe circuit

First of all it is important to understand that we cannot just plug the algorithm in to the MPC protocol. Since we use a MPC protocol that uses garbled circuits for the evaluation of the shuffle algorithm we need to provide a circuit that represent this algorithm. Therefor it is essential to understand how such circuits works and how to construct them. These circuits consist of different gates types. Special for the MPC protocol used here shuch type of circuit are all constructed of gates which has two input and one output wire. This results in 16 different gate types which can be used to construct a circuit. These 16 different gate types comes from the result of having two different values 0 and 1 for each of the wires. This gives 4 different ways to combine the two values which results

l	r	0	NOR	$\neg x$ AND y	$\neg x$	x AND $\neg y$	$\neg y$	XOR	NAND	AND	NXOR	y	If x Then y	x	If y Then x	OR	1
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Figure 2.3: A table of the 16 different gate types that can be used in a circuit of the type used in duplo

in 2^4 different combinations. Of these 16 different gates are some of the well known AND, OR and XOR. All the different gate types can be seen in figure.

Since circuits is a static representation of a given algorithm it grow fast in size and become rather complex. Therefor to make the construction of the circuits for the shuffle algorithms easier I have used a compiler called $Frigate$[1] to help with the circuit generation. The compiler takes a high-level program of a C like structure and translates it to a circuit of the correct format for the MPC protocol chosen to use. This compiler gives the possibility to easy implement the shuffle algorithm and generate a complex circuit that represent the right algorithm.

In the next section I will try to compare the to different algorithms and their circuit representation. The first to look for in the two algorithms is the amount of loops. In the $Fisher$-$Yates$ algorithm there is one for-loop and which makes a direct swap resulting in n total swaps. Where as in the *Conditional swap* algorithm we have both an outer for-loop and an inner for-loop. Which at creates $\frac{n^2}{2}$ calls to the conditional swap function. At first sight this seems to be many more calls to swap than for the $Fisher$-$Yates$ algorithm. But as mentioned earlier circuits are a static representation of an algorithm and therefore the two algorithms do not differ much from each other when comparing their circuit representation. First of all since circuits is a static representation of the algorithm the $Fisher$-$Yates$ algorithm can not do a swap based on an input to the algorithm unless all possible swaps are represented in the circuit. This resulting in the need for conditional swaps in the algorithm. Therefore in each of the rounds of the for-loop it should be possible for a value at a given index to go to all the indexes there or higher. This resulting in $n-1$ conditional swaps in the first round. $n-2$ conditional swaps in the second round and so forth. Since both algorithms now need to use conditional swaps they are easy to compare. We know that this gives exactly the same amount of conditional swaps for both algorithms, which is $(n-1)!$ if we do not take the optimizations into account. The biggest difference is how the two algorithms swap in the rounds of the outer for-loop. Here the $Fisher$-$Yates$ algorithm has conditional swaps from index i to all the indexes i' where $i < i'$. Where the *Conditional swap* algorithm has swaps from j to $j' = j + 1$ where j is running from i defined by the forloop to $n - 1$.

Another different aspect of the algorithms is to compare the input. Both algorithms takes a *deck* and *seeds* as input. Where the *deck* is the representa-

[1] I used version 2 which is linked to in appendix A. This version is optimized to construct circuits for the duplo protocol.

tion of the cards which is the same for both algorithms. But the input *seeds* differs a lot. This is because of how the two algorithms handle the swaps. Therefore this is the interesting part to look at. First of all it is important to remember what the goal is for this thesis. It is to create a secure distributed poker game. Therefore we use a MPC protocol that help us overcome the security part. We can therefore expect that one of the players will try to cheat. As the protocol takes *seeds* from both parties on how to shuffle the *deck* we need to handle this in some way. The easy way is just to XOR these *seeds* together to get a new seed to use in the shuffle algorithm. This is completely fine in the $Conditional$ $swap$ algorithm since it uses one bit of randomness at a time for each swap gate. We know that the XOR of to random bits yields a equally random bit. But for the $Fisher$-$Yates$ it makes the algorithm insecure since the XOR of the two *seeds* can result in a new seed that do not forfill the requirements to the inputs. This will give a bias that result in a higher probability of getting low cards. This is because the algorithm relies on the random r values of the seed to be in given intervals. Therefore when XOR the *seeds* from the two parties it can not be guarantee that the random r values for the shuffle is inside the correct interval. The easy solution to this will be to take the modulo reduction of r to fit inside the desired interval. Here I will come with an example to show the problems by doing so. In our case we need to represent 52 cards. These can be represented in base 2 using 6 bits since $\lceil log_2(52) \rceil = 6$. But as we know $2^6 = 64$. This implies that we have 11 possible values for our r in the first round after the *seeds* have been XOR'ed together. This result in the first 11 values from 0 to 10 to have the probability $\frac{2}{64}$ while the last 31 values from 11 to 51 have probability $\frac{1}{64}$ giving us a bias. Instead of using XOR to construct the new r values for the shuffle algorithm the *seeds* will be added 6 bits at a time. This resulting in a uniform propability on the r values. Here it is important to notice that while 6 bits is enough to hold the 52 card values it is not sufficient to hold the sum of such two values.

At last a short comment on other shuffle algorithmic possibilities. There do exist other types of sorting networks that could be used. In [A2] they also uses a algorithm known as the $Bitonic$ algorithm refering to the way the network is constructed. Such a network is constructed of what is known as $half-cleansers$ known from sorting networks. These $half-cleansers$ are constructed such that the input has one peak, $i_1 \leq \cdots \leq p \geq \cdots \geq i_n$. Then the output is half sorted such that the highest values are in one of the two halfs. This creates a circuit of size $O(n \cdot log(n))$ which is better then what the $Conditional$ $swap$ and $Fisher$-$Yates$ algorithms can aquire since it creates a circuit of size $O(n^2)$. But as argued earlier the $Conditional$ $swap$ and $Fisher$-$Yates$ algorithms are easily optimized to the setting studied in this thesis.

TODO: Bitonic algorithm, optimization? can maybe be done by flipping such a sorting network and removing half cleansers

As a result of this we get that for some card games it can be an idea to check if one algorithm can outperform another. In out case we need only 20 out of 52 cards. We now know that a $Bitonic$ algorithm would produce a smaller output but since the two algorithm studied here are easy to optimize such that they produce a relative small circuit. This implies that there will ba a cross over at some point where it is better to shuffle a complete deck than only parts of it even if only a part is needed. This could be in a setting when playing with more the two players for a game of poker.

Chapter 3

MPC-protocol and Security

TODO: What is needed of the protocol, active security, single wire opening, duplo
TODO: two party protocol
TODO: >two party protocol, maks deltager afh;ngig af poker version

TODO: Preprocessing of duplo, serversetting interresting to study in stead of application

TODO: Gameplay

TODO: Trusted server, state server?

Chapter 4

Implementation

The main idea of this chapter is to introduce the different stages during the implementation. Here I will introduce the different problems and the solutions I have chosen to overcome those problems.

TODO: Introduce different sections

4.1 Shuffle algorithm implementation

TODO: Describe the implementation of the algorithms
TODO: Describe problems
TODO: Describe solutions

TODO: Describe gadgets of the two shuffle protocols

Because the *DUPLO* protocol was chosen as the MPC protocol to use in the implementation the first hurdel was to make the circuits that should handle the shuffling of cards. Since circuits become rather complex when trying to implement functions it was important to have a compiler that could translate the algorithms into the desired circuit representation. Luckly there was a compiler for generating *DUPLO*-circuits that could be used. Therefor before staring to implemt the shuffle algorithms I read up on the documentation for the compiler[1]. The documentation is from the first version of the *Frigate* compiler which was extended to suit the *DUPLO* format. The documentation was not well specified and it was a long time since I last had worked with circuits in such a way. This resulted in some hard earned experience through trial and error on small exapmles.

The most important and different things to take into account here is the possibility of wirre acces and only one level functions. First of all it is possible to specify which and how many wires should represent a value like $y = x\{index : size\}$. This is the way bit inputs is translated into higher level representations. When it comes to functions there are some restrictions. First of all only one level of functions is allowed. Second of all only assignments in the main method

[1] Which can be found at github: `https://github.com/AarhusCrypto/DUPLO/tree/master/frigate`

is allowed through function calls. Otherwise the programing language used resembles C.

In the implementation of the two shuffle algorithms there are five different functions used. For the *Conditional swap* algorithm the first function used is the *xorSeed* which handles the XOR of the *seeds* recieved from the two parties. Which is straightforward. The next function used is the *initDeck* function. This functions inicilises the sorted deck wich is to be shuffled. This is done by a for-loop inserting the right values on the right wires. Here 6 bits is used for the representation of cards. It is importand to notice that the variable holding the start position of every card needs its own representation with at least 6 bits to hold the correct value. If only 6 bits were used only vires up to opsition 63 could be assigned t value. Therefor 9 bit is used for this variable. The 9 bits comes from the fact that $\lceil \log_2(52 \cdot 6) \rceil = 9$ bits. The last function used in the *Conditional swap* algorithm is the $shuffleDeck$ function. This is the function handling the actual shuffling. This is implemented using two for-loops one for the layers of the network and another for the swap gates in each layer.

In the case for the $Fisher$-$Yates$ the things stack up a bit differently. The first function in this case is the *correctSeed* function which takes the *seeds* from the two parties and correct them as descussed earlier. First a card representation of 6 bits is added from each of the parties. This can not be represented in 6 bits therefor a bigger representation is used. The idea is that after the addition a modulo reduction will be use to ensure that the new seed value is inside the right intervall. But because the modulo reduction implemented in $Frigate$ only supports a divisor of power of 2 modulo can not be used. Because the seed values calculated uses devisor different from these. I used some time to figure out that this was only supported since it is not statet anyware in the documentation. After experiencing this I tried to find some articles on how to implement a circuit for the modulo reduction. But during my search I found out that this is not a trivial task to do. Therefor another solution was chosen. Since the input *seeds* to the functionality is assumed to be in the right intervalls the solution was to subtract the boundray of the interval if the sum exeeded this. This was done by introducing an if statment. It is nothworthy to mention that all values have had an unsigned representation until now. Since the comparison of to values is needed a signed representation is introduced since the documentation stats that the comparison of two values only works on signed representations. This implemetations now ensures that the randomness used for the $shuffleDeck$ function has the right form. But only if the original inputs are inside the right intervalls.

The second function is the same as in the case for the *Conditional swap* algorithm which is the *initDeck* function. The last function called is the $shuffleDeck$ function which is different from the one from the *Conditional swap* algorithm. This function constist of an outer-loop that runs through the cards. Because circuits are static as disscused earlier it is not possible to assign a wire value based on a variable input. Therefor layers of conditional swaps are generated. This is represented by the inner-loop. In this way a gadget for each is constructed such that the index specified by the outer-loop can be swapped with any other value in the rest of the card deck.

4.1.1 Circuit generation

In this section I will introduce how the circuit compiler worked. Which problems I encountered and how some of them were fixed.

First of all the compiler needs to be installed which require some special versions of some librarys which are not the latest. This requires that some specific setup for the compiler is done. But folowing the instructions in the installation guide and some internet seach made the compiler run.

Then when a program has been written in the *wir* format specified by the documentation one compiles it to th *DUPLO*-format using the following command:

```
./built/release/Frigate path/to/file.wir -dp
```

This generates some different files where the one with the extention *.wir.GC_duplo* is the one that *DOPLO* can use as input.

The *DOPLO* framework has a function that allowes for evaluation of these inputfiles. This gives the posibility to specify the inputs from the two parties and get the result. This possibilty allowed me to study the implementations of the algorithms to see if thay did as I expected. First I tried to implement the *Fisher-Yates* algorithm and as described earlier this revealed some problems. Since I did not have earlier experience working with the *Frigate* compiler I was not sure if the problem was in my implementation or in the compiler. At first the focus was on my implementation but it was later discovered that it was the version of the *Frigate* compiler used to generate *DUPLO*-circuits. First I expected it was in my implemetation. Why I starte to break the implementaion down into smaller modules that could be tested. So I created a framework that allowed me to test the different modules one by one. After this it was clear that something were off when using the modulo reduction. After different attempts to get it to work without any luck the focus shifted from being on the implemetaion to be on the compiler. After breaking the implementation down and testing only the modulo operator % which is in the documentation for the compiler it could not be the case that I had made any error. Therefor I started to look into the compiler. The compiler I used was an extention of an existing one and the idea was that a bug could have been introduced when adding the new fature. Therefor the old version of *Frigate* was installed to test if this implemetation had the same bug. But the problem was that the old version did not support any circuit output format that *DUPLO* could read. Since *DUPLO* supports what is known as composed circuit format files and *bristol* formatted circuit files. I wrote a parser that took the output from the old version of *Frigate* and translated that to *bristol* format. This gave me two versions of the same program that I could run against each other and compare their results. This showed that there was a difference in the results produced.

When I looked deeper into the problem it came clear that not all gate types was implemeted an the modulo operator triggered one of these gates.

This resulted in a fix of the new version of the *Frigate* compiler and a complete change in the representation format of the circuits. Now all gates in the *DUPLO* circuit format has two input and one putput wire. The representation of gates chaged from a more human readable type like *XOR* to a truth table frendly type like 0110 for *XOR*. And now all 16 gate types from the truth table figure are implemented. The representation of the two constant wires **0** and **1** is handled as special cases since all gates now has two input wires. For the case of **0** it is handled as an *XOR* gate with two input wires with the same value. For the case of **1** it was handled as the *NXOR* of two wires with the same input value.

In this way I have contributed to the *DUPLO* project and helped secure a stronger research product.

On the other side the compiler has not been updated to support modulo with a deviser that is not the power of 2. As mentioned earlier the small amount of reserch I did into that area when I noticed the problem seems to indicate that this is not trivilay fixed. Therefor this is leaft unfixed.

TODO: Describe the parser

TODO: Technical comparison of the circuits, number of gates, XOR gates, NON-XOR gates (table)

4.2 Poker implementation

TODO: Describe the roles of the two parties
TODO: Describe the setting
TODO: Describe the different stages in the protocol and the sequense of the stages

TODO: why can only one hand be played at a time, broken bristol compiler ehn frigate v2 was fixed

TODO: Describe the more interresting setting, why is it interresting and why is it not implemented

Chapter 5

Conclusion

TODO: Conclusion bla ...

Appendix A

Codebase

TODO: Where is the codebase to be found
TODO: How to use the code base

TODO: The poker repo

TODO: The duplo and compiler repo

Primary Bibliography

[A1] Richard Durstenfeld. Algorithm 235: Random permutation. http://doi.acm.org/10.1145/364520.364540, July 1964.

[A2] J. Katz Y. Huang, D. Evans. Private set intersection: Are garbled circuits better than custom protocols? https://www.cs.virginia.edu/~evans/pubs/ndss2012/, 2012.

Secondary Bibliography

[B3] F. Yates R.A. Fisher. Statistical tables for biological, agricultural and medical research, 6th ed. http://hdl.handle.net/2440/10701, 1963.