Secure Distributed Poker using MPC

Christian Bobach, 20104256

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

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

Resumé

Acknowledgments

Christian Bobach,
Aarhus, May 22, 2017.

Contents

Chapter 1

Introduction

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

It is easy to think of how one could be cheated when playing an online game of poker. It is hard for me as a player to know if the dealer and one of the other players has an agrangement such that the dealer always deals better cards to that player such this player wins in the long run. The idea by using a MPC protocol here is to guarantee that the cards are dealt fairly. Such that the player of online poker can trust the protocol and know that the cards are guaranteed to be 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 called *DUPLO* which will be introduced in chapter 2. In this thesis a two party heads up poker game will be studied. 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 another MPC protocol designed for that purpose. It should also be equaly easy to extend the game to work with more players.

For the inplemetation of poker I have studied various fields both in computer science and other fields. 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 on DUPLO

TODO: introduce chapter on shuffle algorithms

In chapter 3 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 on Poker Implementation

TODO: introduce chapter on conclusion and proposals of futher studies

In the next section the variant of poker chosen for this study 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. There exists many different variants of poker but only one will be chosen. The variant chosen to use in this thesis is known as 'five card draw' poker. In this study the game will be played between two parties. In this variant of poker five cards are dealt to each player in the first round. After this the first betting round occurs. Then a swap round occurs where the players have the possibility to chose how many cards to change to try to improve their hand. Then a last betting round is performed before the cards is revealed and a winner is declared.

Five card draw poker is played with a deck of 52. This poses some requirements for our shuffling algorithms. Since there are 52 cards in the deck this yields 52! different permutations. We require a shuffle algorithm that can produce exact these permutations to represent all the possible shuffles of the card deck. Because only the first 20 card of the deck is needed per game 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 only needs to produce

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

different permutations. Since each player is dealt five card and at most can chane all these cards in the swap round. This yealds 10 card per player and therefore 20 in total.

Other variants of poker require a different amount of cards per game. One example could be if the came included tree players instead of two, then 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 is performed. This 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 used. This implies an algorithm producing

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

different permutations of the card deck is needed. This is also known as an m out of n permutation.

From here on when talking about a poker game the five card draw poker will be the reference otherwise it will be specified. This is especially interesting when looking for optimizations on the shuffle algorithms which will be introduced in chapter 3 and when they are compared. When coming to chapter 4 this will have effect when the cards are dealt. Both in terms of the amount of data sent and the time used by the protocol.

Chapter 2

DUPLO

In this chapter I will introduce the *DUPLO* framework introduced in [A2] and why this was chossen to handle the communication and security of the poker game. I will explain how the structure of the *DUPLO* protocol works and describe what the different framework calls handle. I will go over the secury details of the protocol to illustrate how this is guaranteed. Lastly I will introduce the *Frigate* compiler which is shiped whith the *DUPLO* framework to generate circuits for the evaluation.

As it can be read in [A2] the *DUPLO* framework is among the latest papers where the effency of a two party computation(2PC) protocol using garbled circuit in a malicious setting is studdied. In the paper *DUPLO* is claimed to reach the protocol performes better then any existing protocol. Their idea came from the fact that the two extreme variants of cut and chose protocols did not preform well in each end of the spectrum when it comes to the size of the circuits. As they cam up with a new approch to the way cut and chose 2PC protocols could be done in the malicious setting. The idea is to garble subcomponets of the circuit and get a optimum somwhare inbetween the two extremes; garbling of complete circuits or garbling on gate level, cut and chose. The aim was to show that the gate level cut and chose added an overhead when soldering these thogether again when the circuits for evaluation is build. At the same time to show when the number of subcomponents goes up there is a performance gain compared to whole gate cut and chose because of the amortized benefits.

As seen in section 7 on performance in [A2] it is clearly that the experiments done on real life circuits yealds an optimal cut and chose strategy which differs from the earlier known possibilities. The gain in terms of running time encreases as the size of the circuits get bigger. Which shows that the *DUPLO* protocol scales signicicantly better that the rest.

As this is the best performing 2PC protocol at the moment combined with the fact that it is developed at Aarhus University such that the people with knolaged of the protocol is in the same building was the main factors for using *DUPLO*. The fact that *DUPLO* supported the possibility of single and destinct wire openings to a desired party helped the decision.

In the next section I will introduce the different fundtions in the framework and what they achive.

2.1 The Framework

The *DUPLO* 2PC framework was chosen to use during the experiemnt of implemeting a poker game. *DUPLO* consist of two parties, a *Constructor* and an *Evaluator* with different roles during the protocol. The *Constructor* generates the garbled circuits and sends to them to the *Evaluator*. The *Evaluator* verifies a number of these circuits. If these pass the *Evaluator* trust that the remaining circuits are valid. Then these remaining circuits is used during evaluation.

The overall construction of the framework consists of different functions to call to allow for the right communication between the two partis. It is specified in wich order thes function should be called to ensure that the right information are at the parties at the right time. At the same time the spiltup of the functions allow for local computations to be done between these framework calls.

To run the protocol and use the famework a *Constructor* and an *Evaluator* is created. First of all they read the circuitfile specifying the functionality decired. In our case it is the shuffle algorithm which is introduced in chapter 3.

Once these are created they run the framework function calls in parrallel. First the two parties connect to each other via the *Connect* call. In this case it is the *Constructor* hosting the servise and then the *Evaluator* connects to this. When they are connected they each make a call to the *Setup* function to initialize the communication protocol which is the XOR-homomorphic commitment protocol. After this they start the preprocess phase of the componets in the circuit by running the $PreprocessComponentType$ function call. Which generates a garbles representation of the shuffle algorithm.

Then the $PrepareComponents$ function is called to produce input and output authentication such that the inputs and outputs can be transfered securly between the two parties.

After this the composed $Circuit$ is constructed by calling the *Build* function. This constructs the complete circuit which is to be evaluated later by the call to *Evaluate*. This ensures that the function componets specified by the composed circuit file is soldered together in the right way. Such that the ouput wires from one subfunction is feeded to the right input wire on another subfunction. This is also done using the XOR-homomorphic commitments. When the next call is made to *Evaluate* the input to the circuit is given and the circuit is evaluated on this. Such that a garbled output is generated. When this is done a call to $DecodeKeys$ can be made and the output of the circuit can be learned.

The evaluation of circuits in the *DUPLO* protocol allows for openings of outwires to both parties or only one. This will allow us to only reveal some cards to one player and other cards to the other. The split up of *Evaluate* and $DecodeKey$ functions allows for opening of output wires in different rounds which helps us achive good round complexity when doing our implementation.

2.2 Security

In this section I will introduce the security of the protocol to show that plyers playeing a game of poker with an implementation using the $DUPLO$ framework will have *oblivioness*, implying that the oponent can not learn more that supposed to. The players will also be ensured *correctnes* of the protocol, meaning that if garbled evaluation is done it gives the right output. $DUPLO$ also ensures *authenticity* because it is not possible for a player to doing evaluation of the functionality on other input the the party garbling the circuit.

The proff of security for the protocol is done using the Universal Composision(UC) framework. This is an easy digested abstract protocol proof technuiqe which allows for sequential predefined interaction between parties using actions and reactions. It has a modular approche to functionality proofs, when one functionality has been proved it can be used as a steppingstone for the next proof. In $DUPLO$ they use the hybrid model with ideal functionalities $\mathcal{F}_{HCOM}$ and $\mathcal{F}_{OT}$. Where the $\mathcal{F}_{HCOM}$ functionality is for the XOR-homomorphic commitment scheme used by the protocol, and $\mathcal{F}_{OT}$ for the one out of two oblivious transfer. These functions are then used to prove *correctness*, *obliviousness* and *authenticity* of the protocol.

TODO: Protocol overall security proof appendix a

In the section on protocol details in [A2] appendix A they describe and analyse the protocol. Here structioring the main protocol and going into details on how *correctness*, *obliviousness* and *authenticity* is guaranteed therough the different protocol function calls. The proof end up beeing rather complex as the main structur consist of 8 subfunctions, which each is a combination of futher subcircuits. All of these functions are guaranteed to satisfy the properties. During the analysis of these functions they end up with lemmas proving correctness of; soldering and evaluation of subcircuits. They end up with leammas proving robustness of; the key authenticator bucketes, evaluation of key authenticators, input of constructor, input of evaluator, evaluation of subcircuits and output of evaluator. This colminate in the theorem proving robustness of the protocol, showing that if the *constructor* is corrupt and the *evaluator* is honest and the protocol does not abort, then the protocol compleets holding the before mentioned property, except with negligible propability. As known when using MPC protocols where half or more of the parties are corrupt we can not guarantee termination.

TODO: Protocol security analysis appendix b

In appendix B they prove the fact that the protocol is secure agains a corrupt *constructor* or *evaluator*. Since it is a 2PC we may assume that one of the parties is honest as the partis trust in them selfs. When proving in the UC framework it is worth to remember that a poly-time simulator $\mathcal{S}$ should be presented. For the case of a corrupted *constructor*, $\mathbf{G}$ for generator, and a honest *Evaluator*, $\mathbf{E}$. $\mathcal{S}$ plays the role of $\mathbf{E}$ in the protocol, but is not given acces to the inputs of $\mathbf{E}$, $x_{\mathbf{E}}$. Instead $\mathcal{S}$ has access to an oracle $\mathcal{O}_{x_{\mathbf{E}}}(\cdot)$ containing $x_{\mathbf{E}}$. $\mathcal{S}$ might contact $\mathcal{O}_{\S_{\mathbf{E}}}$ giving input $x_{\mathbf{G}}$ and in return learn $y_{\mathbf{G}}$ as if the evaluation of the functionality was done with $x_{\mathbf{E}}$ and $x_{\mathbf{G}}$ as input.

To show that the protocol is secure in this setting we need to show that a *constructor* running the protocol can not distinguige between talking to $\mathbf{E}$ or $\mathcal{S}$. ???

TODO: What is needed of the protocol, active security, single wire opening, duplo

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

2.3 Frigate the *DUPLO* Circuit Compiler

In this section I will introduce how the new version of the *Frigate* circuit compiler workes.

First of all when installing the compiler some special versions of libarys are required, which are not the latest. Folowing the instructions in the installation guide and some amount of internet search I was able to get it up and running. During the thesis I have been using hardware running Ubuntu 16.04 LTS or higher. Where the standard version of *flex* and *bison* is higher that supported by *DUPLO*. This requires that the right versions are installed and keapt back such that these are nut updated later. But as the compilation of circuits are done once and prior to the compilation of the actual *DUPLO* implemetation this is not a complete deal breaker.

TODO: Describe the frigate programming language

The *Frigate* compiler came shipped with the documentation for the first version and was not updated when extended to fit the *DUPLO* framework. This should not be expected since the functionality of the compilers input language was not changed. But this resulted in some time consuming trial and error since it was not well written and specified. It was a long time since I last had worked with circuits in such a way as was the case for generating big circuits with a circuit compiler. This resulted in some hard earned experience on small exapmles.

Some of the most important and different things to take into account when programming for circuit generation is the possibility to do wirre access. First of all it is possible to specify which and how many wires should represent a value by using `y=x{index:size}`. This is the way long bit input strings are translated into higher level representations of smaler instances. This gave some occasions for frustrations because is it not possible to acces vires based on variable inputs which cannot be pre determined by *Frigate*. This makes perfect sence since the compiler can not know which wires should be used as the representation. A second point to remember when working with *Frigate* is that it do not allow for more than one level functions. Which gives some restrictions in programming compared to many other languages. This restriction makes it harder to create small functions with one single specific focus that could be called when the desired functionality was required. This did not give possibility recursive functions. But as the circuits generated is a static representation this is a restriction which cannot easily be handled, since the size of the circuit can

not varriate based on the inputs. Another small odidity is that *Frigate* only allows for assignments in the main method through function calls.

Otherwise the programing language used by *Frigate* resembles the well known *C* language. The compiler requires you to specify how many parties the functionality is used by, by the call to `#parties n`. At the same time it requires that the size of input is specified for each of the parties using `input i size_i` and `#output i size_o`. But other wise it allowed for definitions of constants, types, structures and imports just like in *C* which allows for some easy readability.

When the decired functionality has been implemented using the *wir* described above used by *Frigate* the compiler can be used, from inside the compiled *DUPLO* framework, by calling:

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

The `-db` flag ensures that the right *DUPLO* format is generated. This call to the compiler generates different files with different extentions. The file we are interested in is the file with the extention `.wir.GC_duplo` this is the one that the *DOPLO* framework can use as input file.

The *DOPLO* framework has a functionality that allowed for evaluation of these input files without setting up both parties. This gave the posibility to specify the inputs for the evaluation and learn the result in a farst return cycle. This possibilty allowed me to study the implementations of the programming language and how the algorithms behaved compared to expected while implementing them. First I tried to implement the *Fisher-Yates* algorithm, which is described in details in section 3.1 and can be seen as algorithm 1, in the `.wir` format. During the implemetation I encountered some problems. Since I did not have any earlier experience working with the *Frigate* compiler I was not sure if the problem was in the implementation of the algorithm or in the compiler. At first the focus was on the implementation as the lack of experience working with *Frigate* easily could lead to implementation errors. After a lot of modolaisation and debugging it was clear that something was wrong with the version of the *Frigate* compiler used to generate *DUPLO*-circuits. During the debugging process I created a framework that allowed me to test the different modules of the implementation one by one. It was then clear that something were off when using the modulo reduction in the *Fisher-Yates* algorithm. 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 the modulo operator `%`, as specified in the documentation, it was clear that some error was introduced during compilation. Therefor I started to take a deeper look at the compiler. Since the compiler was a modified version of the *Frigate* compiler the posibility was that a bug could have been introduced when adding the new *DUPLO* featurs. Therefor the old version of *Frigate* was installed to test if that implemetation had the same bug. Then a problem arose because the old version did not support the

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

Table 2.1: A table of the 16 different gate types that can be used in a circuit of the type used in duplo

---

DUPLO circuit format. But because *DUPLO* also has supports for another circuit file format known as *bristol*, I wrote a parser that took the output from the old version of *Frigate* and translated that into the*bristol* format. This gave me two different formatted circuit files that *DUPLO* could use as input files. Based on these two formats is created a test framework to see if there was any differences on the output when ran on the same input. This showed that there was a difference in the results produced, especailly in the case when using the modular reduction. When taking a deeper look at the problem it came clear that not all gate types was created during compilation using the *DUPLO* version of the *Frigate* compiler.

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 by standart and are not explicitly written in the formatted file as earlier. The representation of gates chaged from a more human readable type like *XOR* to a truth table frendly type like 0110 for *XOR*. This change in representation ensured that all 16 gate types which can be seen in table **??** are now implemented. The representation of the two constant wires **0** and **1** is handled as special cases as all gates now has two input wires. For the case of **0** it is handled as an *XOR* gate with input wires with the same value. For the case of **1** it is now handled as the *NXOR* of input wires with the same value.

In this way by going into details and debugging my implementaion of the algorithm I have contributed to the *DUPLO* project by reporting my findings and allowed them to fix this problem before publishing their finding. This has helped to secure a stronger overall 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. This is not mentioned anywhere in the documentation for *Frigate*. When is accounted the problem of the modulo operator I used small amount of reserch to see if this functionality easily could be implemented. During the search I noticed that the implementation of modulo a power of two is simple while research done seems to indicate that the other cases are not trivilay implemented. Therefor this was leaft unfixed and my implementation uses a hack to overcome this problem which is stated in section 3.1.

Chapter 3

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 why these were chosen. I will explain how they were optimized to fit better to the specific needs for a poker game. Lastly I will compare the algorithms to see the different benefits, and based on this choose which algorithm to use in the implementation.

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 [B4]. It may also be known as Knuth shuffle which 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. This algorithm is introduced in section 3.1 and can be seen in pseudocode as algorithm 1.

The second algorithm proposed uses ideas from shuffling networks and [A3] as conditional swap combined with the well known *bubble-sort* algorithm. The idea is simple and use conditional swaps gadgets which swaps two inputs based on some condition. This algorithm is introduced in section 3.2 and can be seen in pseudocode as algorithm 2. This algorithm yields a perfect uniform permutation.

These shuffle algorithms is optimized to fit to the poker setting introduced in the section 1.1 on poker in chapter 1. This is done such that it only shuffles the required cards and not the whole deck.

The implementations of these algorithms will be introduced in section 3.3 where the choises made will be discussed. At last in section 3.4 a comparison of the algorithms is done. Here I chose which algorithm to use in the implemetation of the poker game and benchmark upon. In this section other type of shuffling networks called Bitonic shuffle network will be introduced and discussed shortly. No implementation of such a shuffle network was done.

In the next section the *Fisher-Yates* algorithm will be studied, it can be found as algorithm 1 in section 3.1.

3.1 Fisher-Yates

Algorithm 1 ***Fisher-Yates***
deck is initialized to hold n cards c.
seeds is initialized to hold n random r values where $r_i \in [i, n]$ for $i \in [1, n]$.

```
 1: function SWAP(card1, card2)
 2:     tmp = card1
 3:     card1 = card2
 4:     card2 = tmp
 5: end function
 6:
 7: function SHUFFLE(deck, seeds)
 8:     for i=1 to n do
 9:         r = seeds[i]
10:         SWAP(deck[i], deck[r])
11:     end for
12: end function
```

The *Fisher-Yates* algorithm can be seen in algorithm 1. It is a well known in place permutation algorithm that given two arrays as input; one that contains the values that should be shuffled, here denoted *deck*, and another holding the values specifing how the first array should be shuffled, here denoted *seeds*. These swap values from *seeds* indicate where each of the original values should go in the swap. 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 a position defined 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, *deck* with n vlause $card_i$, for $i = 0, \ldots, n$. The other holding the values for which the different $card_i$ in the *deck* is to be swapped, *seeds* with n values $seed_i$. If the swap values $seed_i$ 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 $seed_i$ values are chosen accordingly to the algorithm. The algorithm states that $seed_i$ is chosen from an interval starting with its own index i to the size n of the *deck*. This gives exactly the number of permutations required as $card_1$ has exactly n possible places to go. $card_2$ has $n-1$ possible places and so forth until the algorithm reaches $card_n$ which has no other place to go. Since $seed_i \in [i, n]$ we have $n!$ because i runs from 1 to n which should be the case as described in section 1.1.

If the $seed_i$ values contained in *seeds* is not chosen for the right interval but instead all is chosen from 1 to n we would end up having a skew on the probability of the different permutations. As $card_i$ in this case has n possible places to go, this yields n^n distinct permutations. This introduces an error into the algorithm as there should only be $n!$ and as n^n is not divisible by $n!$ for

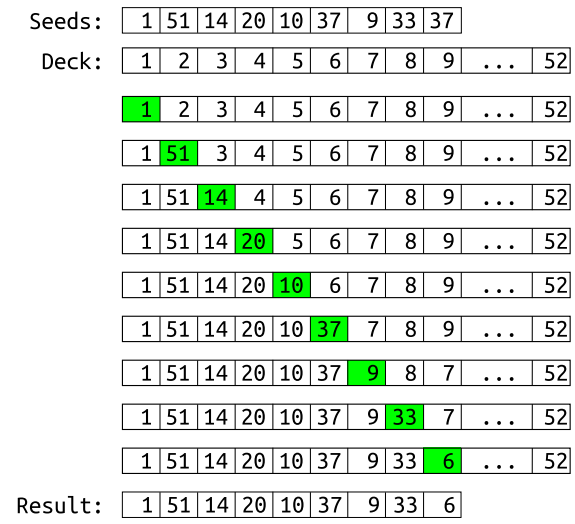


Figure 3.1: Fisher-Yates algorithm in action: In this figure a 9 out of 52 shuffle has been completed to ilustrade how the algorithm works. First 1 is swpaed with 1. Then 2 is swaped with 51. 3 with 14. 4 with 20 and so on until the first 9 numbers has completed a full permutation. Resulting in 1, 51, 14, 20, 10, 37, 9, 33, 6.

$n > 2$. This result in a non uniform probability of the different permutations. The same is the problem if $seed_i$ 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.

In case of the poker game we need 52! permutations. If all $seed_i$ is chosen from $[i; 52]$ we would get 52^{52} possible permutations. As described 52^{52} is not divisible by 52! since $52 > 2$. If $seed_i$ instead is shosen from $]i; 52]$ we get $(52 - 1)!$ permutations which is neither devisible by 52!.

As described in section 1.1 no more then a permutation on the first 20 cards is needed. 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 $Fisher\text{-}Yates$ algorithm is straight forward. Instead of running through n swaps indicated by the size of *seeds* it is enough to run through m swaps. In out case resulting in the input *seeds* only need to have size 20 and therefore the for-loop seen in algorithm 1 in the shuffle function needs to have fewer iterations. Those giving us a full permutation on the first m indexes of *deck*.

In figure 3.1 it is possible to see the $Fisher\text{-}Yates$ shuffle in action. Here the first 9 cards of a sorted deck is shuffled according to the giving seed. Running the algorithm on these inputs give the 9 first cards 1, 52, 14, 20, 10, 37, 9, 33, 6 as output. It is interesting to notice that 37 in the seed twice. Since the algorithm

permute the imput *deck* the value 37 will not be in the output twice. We see that 6 is swapped in the second time the seed 37 is used. This is because the first time 6 and 37 was swapped. This illustate that it is possible for a *card* to be swapped multible times.

3.2 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 through the entire network. On these wires a swap gate is plased. This gate is constructed such that if a condition 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 one I have looked into and implemeted builds on ideas from [A3] where they introduces the conditional swap gate. The algorithm is a combination of the well known *bubble-sort* algorithm and the conditional swap.

In the next section I will introduce the conditional swap algorithm, which can be seen as algorithm 2.

Conditional Swap: The conditional swap algorithm takes two inputs; the first input is an array, denoted *deck* of n cards $card_i$ for $i = 1, \ldots, n$, and the second 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. The first layer contains $n - 1$ conditional swap gates. The second $n-2$ and so on until the las layer consisting of one gate. Each layer is constructed such that a swap gate is placed on two adjacent input wires. Each of these gates overlap with one of the inputwires at the adjacent swap gate. This is illustarated in figure 3.2. The layers are stacked in such a way that the first input wire is only represented in the first layer. Thereby is the first value on the first output wires determined by the first layer of swap gates. Resulting in the first input $card_1$ has n places to go. The second layer determines which output $card_2$ will have and so on. Continuing this way until reaching the last layer where the two last outputs $card_{n-1}$ and $card_n$ will be determined. This gives us a shuffle algorithm with a perfect shuffle and $n!$ different permutations as decired.

If each layer of the swap gates are not decreasing by one on the amount of swap gates this algorithm suffers the problem of producing n^n permutations. Which is not devisible by the decired $n!$ permutations. This resulting in a skew of the probability on the different permutations such that the propability of each permutation is no longer uniform.

Algorithm 2 ***Conditional swap***
deck is initialized to hold n cards c.
seeds is initialized to hold $\frac{n^2}{2}$ random *bit* values where $bit_i \in [0, 1]$ for $i \in [1, \frac{n^2}{2}]$.

```
1:  function ConditionalSwap(bit, card1, card2)
2:      if bit equal 1 then
3:          tmp = card1
4:          card1 = card2
5:          card2 = tmp
6:      end if
7:  end function
8:
9:  function Shuffle(deck, seeds)
10:     index = 0
11:     for i=1 to n do
12:         for j=n-1 to i do
13:             index = index + 1
14:             bit = seeds[index]
15:             ConditionalSwap(bit, deck[j], deck[j + 1])
16:         end for
17:     end for
18: end function
```

Again some optimization can be done to the algorithm since we only need a m out of n permutation. This can be done by letting the outer loop of algorithm 2 run for m iterations instead of n. This yields n possible values for $card_1$, $n-1$ possible values for $card_2$ and so one until $n - m$ values for $card_{n-m}$. This is exactly the amount of permutation we require for our optimized algorithm as this gives us $\frac{n!}{(n-m)!}$. Which is enough for our poker implementation as described in section 1.1

In figure 3.2 a run of algorithm 2 can be seen Here a 9 out of 52 variant is used. It can be seen that the inputs *deck* is soreted and holds the values to be shuffled and *seeds* which are binary and indicates if two values should be swaped. The first 52 bits of the *seeds* decides if the first *card* values should be shuffled. Which is not the case in this run. Then the next 51 bist from *seeds* indicate that 51 should be swapped all the way accross to the wire repecenting the second out card. This implies that all cards 51 passed on its way will now be on the right adjacent wire to where it was prior to the swap. That is why the third output *card* with value 14 starts at wire index 15 and output wire four with value 20 starts at wire index 21. So the algorithm continiues until it outputs the first 9 cards shuffled as 1, 51, 14, 20, 10, 37, 9, 33, 6.

3.3 Implementation

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

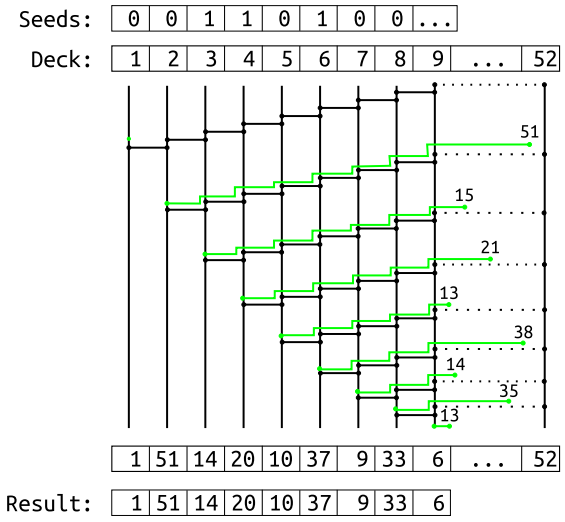


Figure 3.2: Conditional swap algorithm in action: In this figure a 9 out of 52 shuffle has been completed to ilustrade how the algorithm works. Each bit in the *seeds* indicate if a gate should be swapped. Since the size of *seeds* is so big I have tried to ilustrate which wire each value is located at before moved in a layer resulting in 1, 51, 14, 20, 10, 37, 9, 33, 6.

TODO: Describe solutions
TODO: Describe gadgets of the two shuffle protocols

Because the *DUPLO* protocol was chosen as the MPC protocol to use in this study the first hurdel was to generate the circuits that should handle the shuffling of the 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 the *DUPLO* project came shiped with a compiler for generating circuits with the right format. Therefor before staring to implemt the shuffle algorithms I had to read up on the documentation for the compiler. The documentation is from the first version of the *Frigate* compiler which was extended to suit the *DUPLO* format.

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 $Fisher\text{-}Yates$ 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.

EVT SOMEWHERE HERE: BEGIN This algorithm requires much more randomness as input compared to the $Fisher\text{-}Yates$ 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 $Fisher\text{-}Yates$ because it will produce n^n distinct possible sequences of swaps which is not compatible with the $n!$ possible permutations. END

3.4 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 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 $Fisher\text{-}Yates$ 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 $Fisher\text{-}Yates$ 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 $Fisher\text{-}Yates$ 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 $Fisher\text{-}Yates$ algorithm has conditional swaps from index i

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

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 representation 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 $Fisher\text{-}Yates$ 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 [A3] 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 $Fisher\text{-}Yates$ algorithms can aquire since it creates a circuit of size $O(n^2)$. But as argued earlier the *Conditional swap* and $Fisher\text{-}Yates$ algorithms are easily optimized to the setting studied in this thesis.

TODO: Bitonic algorithm, optimization? can maybe be done by flipping

Circuit	*Fisher-Yates*	*Conditiona swap*	Difference(%)
Number of wires	835	4151	397
Number of gates	59790	57364	4
Number of free XOR gates	37433	39001	4
Number of non-free gates	22357	18363	21

Table 3.1: Compariason of the two implemented algorithms after compilation to $DUPLO$ circuits.

---

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.

In this section I will do a compariaon of the two compiled circuits. I chapter 3 on the shuffle algorithms I compared the idea of the algorithms. Now after compiling them it is easier to argu how the two implemetations stack up against each other. In the table above where the circuits are compared on four different parameters. The first parmeter are the number of unique wires in the circuits. It is clearly that the *Fisher-Yates* algorithm has less unique wires than the *Conditional swap* algorithm.

TODO: Does the number of unique wires effect the performance of the protocol?

The second parameter mesured in the table is the total number of gates in the circuit. Here the numbers are farly similar. Doing a calculation shows that *Fisher-Yates* has around 4% more gates than *Conditional swap* which is not that big a differend. But when looking into the difference on the two last parameters. First we rememer that we are in a setting where XOR gates are assumed for free. The parameters are the number of free XOR and non-free. When the amount of these gates are mesured we see a bigger difference. When doing the calculations we see that *Fisher-Yates* has 4% less free gates than *Conditional swap* which is not much. We see the big difference when looking at the non-free gates. Here *Fisher-Yates* has 21% more non-free gates that what is the case for *Conditional swap*. This is a big difference and in this case the effect of the difference is negativ on the performance of the protocol. Therefor even though the amount of randomness used in the algorithm is much smaller the overhed of ensuring that the seed is in the right intervall and the swap gadgets constructed for each card index result in a wors circuit for the $DUPLO$ protocol. Therefor the *Conditional swap* algorithm is used in the implemetation.

Chapter 4

Poker Implemetation

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 The Poker Game

This is the overall structure of every implemetation that wants to use the *Duplo* protocol. In the next part I will introduce how this was used to implement the poker game in this project.

TODO: The pokser setting

Before starting on the implemetation it was inportant to figure out which setting to study. Different settings was proposed before a choise was made. Mainly two settings were discussed. One where the *Constructor* and *Evaluatr* acteded as players of the poker game themselves or a setting where they acted as servers where the players connected to. The first one was the one chosen mainly because of the simplicity of doing it and timelimmet in the project. But

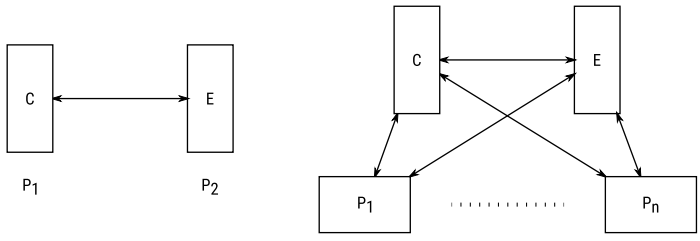


Figure 4.1: The two different settings discussed to implement. The one on the left where the *Constructor* and the *Evaluator* also are the players and the one on the right wher they act as servers where players connect and then localy construct output based on output from the servers.

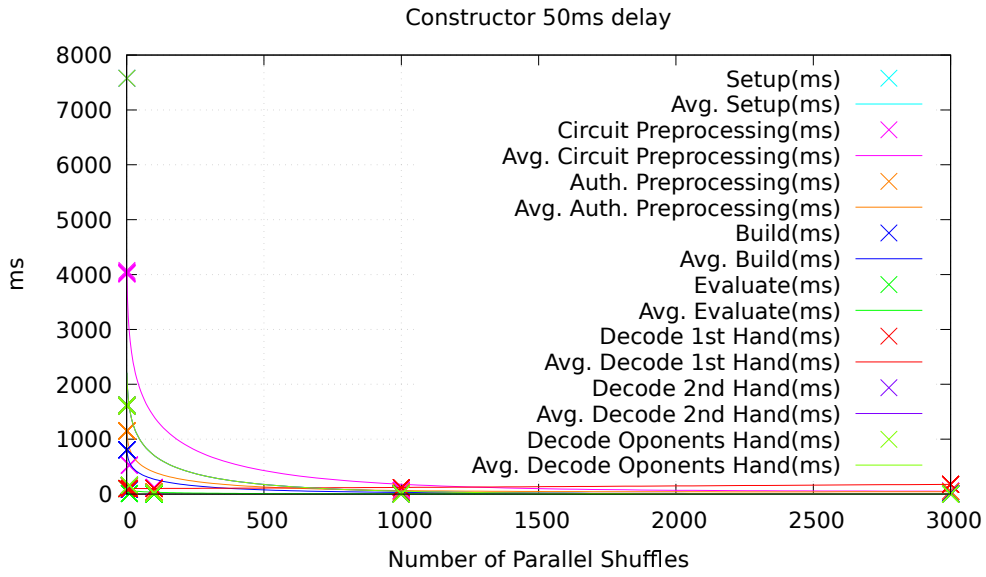
the other setting has some nice features which I will note here. That setting resembles more what is done in online poker to day. Where a client connects to a game provider which deals the cards. Then instead of connecting to one party which is not completly trusted the player connects to two servers which runs a secret share of the protocol together. Such that th eplayer recieves a share from each party. One of these serves could be a Stat authorized server in which the palyer would put its trust. This would bring down the main part of the latency from the setting implemented in this study. Which will be discussed later. This could be done as the preprocessing of the circuit could be done on forehand and in times with low load. But this implemetation adds a complexity in the implemetation phase where message authentication codes are needed to ensure that non of the two servers changed the output. This second setting also allow for more the two player to participate in the game.

TODO: Describe the roles of the two parties

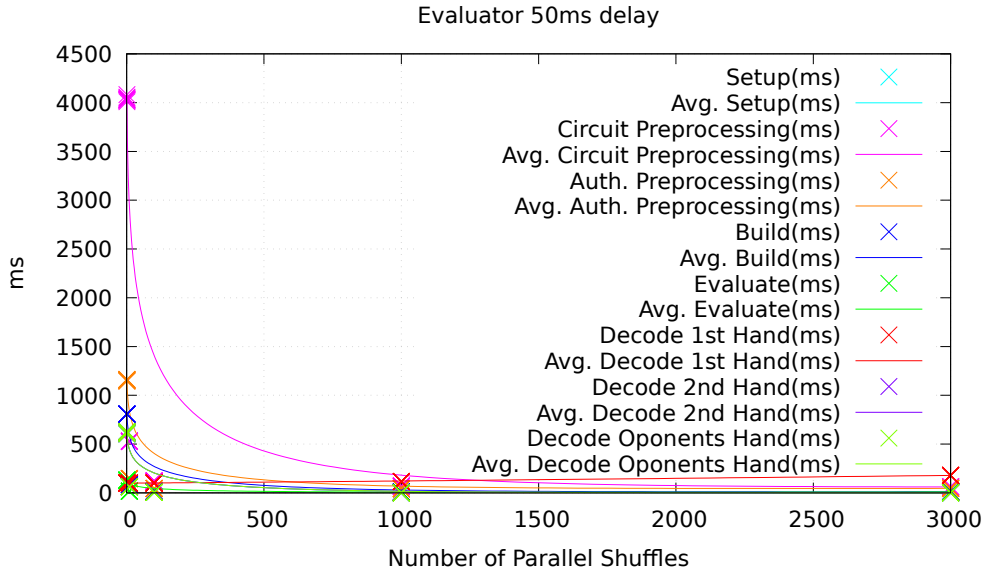
TODO: why can only one hand be played at a time, broken bristol compiler ehn frigate v2 was fixed

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

TODO: Discuss latensy in different framework calls

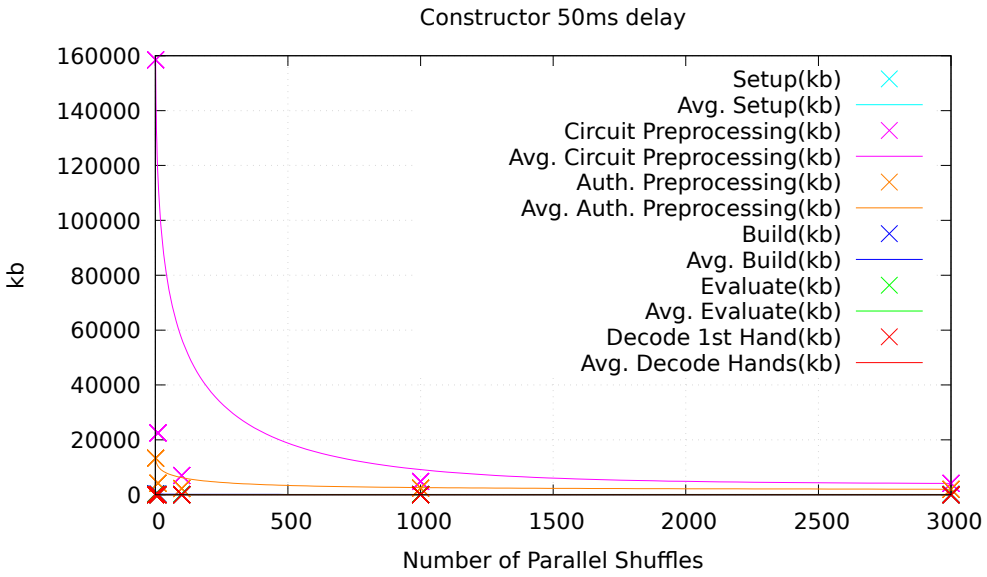


(a) Constructor: 50ms simulated network latency. Showing the time used in different phases per deck shuffled.

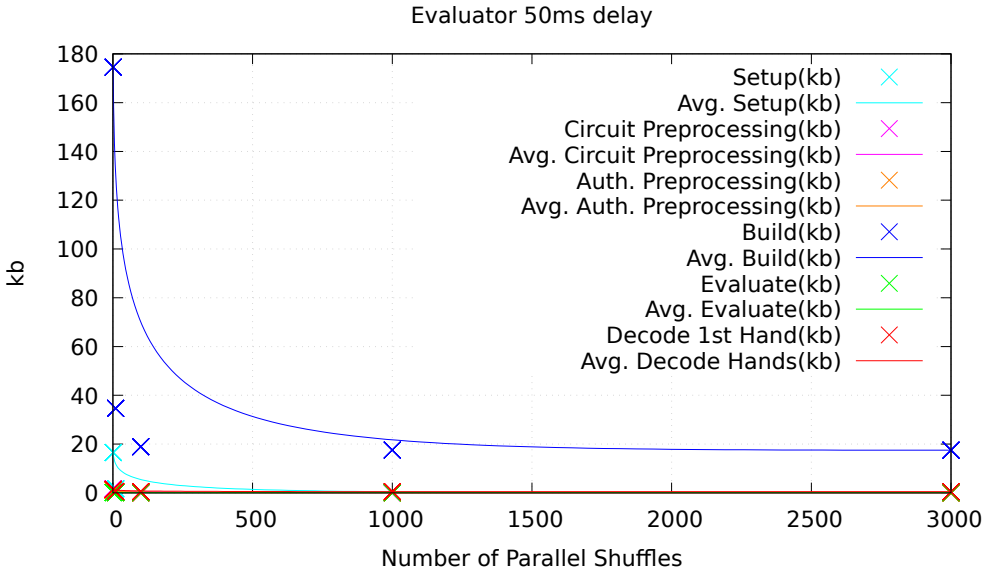


(b) Evaluator: 50ms simulated network latency. Showing the time used in different phases per deck shuffled.

Figure 4.2: Comparison of the Constructor and Evaluator on which phases requires the most time.



(a) Constructor: *kb* sent in different phases.



(b) Evaluator: *kb* sendt in different phases.

Figure 4.3: Comparison of Constructor and Evaluator on which phases most *kb* are sent to the other party.

Phase	Delay(ms)	Number of Parallel Shuffles 1	10	100	1000	3000
Setup	50	2210.37	162.11	17.63	2.86	2.17
	0	1431.11	114.23	12.66	2.36	2.00
	-	779.29	47.88	4.97	0.50	0.17
Circuit preprocess	50	4033.93	527.35	119.99	67.52	51.36
	0	1564.52	201.07	50.75	29.75	19.50
	-	2469.41	326.28	69.24	37.77	31.86
Auth preprocess	50	1150.64	143.22	39.92	37.93	42.54
	0	93.10	31.95	20.02	22.90	33.57
	-	1057.54	121.27	19.90	17.03	8.97
Build	50	807.18	86.06	15.78	8.42	10.82
	0	22.84	6.79	6.27	6.46	9.02
	-	784.34	79.27	9.51	1.96	1.8
Evaluate	50	100.82	10.22	1.18	0.30	0.30
	0	0.69	0.24	0.24	0.23	0.24
	-	100.13	9.98	0.94	0.07	0.06
Decode keys 1st hand	50	100.37	100.39	100.91	115.62	177.92
	0	0.20	0.13	0.37	10.20	77.15
	-	100.17	100.26	100.54	105.42	100.77
Decode keys 2nd hand	50	100.37	10.23	1.24	0.50	0.86
	0	0.13	0.08	0.90	0.25	1.30
	-	100.24	10.15	0.34	0.25	-0.44
Decode Opponents hand	50	100.35	10.22	1.23	0.48	0.57
	0	0.11	0.08	0.09	0.23	0.91
	-	100.24	10.14	1.14	0.25	-0.34

(a) Constructor: comparison of the different phases and concurrent shuffles with and with out delay.

Phase	Delay(ms)	Number of Parallel Shuffles 1	10	100	1000	3000
Setup	50	620.25	61.89	6.15	0.62	0.21
	0	117.58	11.36	1.05	0.09	0.03
	-	502.67	50.53	5.01	0.53	0.18
Circuit preprocess	50	4037.57	528.96	120.39	68.30	60.86
	0	1574.70	202.97	51.89	30.79	29.63
	-	2462.87	325.99	68.50	37.51	31.23
Auth preprocess	50	1154.60	146.40	44.90	42.32	46.33
	0	94.98	34.70	24.73	27.37	37.65
	-	1059.62	111.70	20.17	14.95	8.68
Build	50	807.23	86.12	15.83	8.50	10.88
	0	22.88	6.82	6.33	6.59	9.14
	-	784.35	79.30	9.50	1.91	1.74
Evaluate	50	129.49	17.91	5.43	5.44	9.58
	0	15.71	5.13	2.92	2.29	7.96
	-	113.78	12.78	2.51	3.15	1.62
Decode keys 1st hand	50	100.60	100.63	101.17	115.91	179.12
	0	0.29	0.19	0.43	15.01	125.68
	-	100.31	100.44	100.26	100.90	53.44
Decode keys 2nd hand	50	100.59	10.23	1.47	22.08	128.10
	0	0.18	0.08	0.22	14.31	124.84
	-	100.41	10.15	1.25	7.77	3.26
Decode Opponents hand	50	100.57	10.23	12.70	21.81	122.89
	0	0.17	0.07	0.22	14.16	123.14
	-	100.40	10.16	12.48	7.65	-0.25

(b) Evaluator: comparison of the different phases and number of concurrent shuffles with and without delay.

Table 4.1: Comparison of the time consumption of the Constructor and Evaluator.

Phase	Number of Parallel Shuffles 1	10	100	1000	3000
Setup	12.70	1.27	0.12	0.01	0.00
Circuit Preprocess	158470.00	22518.10	7023.94	4837.94	4127.91
Auth. Preprocess	13329.90	4170.08	2249.69	2437.49	1986.51
Build	327.71	128.49	107.24	105.12	104.96
Evaluate	122.84	122.84	122.84	122.48	122.84
Decode keys	5.94	2.38	2.02	1.99	1.99

(a) Constructor: *kb* sent per shuffled deck in different phases.

Phase	Number of Parallel Shuffles 1	10	100	1000	3000
Setup	16.55	1.65	0.17	0.02	0.01
Circuit Preprocess	1.59	0.16	0.02	0.00	0.00
Auth. Preprocess	1.06	0.11	0.01	0.00	0.00
Build	174.55	34.67	18.95	17.55	17.49
Evaluate	0.11	0.10	0.10	0.10	0.10
Decode keys	1.44	0.58	0.49	0.48	0.48

(b) Evaluator: *kb* sent per shuffled deck in different phases.

Table 4.2: Comparison of the Constructor and Evaluator on the amount of data sent in the different phases.

Chapter 5

Conclusion

TODO: Conclusion bla ...

Appendix A

Codebase

In this appendix references will presented to the different codebase used during the thesis. An url to the repositories on github will be presented togheter with a short description of where the most interresting parts for this project can be found.

TODO: What hardware has been used during the project!

A.1 DUPLO

The *DUPLO* repository at GitHub can be found here `https://github.com/AarhusCrypto/DUPLO`

The documentation on the site is clear and illustrates clearly how it is compiled such it can be tested. No documentation is presented for how interacting with the framework can be done for new implementations. The most interesting part for the sake of this project is located in the `src` folder. Here the `CMakeLists.txt` file is located which specifies how the project is compiled, this is overwritten when compiling the poker implementation. The folder `src/dublo-mains` is where the actual implementations of the *Constructor* and *Evaluater* can be found. Here the implementations of the poker *Constructor* and *Evaluator* will be placed.

For a easy setup of duplo a docker instance is created and can be found at `https://hub.docker.com/r/cbobach/duplo/`. This can be started in docker version `17.05.0-ce` with the command;

```
docker run -it --network:host cbobach/duplo
```

The `--network:host` flag is not secure but is the fart easy way to let the container running the *Constructor* expose the port on wich the container running the *Evaluator* needs to connect. When running two instances of these docker containers the *Constructor* and *Evaluator* is runned using one for these commands for the default setting:

```
./build/release/DuploConstructor
./build/release/DuploEvaluator
```

A.2 Frigate

The *Frigate* repository on GitHub is a subrepository to *DUPLO* and can be found at `https://github.com/AarhusCrypto/DUPLO/tree/master/frigate`.

The documentation of how *Frigate* is installed with the special versions of some of the libraries used is specified in the documentation of *DUPLO*, the link can be found in appendix A.1. It is also here the documentation of how to compile *DUPLO* circuit formats are done.

To find the documentation of the `.wir` file format a look should be taken at the link above. Here the specifications are of how wire acces is done for example. It is here all functionallities that are implemented in the language is listed and how they are used. This documentation is narrow at some places. It does for example not specify that the modulo operator `%` does only work on powers of 2.

Using the docker image from `https://hub.docker.com/r/cbobach/duplo/` and running the followint command, in docker version `17.05.0-ce`:

```
docker run -it -v host/dir:container/dir cbobach/duplo
```

Will start a container where it is possible to compiler a `.wir` file using the comtainer. For it to work the `.wir` files has to be located in the `host/dir` then the following commad can be run to compile the functionality:

```
./build/release/Frigate container/dir/functionality.wir -dp
```

The `-db` flag ensures that the *DUPLO* file format is generated. The *DUPLO* generate file will have the extention `.wir.GC_duplo` this can then be feeded to the *DUPLO* framework using

```
./build/release/DuploConstructor -f container/dir/functionality.wir.GC_duplo
./build/release/DuploEvaluator -f container/dir/functionality.wir.GC_duplo
```

Then the new functionallity will be runned in the default *DUPLO* environment.

A.3 Poker

In this section the GitHub reposioties to the different pahses will be linked. A short description to where the intersting pars are will be presented.

A.3.1 Circuit implemetation

The different circuit `.wir` files can be foun in the Github repository at `https://github.com/cbobach/sp`

Here the implementations of the shuffle algorithms are present as `fisher_yates_shuffle.wir` and `conditional_swap_shuffle*.wir`. Multiple versions of the `conditiona_swap_shuffle*.wir` file are present with different values for `*`. This is to allow for multiple sequential hands to be plyead, these files are then used when testing the *DUPLO* framework to show its capabilities.

Only one version of `fisher_yates_shuffle.wir` is located in the repository since this is a slover algorithm in this setting as discussed in section 3.4.

The files `init_deck.wir`, `xor_seed.wir` and `correcred_seed.wir` are all modules that are called by the shuffle algorithms. The `init_deck.wir` file is used by both algorithms and hardwires the card values to their respective wires. The `corrected_seed.wir` file is used by the *Fisher-Yates* algorithm to ensure thet the seed feeded to the shuffle algorithm is in the correct intervalls as explained in section 3.1. The `xor_seed.wir` file is used by the *Conditional swap* algorithm ot generate the seed used by the shuffle.

It is also here that the parser used to debug the *Frigate* compiler is located and is found as `parse.py`. The other python script found here `count-gate-types.py` is the one used to compare the amount of gates types for the compiled circuits.

A.3.2 DUPLO implemetation

A.3.3 Test results

Primary Bibliography

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

[A2] Vladimir Kolesnikov, Jesper Buus Nielsen, Mike Rosulek, Ni Trieu, and Roberto Trifiletti. Duplo: Unifying cut-and-choose for garbled circuits. Cryptology ePrint Archive, Report 2017/344, 2017. http://eprint.iacr.org/2017/344.

[A3] 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

[B4] F. Yates R.A. Fisher. Statistical tables for biological, agricultural and medical research, 6th ed. http://hdl.handle.net/2440/10701, 1963.