Secure Distributed Poker using MPC

Christian Bobach, 20104256

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

AARHUS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

TODO: abstract

Resumé

TODO: Resume

Acknowledgments

TODO: Acknowledgments

Christian Bobach,
Aarhus, June 6, 2017.

Contents

Chapter 1

Introduction

In this thesis I have made a practical study of the application for 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.

The hope is to show that a MPC protocol can be used to guarantee that the cards are deal fairly without any big cost in terms of time delay for the players.

For the inplemetation of the poker game 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 forfills 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.

I will now give a short introduction to each chapter such that you as a reader know what to expect. In chapter 2 I cover the bacis of *DUPLO* . The idea and their claims. I introduce why this protocol and framework was chosen. I argue for the security of the protocol and why it covers the case of implementing poker. Lastly I explain how the circuit compiler used in this project which shiped with the *DUPLO* project works.

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. I explain how the implementation of the algorithms was done. I introduce some optimizations to the algorithm such that their size in terms of gates are reduced. At last the algorithms are compared such that the most efficient one can be chosen.

In chapter 4 I describe how the poker game was implemented and how I used the *DUPLO* framework. I argue for the setting chosen to implement. I discuss some of the choises done when doing the implementation which resulted in reduced communication between the parties. Lastly I discuss the benchmarking of the implementation. The benchmarking was done on different parameters, the amount of simuntainios shuffles, the effect of network latency and the effect of bandwith.

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 [A3] 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 do. 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 evaluation.

As it can be read in [A3] 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 perform well in each end of the spectrum, when it comes to the size of the circuits but not the other. In the paper they come up with a new approch to do cut and chose in a 2PC protocol in the malicious setting. The idea is to garble subcomponets of the circuit and get a optimum somewhere inbetween the two extremes; garbling of complete circuits or garbling on gate level, cut and chose. Their 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 [A3] it is clear 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 then those compared to.

The fact that it is developed at Aarhus University, such that the people with knowleged of the protocol is close by is one of the main factors for using *DUPLO* . But the fact that *DUPLO* supported the possibility of single and distinct wire openings helped the decision. Exactly this property is needed to be able to hadle uniqe opening of cards to one player without the other learning anything abouth the card.

2PC Before going in to details on why the *DUPLO* framework was choosen I will give an introduction to what $2PC$ and *Garbled circuits* is. $2PC$ is a special case of MPC from [A2] where only two parties P_1 and P_2 participate in a distributed evaluation of the functionality $f : \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^*$. The goal in a protocol allowing for evaluation of $f(x_1, x_2) = (y_1, y_2)$ between P_1 with input x_1 and output y_1, and P_2 with input x_2 and output y_2, is to guarntee *privacy* and *correctness*. *Privacy* is to guarntee that no more than the output is learned from the computation. *Correctness* guarantees that both parties recieve the correct outputs. When discussing MPC we have to take *independacen of input*, *guaranteed output delivery* and *fairness* into account. If we let m denote the number of parties in the MPC protocol, in case of $2PC$ m is 2. Then let t be the treshold for the number of corrupted parties in the protocol, in case of $2PC$ t is 1. Since party P_i can trust it self. In a MPC setting *guaranteed output delivery* and *fairness* can be achieved for any protocol with a broadcas channel, with $t < \frac{m}{2}$. This implies that non of both *guaranteed output delivery* and *fairness* can be achived in the $2PC$ setting used by *DUPLO* , because $t = \frac{2}{2} = 1$. But *privacy*, *correcness* and *independance of input* can be achived in the $2PC$ setting when the parties have access to a braodcase chanel and we assume the existence of enhanced trapdor permutations. This only holds for the computational setting of adeversary powers.

This ensures us that in the setting studied in this paper we can get *privacy*, *correctness* and *independance of inputs* from the $2PC$ *DUPLO* protocol.

Garbled circuits The *DUPLO* protocol uses encrypted circuits or *garbled circuts* from [A2] which is a setting where the functionality f for computation is represented as a boolean circuit $\mathcal{C}$. The garbeling of $\mathcal{C}$ gives the protocol the desired properties as argued above. Let $g : \{0,1\} \times \{0,1\} \rightarrow \{0,1\}$ denote a gate in $\mathcal{C}$, then garbling $\mathcal{C}$ follows from garbling all gates. Let the input wires of g be labled w_1 and w_2, and output w_3. Let k_i^0 and k_i^1 be generated for each wire in g with $i = 1, \ldots, 3$. The desire is to be able to learn $k_3^{g(\alpha,\beta)}$ from k_1^α and k_2^β without revaling any of $k_3^{g(1-\alpha,\beta)}$, $k_3^{g(\alpha,1-\beta)}$ or $k_3^{g(1-\alpha,1-\beta)}$. g is then defined by these values:

$$
\begin{aligned}
c_{0,0} &= Enc_{k_1^0}(Enc_{k_2^0}(k_3^{g(0,0)})) \\
c_{0,1} &= Enc_{k_1^0}(Enc_{k_2^1}(k_3^{g(0,1)})) \\
c_{1,0} &= Enc_{k_1^1}(Enc_{k_2^0}(k_3^{g(1,0)})) \\
c_{1,1} &= Enc_{k_1^1}(Enc_{k_2^1}(k_3^{g(1,1)}))
\end{aligned}
$$

Where Enc is a private-key encryption scheme that has indistinguishable encryptions under chosen plain-text attacks. g is then represented as a random permutation of the values $c_{0,0}$, $c_{0,1}$, $c_{1,0}$ and $c_{1,1}$. Now the correct $k_3^{g(\alpha,\beta)}$ can be learned by the followin computation $Dec_{k_1^\alpha}(Dec_{k_2^\beta}(c_{i,j}))$, for $i, j \in 0, 1$. If more than one value yealds non-$\perp$ return `abort` else define k_3^γ to be the only non-$\perp$ value. Because of the requirements of the encryption scheme k_3^γ is the correct value with negligable probapility. To generate a complete garbled $\mathcal{C}$ the description above is followed for each gate. Resulting in a garbled circuit representing f that ensures *privacy* and *correctness* during evaluation.

2.1 The *DUPLO* framework

In this section I will introduce the different functions from the framework and what they achive. 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 passes, the *Evaluator* trust, that the remaining circuits are valid and then these are used during evaluation.

The overall construction of the framework consists of different functions, which allow for the right communication between the two parties. The functions should be called in a predetermined order to ensure that the correct information is at the parties at the time when needed. At the same time the functions have been spilted up such that local computations can be done inbetween these framework calls.

To run the protocol and use the framework a *Constructor* and an *Evaluator* is created. First of all they read the circuit file specifying the functionality desired. 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 `Setup` function, which initializes the commitment protocol. After this, they start the preprocess phase of the componets in the circuit by running the `PreprocessComponentType` function call. This takes n and f as inputs; n is the amount of garbled circuit to produce, and f is the functionality that will be evaluated. Then the function enerates n garbled representations of f to be securly evaluated.

Then the `PrepareComponents` function is called. This takes i as input, the amount of input authenticators to produce. These authenticators is used to securly transfer the input keys from the *Constructor* to the *Evaluator*. This call also attach all required output authenticators. The authenticators ensures that only one valid key will flow on each wire of the garbled componets.

After this, the `Build` function is called, this takes a boolean circuit $\mathcal{C}$ as input. This constructs the complete garbled circuit, which is to be evaluated later by the call to `Evaluate`. The `Build` call ensures that the function componets specified by the composed circuit file is soldered together, such that they compute the functionality specified by $\mathcal{C}$. This is done such that the ouput wires from one subfunction is feeded to the right input wire on another subfunction.

The next call is then made to `Evaluate`, which takes x_1 and x_2 as input. Here x_1 is the first input to the computation, and x_2 is the second. The call then evaluates the garbled circuit given these two inputs. This yealds a garbled output of the functinality $f(x_1, x_2)$. When the parties hold a gabled output a call to `DecodeKeys` can be made and the output of the functionality can is revealed.

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 player. The split-up of `Evaluate` and `DecodeKeys` functions allows for opening of output wires in different rounds which helps us to achive good round complexity when creating our poker implementation. This can also be used if the output will be used as input for another secure computation.

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.

In the section on protocol details in [A3] 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.

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*, here denoted **G** for generator, and a honest *Evaluator*, denoted **E**. The simulator $\mathcal{S}$ plays the role of **E** in the protocol, but is not given access to the inputs $x_{\mathbf{E}}$ of **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 **G** running the protocol can not distinguige between talking to **E** or $\mathcal{S}$.

Theorem 1 *If generator* **G** *is corrupt and evaluator* **E** *is honest and the protocol does not abort then the following holds with negligable probability. For each input gate id,* **E** *holds* $k_{id} = K_{id}^{x_{id}}$*. For all input gates of* **E** x_{id} *is the correct input of* **E***. For each output gate* id'*,* **E** *holds* $k_{id'} = k_{id'}^{y_i d}$ *where* y_{id} *is the plaintext value obtained by evaluating circuit* $\mathcal{C}$ *on* x_{id}*. The probability of the protocol aborting is independant of the inputs of* **E***.* [1]

$\mathcal{S}$ is constructed such that it first constructs $x_{\mathbf{E}} = \mathbf{0}$ as the zero inputvector for **E**. It then inspects the commitment of the input gates of **G** and learns k_{id}^0 and Δ_{id}, where id is the gate identifier. From these k_{id}^1 is computed. By theorem 1 $k_{id} = k_{id}^{x_i d}$ can be retrieved. This defines the input $x_{\mathbf{G}}$ for **G**. $\mathcal{S}$ then calls $\mathcal{O}_{x_{\mathbf{E}}}(\cdot)$ with input $x_{\mathbf{G}}$ and learns $y_{\mathbf{G}}$. If $y_{\mathbf{G}} = \perp$ then $\mathcal{S}$ aborts, else $\mathcal{S}$ sends k_{id} as computed in recovery mode. This can be done since k_{id}^0, k_{id}^1 and y_{id} is known to $\mathcal{S}$.

It follows from theorem 1 that the protocol and the simultaion aborts with the same probability. When they do not abort the key returned to **G** is the same as **E** would have sent, except with negligable propability.

The same type of simulation proof is done for the case with a honest *Constructor* and a corrupt *Evaluator*. This proof can be found in appendis B.2 in [A3].

While I whent through the proof of *DUPLO* I found a typo both in section $B.1$ and $B.2$ where they had switched around on the corrupt and honest party when they recall the task of the proof. This has been anaunced to them and a fix will be made.

[1] This theorem is a cited from [A3] with small textual modifications. It can be fond as thoerem 2 in appendix A.

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.

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 *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

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

---

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 2.1 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 I accounted the problem of the modulo operator I used a small amount of time to reserch if it was posible to implement this functionality easily. During the search I noticed that the implementation of modulo to some power of 2 is simple. While the research I did seemed to indicate that for valuse not a power of 2 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 [B5]. 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 [A4] 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.

Algorithm 1 ***Fisher-Yates***

deck is initialized to hold n cards c.

seed is initialized 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
```

3.1 Fisher-Yates

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 *seed*. These swap values from *seed* 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 $i = 0, \dots, n$. The other holding the values for which the different $card_i$ in the *deck* is to be swapped, *seed* with n values $seed_i$. If the swap values $seed_i$ from the *seed* 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 *seed* 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

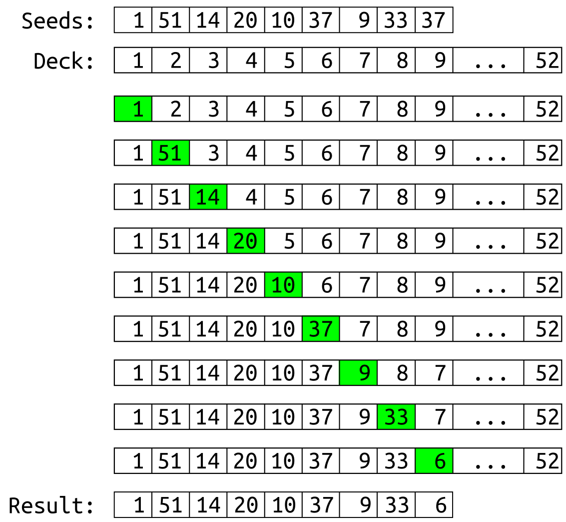


Figure 3.1: Fisher-Yates algorithm in action: In this figure a 9 out of 52 shuffle has been completed to ilustrade how the algorithm works. First 1 is swpaed with 1. Then 2 is swaped with 51. 3 with 14. 4 with 20 and so on until the first 9 numbers has completed a full permutation. Resulting in 1, 51, 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-Yates* algorithm is straight forward. Instead of running through n swaps indicated by the size of *seed* it is enough to run through m swaps. In out case resulting in the input *seed* 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-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 gadget is plased. This gadget is constructed such that if a condition is satisfied the input on the two wires are swapped. By placing these swap gadgets correctly on the input wires it is possible to get a complete uniform random permutation of the input on the output wires. The swap gadgets are created according to [A4] as figure 3.

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 gadgets 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 [A4] where they introduces the conditional swap gadget. 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 *seed* of size $l = \frac{n^2}{2}$ bits b_j where $j = 1, \ldots, l$. The algorithm creates $n-1$ layers of conditional swap gadgets. The first layer contains $n-1$ conditional swap gadgets. The second $n-2$ and so on until the las layer consisting of one gate. Each layer is constructed such that a swap gadget is placed on two adjacent input wires. Each of these gates overlap with one of the inputwires at the adjacent swap gadget. 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 gadgets. 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 gadgets are not decreasing by one on the amount of swap gadgets this algorithm suffers the problem of producing n^n permutations. Which is not devisible by the decired $n!$ permutations. This resulting in a skew

Algorithm 2 ***Conditional swap***

deck is initialized to hold n cards c.

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

```
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
```

of the probability on the different permutations such that the propability of each permutation is no longer uniform.

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 *seed* which are binary and indicates if two values should be swaped. The first 52 bits of the *seed* decides if the first *card* values should be shuffled. Which is not the case in this run. Then the next 51 bist from *seed* 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 $1,\ 51,\ 14,\ 20,\ 10,\ 37,\ 9,\ 33,\ 6$.

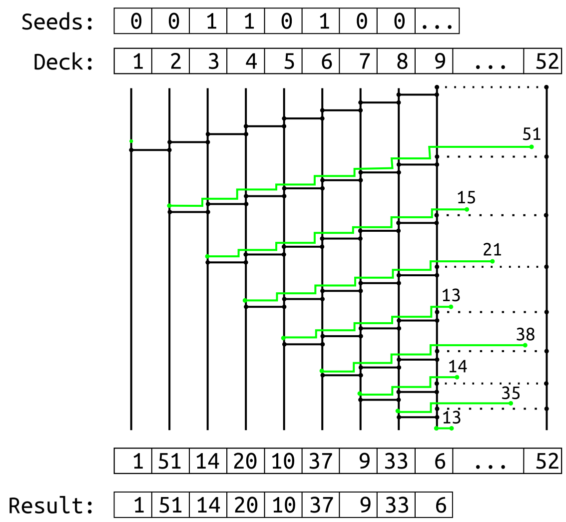


Figure 3.2: Conditional swap algorithm in action: In this figure a 9 out of 52 shuffle has been completed to ilustrade how the algorithm works. Each bit in the *seed* indicate if a gate should be swapped. Since the size of *seed* is so big I have tried to ilustrate which wire each value is located at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

3.3 Implementation

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, which can be found by following the descriptions in appendix A.3. The documentation was from the first version of the *Frigate* compiler which was extended during the *DUPLO* project generate the decired *DUPLO* format. An introduction to the *Frigate* compiler created in the *DUPLO* project can be fondu in section 2.3 and the codebase is introduced in appendix A.3. Because it was a long time since I last had worked with circuits and the documentation of the incorporated functionality of *Frigate* I used some time getting up to speed. Much of this time was by trial and error which gave me some hard earned experience throug small examples.

Before going into details on the implementations a link to the sourcecode can be found in appendix A.4.1. In the implementation of the two shuffle algorithms there are five different functions was implemented as different modules that could be used. Both shuffle algorithms *Fisher-Yates* and *Conditional-swap* makes calls to the function `initDeck`. Which initializes a hardcoded representation of the card deck which is to be shuffled when evaluating the generated

Gate Type	Non-optimized	Optimized	Difference(%)
Free Gates	97753	39001	60
Non-free Gates	47739	18363	62
Total	145491	57364	60

Table 3.1: *Conditional-swap* : Comparison of the non-optimized and optimized versions of the algotithm. The comparison is done on the amount of each gate type in the compiled circuit.

---

circuit. This is done by a for-loop inserting the values of the deck on the right wires. In the implementation 6 bit variables is used for the representation of each card, as 6 bist allows for 64 different representations, because $2^6 = 64$. Which is enough to represent each unique card in the deck. It is importand to notice that the variable indexing the start position of every card needs a representation with at least 9 bits to hold the correct value, since $\lceil \log_2(52 \cdot 6) \rceil = 9$. If only 6 bits were used for indexing only wires up to opsition 63 could be assigned and not all 312, as required since $52 \cdot 6 = 312$. Therefor 9 bit is used for this variable.

Then looking into the rest of the stucture of the *Conditional-swap* algorithm we see that the first function used is the `xorSeed`. This function handles the XOR of the *seed* recieved from the two parties. This is a straightforward implementation using the build-in XOR function `^`. After this the calle to `initDeck` is done shuch that the result for the two functions can be fead into the shuffle algorithm. Such that 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 constructing the layers in the network, and another for generating the swap gadgets in each respective layer. Folowing the structure of algorithm 2 we ensure that we are ending up with an algorithm producing the right amount of permutations according the discussion made in section 3.2. Using the optimizations proposed in the same sections results in a clear reduction in the number of gates, as seen in table 3.1. Most important we see a 62% reduction in the non-free gates on the optimized version.

In case of the *Fisher-Yates* algorithm the things stack up a bit differently. The first function in this case is the `correctSeed`. The function takes the *seed* from the two parties and correct them as described in section 3.1. At first each of the inputs are splitted up into representations of 6 bits such that they each hold 52 values, $seed_{C_i}$ for the $Constructors$ and $seed_{E_i}$ for the $Evaluator$. Then the new input reprsentations are added such that $seed_{C_1}$ is added with $seed_{E_1}$. This ensures that the new value $seed_i$ is at most $2 \cdot (2^6 - 1)$ because of the representation. Since the addition of two 6 bit values can not be guaranteed to fit inside another 6 bit value a representation with more bits is used to store the resulting value. The idea was to use a modulo reduction on $seed_i$ to gurantee that it was inside the intervall describetd in section 3.1. Since the modulo reduction implemented in $Frigate$ only supports divisors that is a power of 2 the modulo reduction in $Frigate$ can not be used, because $seed_i$ is not guaranteed to have this property. I used a lot of time to figure out that the $Frigate$ modulo operator only worked on powers of 2. This implemetation detail was not specified anywhere in the documentation. The first idea was to fix the problem by implementing a modulo function that I could use instead. I therfor put some research time into this problem, but it is to be rather complex to achive. Therefor another solution was chosen to overcome the problem. Since the input $seed_{C_i}$ and $seed_{E_i}$ to the functionality is assumed to be in the right intervalls the solution was to subtract the boundray of the interval $I_u = 52 - i$ from $seed_i$ if this exceets I_u. Doing it this way it yealds a resulting value $seed_i$ in the right intervall. This is do to the fact that $seed_{C_i}$ and $seed_{E_i}$ can at most be I_u. This ensures that $seed_i$ is at most $2 \cdot I_u$. Then $seed_i - I_u$ is guaranteed to be at most I_u. In the implemetation this was done by introducing an if statment checking if $seed_i$ exceeded I_u. It is nothworthy to mention that all values have had an unsigned representation until now. But since the comparison of to values needs a signed representation as stated by the documentation $seed_i$ was converted. This corrections to the implemetations now ensures that the randomness given to `shuffleDeck` has the right form. But only if $seed_{C_i}$ and $seed_{E_i}$ are inside the right intervall $[0; I_u]$.

The second function used in the implementation is the same as in the case for the *Conditional-swap* algorithm where the `initDeck` function is called to initialize the representation of the *deck* which is to be shuffled. The last function called is the `shuffleDeck` function which is different from the one from the *Conditional-swap* algorithm. This function consist of an outer for-loop that runs through the cards $card_i$ of the deck. Because circuits are a static representation as disscused earlier in section 2.3 it is not possible to assign a wire value based on a variable input. Therefor this for-loop generates layers of conditional swap gadgets. Resulting in $52 - i$ swap gadgets in each layer, for $i = 0, \ldots, 51$. This is represented by the inner for-loop. In this way a composed gadget is generated for each $card_i$ such that the $card_i$ can be swapped with any other $card_j$, where $j = i, \ldots, 51$. This composed gadget is a composition of $52 - j$ desitinc swap gadgets. Such that the desired propability is reached as described in section 1.1. Both an optimized and non-optimized version of the algorithm was implemented to see how big the gain of the optimization was. This can be seen in table 3.2, where we see that the optimization result in a 40% decresae in the number of non-free XOR gates.

3.4 Comparison

In this section I will try to compare the two algorithms on their internal structure. I will compare the algorithms based uppon their gate composition and based on that choose which one to continiue with in the implementatin of the poker game. When comparing the algorithms I use the optimized versions as it would be one of these that will be used because of their gain in the number of non-free XOR-gates.

Gate Type	Non-optimized	Optimized	Difference(%)
Free Gates	61806	37433	39
Non-free Gates	37344	22357	40
Total	99150	57790	42

Table 3.2: *Fisher-Yates* : Comparison of the non-optimized and optimized versions of the algotithm. The comparison is done on the amount of each gate type in the compiled circuit.

The first we will look at is the input to the `shuffleDeck` functions. The both take the *deck* as input, which in both cases is generated by the function `initDeck`. This does therfore not yeald any difference to the algorithms. Then when looking at the *seed* it is clear that there are some differences. Both in terms of representation and in size. First looking at the representation of *seed*, in *Fisher-Yates* $seed_{FY}$ and *Conditional-swap* $seed_{CS}$. The $seed_{FY}$ is a representation of 20 values $seed_{FY_i}$ in the interval $[0; 52 - i]$, for $i = 1, \dots, 52$. Where $seed_{CS}$ does not have any abstrac representation and therefore $seed_{CS_i}$ has the binary representation $[0; 1]$. The difference in the representations is one reason why we se a difference in the size of the $seed_{FY}$ and $seed_{CS}$ in terms of bits. Where the size of $seed_{FY}$ is 112 since 6 bists are used for the representation of the 20 seed values. The size of $seed_{FY}$ is 830 because one bit is needed per swap gadget, which is $\sum_{i=52-20}^{51} i$. As we see it is also the way the algorithm uses the *seed* that effect the size. Where the *Fisher-Yates* algorithm constructs composed gadgest consisting of multiple swap gadets the *Conditional-swap* algorithm only constructs swap gadgets. Even thoug the algorithms has different approches to generate the swap gadgets they end up generating the same amount of swap gadgets. Because *Fisher-Yates* generates 19 composed gedgest CG_i consisting of $52 - i$ swap gadgets, for $i = 1, \dots, 19$. Yealding the same number of swap gadgets as in *Conditional-swap* .

If we then instead turn our attention to the way the algorithms handle the *seed* before they are fead to `shuffleDeck` we se some big differences. Because both algorithms takes $seed_C$ and $seed_E$ as inputs from *Constructor* and *Evaluator* the algorithms needs to generate one single *seed* that can be used by `shuffleDeck`. This adds an overhead to the algorithms. This is not much for the *Conditional-swap* algorithm since it can use the XOR function because it uses one bit of randomness at a time. As described in section 3.1 this is not the case for *Fisher-Yates* since it uses 6 bits of randomness $seed_i$ at a time. Because there is the restriction on the $seed_C$ and $seed_E$ and the *seed* input to `shuffleDeck` in *Fisher-Yates* the overhead added is more. The split-up of $seed_C$ and $seed_E$ into $seed_{C_i}$ and $seed_{E_i}$ does not add any overhead, but the addition of these does. Also the check to test if $seed_i$ is greather then I_u and the subtraction adds an overhead to the overall circuit. The differences can be seen in table 3.3 where it is clear that `xorSeed` adds significantly less overhead to the circuit compared to `correctedSeed`.

We see that the `xorSeed` has two non-free gates, which is strange since it only does XOR as should be free. This is because of the bit constants **0** and

Gate Type	`correctSeed`	`xorSeed`	Difference(%)
Free Gates	4347	1661	62
Non-free Gates	1873	2	100
Total	6220	1663	73

Table 3.3: Comparison of the overhead added to the algorithms by handling the *seed*'s. The comparison is done on the amount of each gate type in the compiled circuit.

Gate Types	*Fisher-Yates*	*Conditional-swap*	Difference(%)
Free Gates	37433	39001	-4
Non-free Gates	22357	18363	18
Total	59790	57364	4

Table 3.4: Compariason of the *Fisher-Yates* and *Conditional-swap* algorithms after compilation to *DUPLO* circuits. The comparison is done on the amount of each gate type.

1 which are implemented using AND or $NADN$ with both inputs comming from the same wire. Therefore will every circuit generated with the compiler have these two non-free XOR gates.

When looking in to the overall composition of *Fisher-Yates* and *Conditional-swap* we get the results as seen in table 3.4. We see that *Conditional-swap* is overall 4% bigger in terms of the amount of gates than *Fisher-Yates* . On the more important comparison is that *Conditional-swap* has 18% less non-free XOR gates. Therefore is the *Conditional-swap* algorithm the one that will be used in the implementation of the poker game.

As a note to the comparison it shouls be mentioned that some of the variables used in `correctedSeed` are bigger in terms of bit size than needed. But since this is only a fraction of the 1873 non-free gates seen in table 3.3 this wouls not change that *Conditional-swap* has less non-free gates then *Fisher-Yates* .

At last I will give a short comment on another possible shuffle algorithm that could have been used. This other types of sorting networks that the one studdied in section 3.2. In [A4] they also uses a algorithm known as the *Bitonic* merge sort algorithm. Such an algorithm is constructed of what is known as $half - cleansers$. These $half - cleansers$ are constructed such that the input is guaranteed to have one peak p, $i_1 \leq \cdots \leq p \geq \cdots \geq i_n$. Then the output is half sorted such that the highest values are in one of the two halfs. Resulting in a algorithm that can sort any input. This guarantees that one input $seed_i$ can every other place. If the conditional swap gadget is used then it can be used as a shuffle algorithm instead of a sorting algorithm. This type of sorting network generates a circuit of size $O(n \cdot log(n))$ which is better then what the *Conditional-swap* and *Fisher-Yates* algorithms can aquire which is $O(n^2)$. But as argued earlier the *Conditional-swap* and *Fisher-Yates* algorithms are easily optimized. This seems not to be the case for a *Bitonic* shuffle network.

As a result of this we see that for some card games it can be better to use another algorithm then the ones studied in this thesis since it may outperform them. We now know that a *Bitonic* algorithm would produce a smaller circuit that *Fisher-Yates* and *Conditional-swap* but since the two algorithm are easy to optimize such that they produce a relative small circuit. It is not clear that a *Bitonic* algorithm will produce a circuit that is smaller. This implies that there will ba a cross over at some point where it is better to shuffle a complete deck then only parts of it as is done in our case. This can even be the case when only a part of the deck is needed.

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 I have encountered and how I have chosen to overcome those.

In section 4.1 I describe the process of implementing the poker game using *DUPLO* . First I discuss the setting in which the implementation is done. Then an introduction to the decissions made and woh the interaction with the framework was done.

In section 4.2 I introduce the testing that was done on the implementation. I explain what and how it was done. The implementation was tested on several parameters to try to compare the perfoemance against what could be expected of a real world online poker game. The inplementation was tested against the amount of data sent and time consumption in different phases. When the optimal setting was found this was used to test the effect of network latency and bandwith on the protocol. All test were done per simountaious shuffled deck.

4.1 The Poker Game

As described in section 2.1 where the structure of the framework was introduced. This structure is what every implemetation using the *DUPLO* protocol needs to have. In the next part I will introduce how this was used to implement the poker game in this project.

Before starting on the implemetation it is inportant to decide which setting of the poker game should be studdied. Two different settings was proposed and discussed, these can be seen in figure 4.1. The first is one where the *Constructor* and *Evaluatr* act as players of the poker game themselves. They will play against each other and each decide on which and how many cards should be changed. The other setting is where the *Constructor* and *Evaluator* act as servers where players connect to. These players will act as clients connecting to both the *Constructor* and *Evaluator*. Here the *Constructor* and *Evaluator* will run the protocol as described, but use inputs from the clients. The first one is the one chosen mainly because of the simplicity and timelimit of the

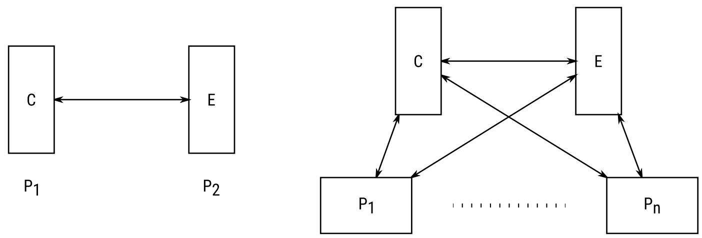


Figure 4.1: The two different settings discussed to implement. The one on the left where the *Constructor* and the *Evaluator* also are the players and the one on the right wher they act as servers where players connect and then localy construct output based on output from the servers.

---

project. The other setting has some interesting features which I will mention here. The second setting resembles what is used in real world online poker today. Today a player connects to a server which handles the shuffleing and dealing of cards. Then instead of connecting to one server which is not completly trusted the player connects to two servers which runs a secret share of the *DUPLO* protocol. Then the player will recieve a share from each party and reconstruct the computed output. This setting distribut the trust issiue with the setting used to day. It could be the example that one of the serves in the *DUPLO* setting is a State authorized server in which the palyer then would put its trust. Then game makers is then required to use the protocol against this server to be allowed to sell games in that contry. It will alow for a distributed instead of a single piont trus issiue. In the second setting there are no limits to how many playes could connect to the servers to play a game of poker. In this way the $2PC$ is not a restriction for the amount of players. This setting requiers a way of handling authentisity between the players, the *Constructor* and *Evaluator*. This is needed to encure that non of the players sends different inputs to the *Constructor* and *Evaluator*, and to encure that the *Constructor* and *Evaluator* does not send different outputs to the players. This is not a simple task to overcome, therefor the first setting was chosen.

The way *DUPLO* is constructed it allows for preprocessing of the circuits, such that the right informtion is at the right party before evaluation. The overall protocol can be catogorized in 3 different main phases. The first phase is the *Preprocess* phase which consist of the framework calls from section 2.1; `Connect`, `Setup`, `PreprocessComponentType`, `PrepareComponets` and `Build`. The next phase is the *Evaluation* phase which calls the `Evaluate` framework function. The last phase is the *Online* phase which consists of the `DecodeKeys` calls. Building the protocol this way allows for an intens *Preprocess* phase where a lot data is processed and communications is done. This allows for a faster *Evaluation* phase and results in a small overhead when running the *Online* pahse.

I will now describe the implementation of the poker game. First of all we need to both implementa a *Constructor* and an *Evaluator*. These runs the protocol in parallel with the same framework calls, but holds different information during the phases. First we read the composed circuits in to both parties such that they know which functionality is to be computed. Then the first interaction between the two parties is when the `Connect` call is done. On the *Constructor* site this generates the server functionality which the *Evaluator* connect to with this call. After this a call to `Setup` was done to initilize the commitment scheme used by the protocol. Then a call to `PreprocessComponetType` is done on each of the subcircuits. Given the subfunctions and the number of garbled copies to generate. Efter this a call to `PrepareComponets` is done, given the number of input wires in the circuit. It the generates the key authenticators to securly transfter the input keys and attach all output authenticators. At last the circuit are generated by the call to `Build`. Given a representation of the composed circuit, this call genrates a garbled representation of the prepared components such that they compute the functionality specified by the composed circuit. No special input is given to any of the parties during the phase except of the circuit represention of the functionality to be computed. Therefor this will be denoted the *Preprocess* phase. As these calls can be done done ahead of time and without knowing anything other then the functionality to compute.

The next phase is the *Evaluate* phase, where the evaluation of the garbled circuit is done. Before the call to `Evaluate` can be done each of the parties need to generate their inputs to the shuffle algorithms. Since the shuffle algorithms always shuffles the same values the *deck* is harcoded into the circuit. Therefore it is sufficient for the parties to provide the *seed* for the algorithm. This is done by generating 16 bytes of random data. This data is then used as a seed for a pseudo random generator producing the 830 bit randomness used for each *deck* that needs shuffled. This generated randomness is used as the input to `Evaluate`. The function also takes an empty array as input to store the garbled output.

The last phase is the *Online* phase it is here the calls to `DecodeKeys` is done. This phase is run for each game played. The amount of games possible is specified in the number of simuntainiusly shuffled decks. In this phase the *Constructor* and *Evaluator* must first agree on which outputwires should be opened to which party. Therefor the first they do is to generate an array containing the wire indexes to be opened to the *Constructor* and to the *Evaluator*. These are then opened by a call to `DecodeKeys`. The values are then translated into card representations and displayed to the players. The translation from values $n \in \{1, \dots, 52\}$ to cards (v, s), where v is the value and s is the shade of the card, are done by letting $v = (n \mod 13) + 1$ and $s \mod 4$. This yealds no collitions on (v, s) since 13 and 4 has no common multiplum less then 52.

When drawing the first hand the *Constructor* is always dealt the first five cards from the deck. The *Evaluator* are dealt the five cards starting from index 10 to 14. This is the most optimal way to deal the cards as this requires the least calls to `DecodeKeys`. This does not change the propability of the cards dealt. As known for real world card games one or two cards are normaly dealt at a time, this is propably done because the shuffle algorithm doeas not have a uniform distribution. But has a skew such that sequences of cards are more likely to repeat in the next game. This is not the case for our shuffle algorithms, since they have a uniform propability distribution on the output.

After the first hand have been dealt an interface is displayed to the players such that they can choose which cards to change. This is done using a terminal interface where the user inputs; 0 if no cards needs to changes, 1 if the first card should be changed, 2 for the second and so on. If multiple cards is to be changes this is done by seperating the card indexes with a comma $','$ like $4, 5$. To allow for a new hand to be dealt with the specified cards changed the parties first sents the amount of cards thay want to change. The they sent the indexes of the cards they want to change. Now each party knows which wires should be opened. The old array holding the index of output wires is updated to hold the new index wires. Then the second and final hand is opened by a call to `DecodeKeys`. The card are translated and displayed to the parties. The cards with indexes from 5 to 9 is reserved for the *Constructor*'s second hand, while the cards with indexes from 15 to 19 is reserved for the *Evaluator*. Opening the last hand this way adds an overhead since one card may be openend in both the first and second hand. On the other hand this allows for less communication before the last revilation round in the game.

The last round of each game is the round where the revelation of the oponents hand is done. This is done without any communicatin other than a last call to `DecodeKeys`. The input to this last call is done by switching the inputs for the *Constructor* and *Evaluator* as they already know which wires was openend to the oponent in the last hand. By opening the oponents card this way we ensure that the oponenet does not learn the cards which were discarded.

At last when all the decks are played the statisitcs are writen to the log files. This is the timings and amount of data send by the different calls. This data is used in the section 4.2 to discuss the effeciency of the *DUPLO* protocol in the setting of a poker game.

A small note on the leak of information when communication indexes for `DecodeKeys`. The oponent always know how many and which cards are changed because the parties needs to agree on which wires to open, but this is not any different from real life poker. Therefor this leak of information should not be considered a security problem.

Another note on the `DecodeKeys`. When implementing the poker game I experienced problems with the call to `DecodeKeys` when trying to do unique openings to the single parties. It was later discovered that an update to the protocol had only been done on the oneside of the protocoal and not the other.

4.2 Benchmarking

In this section I will introduce all the testing that have been done on the poker implementation. This has been done to see if a implementaion using a $2PC$ framework can reach runningtimes close to real life online poker.

To do benchmarking a local machine was used. The hardware setup can be found in appendix A.1. This setup was used to ensure a stabile environment such that no newtwork latency chunks would obstruct the results. To handle the experience of real networks a script was used to simulate bandwidth and latency. The bandwith was set to 1 Gb/s during all test. This is in the high end compared to most bandwith connections found in Denmark, but was choosen to get a faster roudtrip on the test environment. The latency was changed during the testing to see how it effected the implementation.

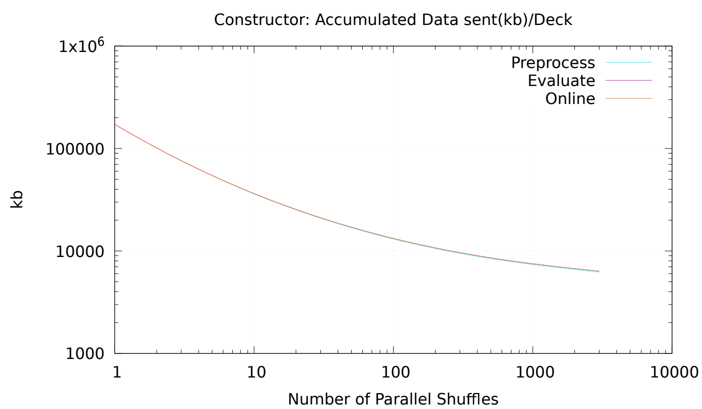
In the implemntation different flags was implemnted to allow for an easy change of the setup. The flags implemented were; `-f` for specification of the circuit file to use, `-e` for the number of threads to use in the different phase, `-n` for the number of parallel shuffles in the circuit, `-i` to allow for interaction or not in the card change phase, `-ip_const` for specifing the ip address of the constructor, `-p_const` for specifing the port the *Constructor* is listening on and lastly the `-d` flag for ram only mode, where the computation is done without writing anythig to disk.

The first tested was the timing of each shuffle done when multiple was done in parallel. This was done to see how the approch to cut and chose in *DUPLO* effected the poker implementation. This allows us to see if we reach a optimum of the number of simuntainious shuffles. To do this different circuits was generated and compiled to mesure the timings. Because the *DUPLO* framework do not allow for soldering of *DUPLO* format circuits, the ciruit file used should contain all simuntainiously shuffle of decks. Therefor different variants of the *Conditional-swap* algorithm was made where 1, 10, 100, 1000 and 3000 simuntainius decks was shuffled. The 3000 was choosen as a maximum since the memory limit was exceede on the hardware when more was tested. These variants of circuits was all compiled following the instructions in appendis A.3. The *Conditional-swap* implementations from section 3.2 was used because it has the least amount of non-free *XOR* gates and therefor should be faster then *Fisher-Yates* . This is was is shown in table 3.4.

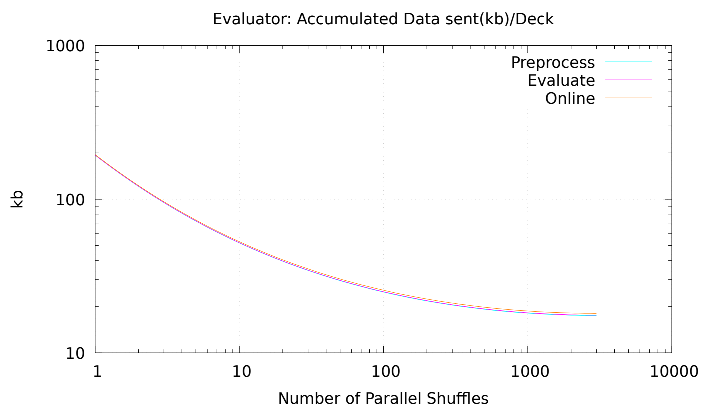
A bash script was setup to allow for automatic testing of all these different circuits, this is explained in A.4.3. This script ensured that 10 timings was done for each of the circuits to guarantee more fair timing when taking the average. All the timings were loged and can be found via appendix A.4.3. One timing was removed from the *Constructor* since it differentiated form the rest. This is the timing for the first run of the poker game where the timing of `Setup` was long, this is because of the time it takes to start the *Evaluator* and prepare the bandwith. These tests was done with a latency of 50ms. This was found to be a fair latency based on pinging different ip addresses in europa as seen in table 4.1.

When dicussing the results I will refer to the phases as described in section 1.1, *Preprocess*, *Evaluate* and *Online*. This is done to reduce the amount of information in the figures auch that only the most necesary information is precent.

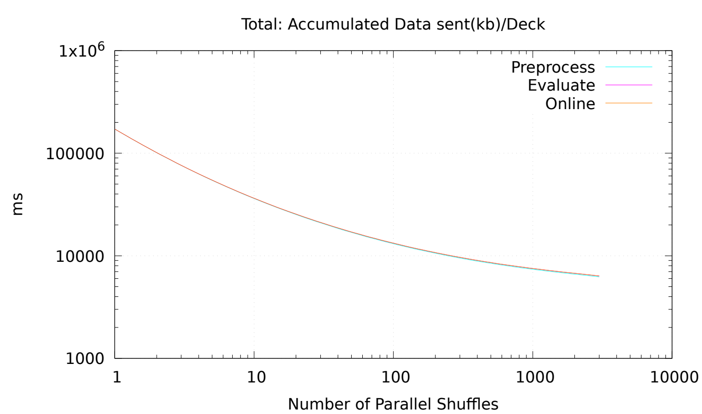
The first we will look at is the amount of data send in *kb* per shuffled card deack as seen in figure 4.2. We see the accumulated dats sent per deck shuffled. The data is represented on a duble logaritmic scale. It is easy to see that as more simuntainiously shuffles are done the least data is sent. This implies that the most data sent is the overhead of setting up the protocol. As we see it is har



(a)



(b)



(c)

Figure 4.2: Data sent: Comparison of *Constructor* and *Evaluator* in *kb*'s sent to the other party. (a) *Constructor*: Accumulated data sent per deck shuffled on a duble logarithmic scale. (b) *Evaluator*: Accumulated data sent per deck shuffled on a duble logaritmic scale. (c) *Total*: Accumulated data sent per deck shuffled on a duble logaritmic scale.

Homepage	Avg. latency(ms)
google.dk	18
google.com	54
au.dk	2
uoa.gr	132
uzh.ch	41
univie.ac.at	36
Total avg.	47

Table 4.1: Ping: Timings of network latency to different locations in europa.

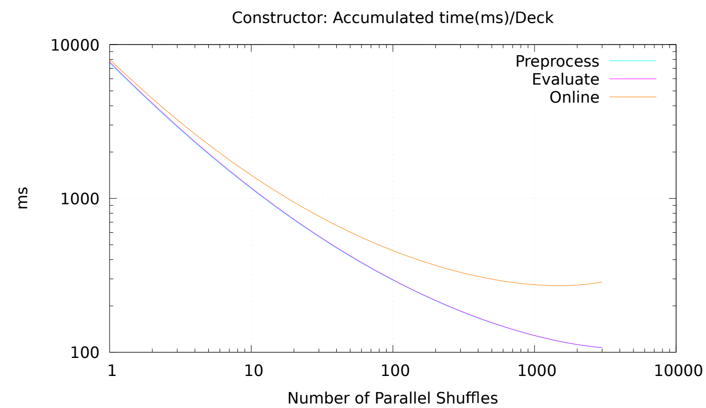
---

to distinguish the different lines on the plot. This is because the *Preprocess* phase is the one where the most data is sent. Relative to this nearly no data is sent in the *Evaluate* and *Online* phase. Remembering har it is only the input to the functionality that is sent in the *Evaluate* pahse. In the *Online* pahse we call `DecodeKeys` three times and therefore is only these keys that are sent. Where as in the *Preprocess* phase the information of the garbling, soldering and authentication is done. Therfore more data is sent.

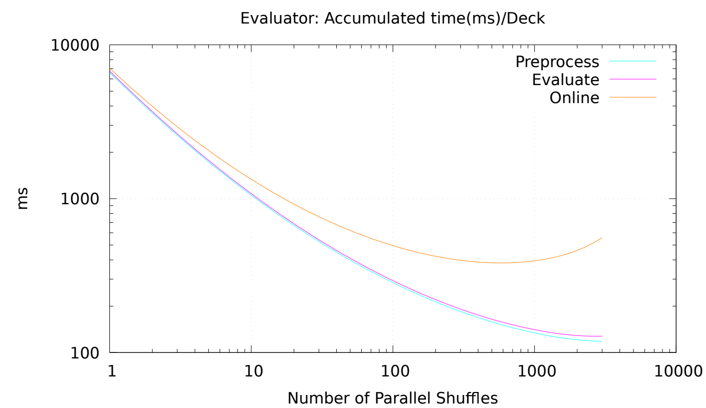
In figure 4.2b we see that the line is flattening out indicating that there are not much more to gain on the *Evaluator* site in terms of *kb* sent, by shuffleing more decks simuntainious. Looking at figure 4.2a we see a different tendency where the plot is still decreasing indicating that some gain can still be done on the *Constructor* site. This may indicate that more than 3000 simuntainus shuffles can benefit on the amount of data sent as seen in 4.2c, which is still decreasing. Looking at the scale on *kb* axis it is obvious that it is the *Constructor* that sents the most data and therefore the one that require the most simuntanious shuffles to bring the overhead of doing $2PC$ down.

This is excatly as expected before doing the experiments. The amount of *kb* sent would decrease as more shuffles were done because of the overhead of doing $2PC$. Even going beond the 3000 shuffles seems to give a decrese in data sent. Even though the gain is not the same as for the first 1000 shuffles there seems to be some data transfer gain.

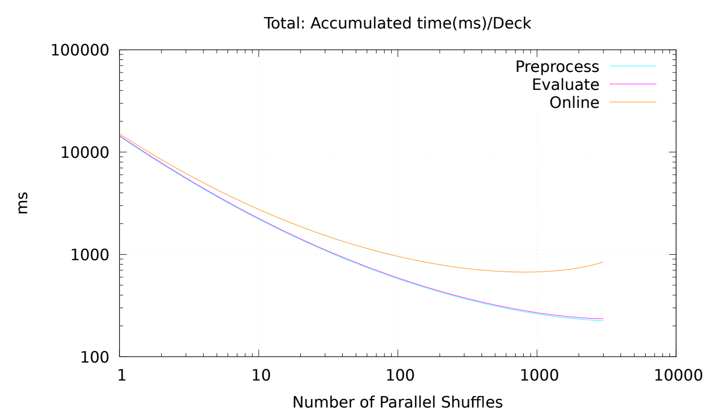
In the next section we will look a the time used per shuffled deck, which can be seen in figure 4.3. The plots are the accumulated runningtimes per shuffled deck on a duble logarithmic scale. In figure 4.3a we see the time used in the different phases for the *Constructor*, in 4.3b the *Evaluator* and lastly in 4.3c we see the total time used on the framework calls. In figure 4.3a on the *Constructor* side we see that the *Preprocess* phase accumulates the most of the time. We also see that the time use in the *Preprocess* phase is decreasing an approching 100 ms per deck shuffled when shuffeling 3000 decks simuntainiously. We also see that the *Evaluate* phase does not add any significant time consumption to the evaluation. Where it is a completly different case when looking at the *Online* phase. Here we see that the space between graphs encreases significantly when approching the 3000 shuffled deck. We will take a look at this later. But for now we can conclude from the perspective of



(a)



(b)



(c)

Figure 4.3: Time: Comparison of *Constructor* and *Evaluator* in *ms*'s used. (a) *Constructor*: Accumulated time per deck shuffled on a duble logarithmic scale. (b) *Evaluator*: Accumulated time per deck shuffled on a duble logaritmic scale. (c) *Total*: Accumulated time per deck shuffled on a duble logaritmic scale.

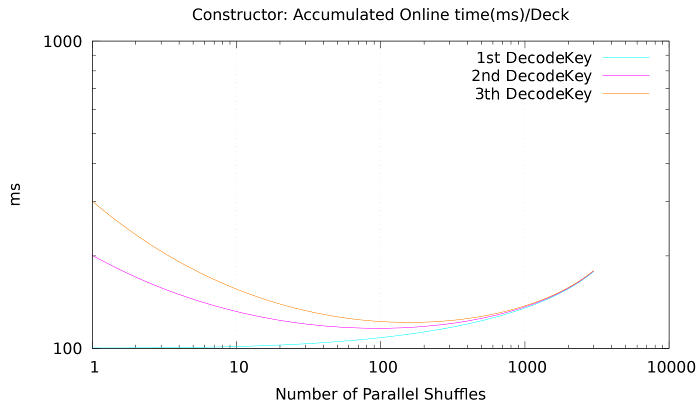
the *Constructor* and in terms of accumulated running time per shuffled deck an optimum on this hardware is around 1500 siminutanious shuffels. When looking at the *Evaluator* in figure 4.3b we see the same tendency as for the *Constructor*. The *Preprocess* is the one using the most time and approching 150 ms per shuffled deack. For the *Evaluator* we see a small encreas in time used per shuffled deck when pasing the 1000 mark. But over all the time spent on these two phases is still slightly decreasing. Once again we see a increase in the time used on the *Online* phase. For the *Evaluator* the encrease is more significant than for the *Constructor*. Therefore the optimal amount of shuffled deck from the *Evaluator*'s point of view is around 500 decks. When combining the running times from the *Constructor* and *Evaluatro* we get what we see in figure 4.3c. We see the same tendencies but with a optimum around 1500.

This was not what was expected before doing the experiments. The expected outcome was that we would se a decrease in time used per shuffle. It was not expected that an encrease in time consumed would encrease when doing more than 2000 shuffles. Therefore more experiments was done to cover the reasons.

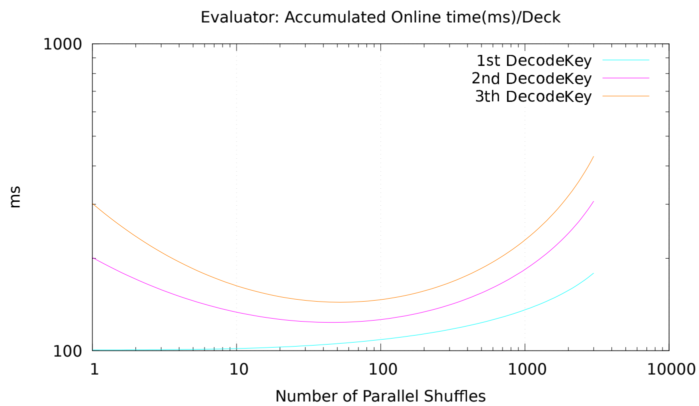
In this part I will cover the reasons why we see a encrease in time consumed by the *Online* phase. First off all we remember that the *Online* phase has three `DecodeKeys` calls. In figure 4.4 we see the accumulated times used on the `DecodeKeys` framework calls. For the *Constructor* in figure 4.4a we see a encreas in the time used on the first `DecodeKeys` call. While the other calls decreases as expected. This implies that somthing happens in the first `DecodeKeys` call that we did not expect. Looking at the *Evaluator* in figure 4.4b we see a graph that looks different, this is because the main work done in the `DecodeKeys` call is done by the *Evaluator*. Here we see a increase in bothe the first and second `DecodeKeys` call, while the third seems constant. When consulting the figure 4.4c for the combination of *Constructor* and *Evaluator* we see that the encrease in time used by the first `DecodeKeys` call happend before 100 shuffles, while the encrese by the second call happens after 500 shuffles.

The encrease in time spent on the `DecodeKeys` calls is probably from the fact that the implementation tries to cach as much as possible. From figure 4.2 we see that in the *Online* phase nearly no data is sent. By consulting the data appendix A.4.3 we see that approximatly 2.5kb is sent. Sending this data on a network with a bandwith of 1Gb/s takes 2.5ms, which is negligable as the amount of data decreses. As explained earlier the test was done on a network with 50ms latency. Since we do not know how many rounds of communication the two parties has, we can not conclude any thing from this, beside the fact that this can be seen as a constant. Therefor it must be some implementation specific detail of the framework which is different at the two parties. The fact that the *Constructo* does not use any time in the second and third `DecodeKeys` call indicate that some form of caching is taking place.

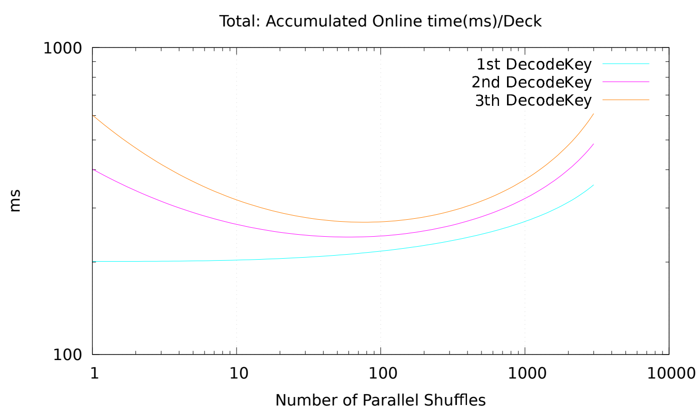
TODO: Discuss the online phase.
TODO: Main idea is IO wait.
TODO: Timings in online phase is fluctuating, maybe I/O wait. More shuffles higher propability that some will end up waiting.



(a)



(b)



(c)

Figure 4.4: *Online* Time: Comparison of *Constructor* and *Evaluator* in *ms*'s used in the *Online* phase. (a) *Constructor*: Accumulated time per deck shuffled on a duble logarithmic scale. (b) *Evaluator*: Accumulated time per deck shuffled on a duble logaritmic scale. (c) *Total*: Accumulated time per deck shuffled on a duble logaritmic scale.

TODO: Test ram only variant to see the rise in ms in online pahse is do to disk read, evt on 1000 shuffles

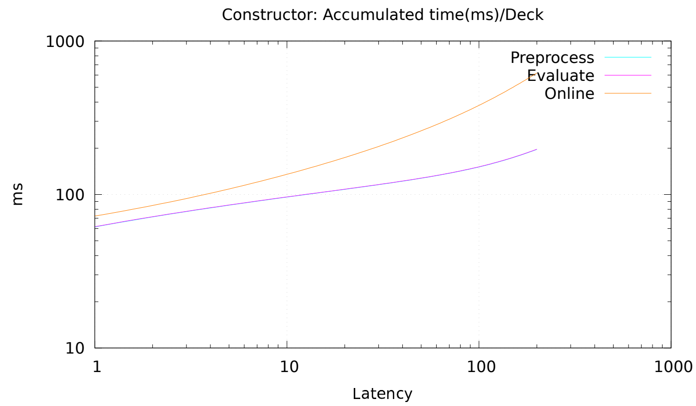
TODO: different amount of delay

In this section I will describe how the protocol handles network latency. This can be seen in figure 4.5. In figure 4.5c that the overall impression is that the protocol handles delay fine until 100ms. From here on it seems that there are a increase in the slowdown of the protocol. When looking at the *Constructor* in figure 4.5a we see that the *Online* phase is the one effected the most by the network latency. When comparing these graphs with table 4.1 it seems the protocol is performing fine for the latencies found there.

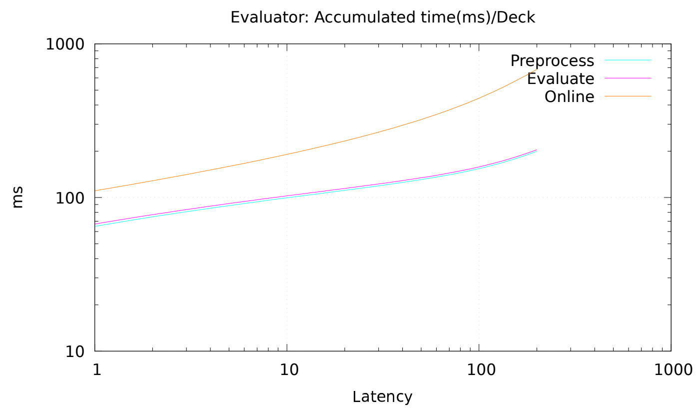
TODO: different amount of bandwidth

4.3 Discussion

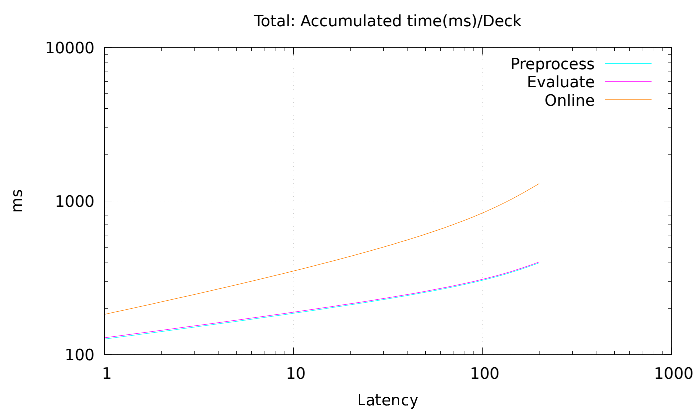
TODO: ahead of time generation



(a)

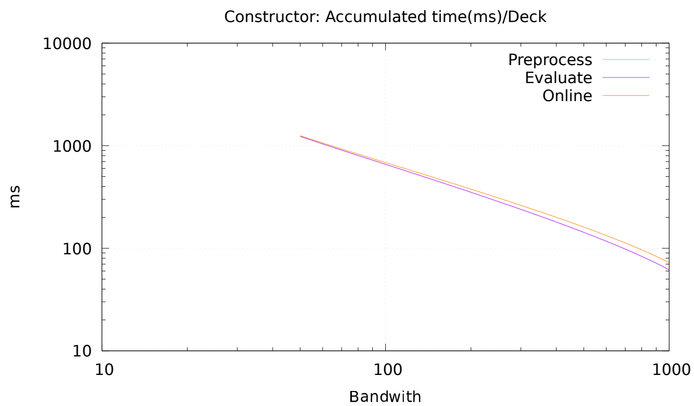


(b)

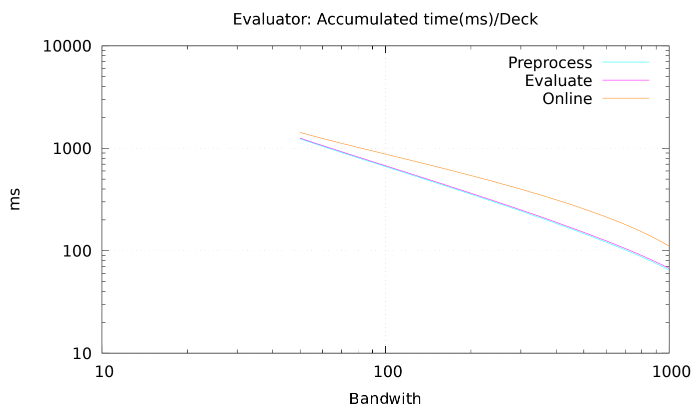


(c)

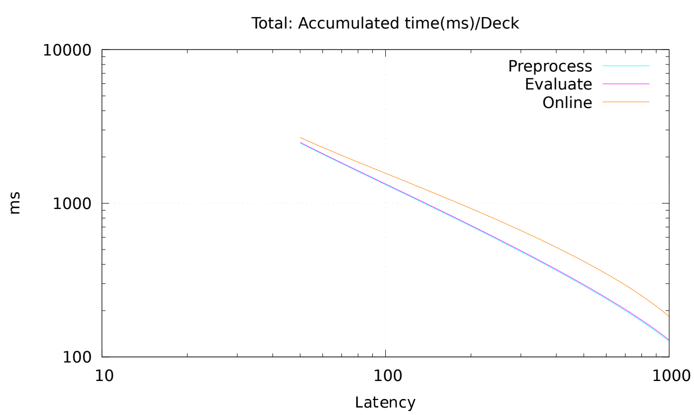
Figure 4.5: Delay: Comparison of *Constructor* and *Evaluator* in *ms*'s used when 1000 shuffles are done with different latency on the network. (a) *Constructor*: Accumulated time per deck shuffled on a duble logarithmic scale. (b) *Evaluator*: Accumulated time per deck shuffled on a duble logaritmic scale. (c) *Total*: Accumulated time per deck shuffled on a duble logaritmic scale.



(a)



(b)



(c)

Figure 4.6: Bandwith: Comparison of *Constructor* and *Evaluator* in *ms*'s used when 1000 shuffles are done with different bandwith on the network. (a) *Constructor*: Accumulated time per deck shuffled on a duble logarithmic scale. (b) *Evaluator*: Accumulated time per deck shuffled on a duble logaritmic scale. (c) *Total*: Accumulated time per deck shuffled on a duble logaritmic scale.

Chapter 5

Conclution

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.

A.1 Hardware

For compilations of the circuits with the *Frigate* compiler the following setup has been used:

```
OS:        Ubuntu 16.10/17.04
Processor: Intel i5-4210U CPU @ 1.70GHz
Cores:     2
Threads:   4
RAM:       12 GB
```

I did not encounter any problems or slow compilations of circuits using this setup.

For the testing of the poker game that setup was not sufficient as it could not handle more than 500 simuntainios shuffles. Therefore another setup up was used:

```
OS:        Ubuntu 16.04 LTS
Processor: Intel i7-3770K CPU @ 3.50GHz
Cores:     4
Threads:   8
RAMS:      32 GB
```

This setup allowed for testing up to 3000 simuntainious shuffles, which is the highest done in the testing phase. When going up to 4000 this setup ran out of memmory on the *Evaluator* site of the execution.

A.2 DUPLO

The *DUPLO* repository at GitHub can be found here [1].

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 here [2]. 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.3 Frigate

The *Frigate* repository on GitHub is a subrepository to *DUPLO* and can be found here [3].

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.2. 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 docker [4] and running the following command, in docker version `17.05.0-ce`;

[1] https://github.com/AarhusCrypto/DUPLO

[2] https://hub.docker.com/r/cbobach/duplo/

[3] https://github.com/AarhusCrypto/DUPLO/tree/master/frigate

[4] https://hub.docker.com/r/cbobach/duplo/

```
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.4 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.4.1 Circuit implemetation

The different circuit `.wir` files can be foun in the Github repository here [5].

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 in `count-gate-types.py` is the one used to compare the amount of gates types for the compiled circuits.

[5] `https://github.com/cbobach/speciale_circuit`

A.4.2 DUPLO implemetation

The poker repository for the implementation using the duplo framework can be found here [6].

Here the `CMakeLists.txt` file is the one used to overwrite the original file found in the *DUPLO* framework to allow for compilation of the poker *Constructor* and *Evaluator*. In the folder `duplo-mains` the implementations of these are located as `poker-const-main.cpp` for the *Constructor* and `poker-eval-main.cpp` for the *Evaluator*. In this filder thir shared functionality is found in the `poker-mains.h`.

Back in the main dir of the repository the docker files are found for generating the docker instanc of *DUPLO* used in appendix A.2 and A.3. This is the `Dockerfile.DUPLO` where as the `Dockerfile` is the one used for running the poker implementation. The `entry-point.sh` files is used to start the docker containers correct shuch that thay can run in the background. The docker image can be found here [7]. The containers can be started using the following commands in docker:

```
docker run -d -p 2800:2800 cbobach/duplo-poker --profile const -i 0
docker run -d -network:host cbobach/duplo-poker --profile eval -i 0
```

The `-d` flag tells docker that the continers should run detached. The `-p` flags tells docker to connect the host port 2800 to the containers internal port 2800. `--network:host` is the easy way to let the container have acces to the hosts network ports. These commands will play one hand of poker in the background, if more are required the `-f` flag can be used to specify which circuits should be used. To get the the right timings the `-n` falg is required together with the `-f` flag. This flag needs to reflect the nomber of simuntainus shuffles in the circuit.

Using the `-it` flag in docker instead of the `-d` flag allow for interactive rounds of poker if the `-i` flag is set to 1 instead of 0.

A.4.3 Test results

In this section a link to the repository on GitHub with all the generated statistics. Here all generated graphs and timings can be found. The repositor can be found here [8]

In the tables here the actual data used to generate the figure 4.2 and 4.3 in section 4.2 can be found.

TODO: Add log files to GitHub

TODO: describe test bash script

[6] `https://github.com/cbobach/speciale_implementation`
[7] `https://hub.docker.com/r/cbobach/duplo-poker/`
[8] `https://github.com/cbobach/speciale_thesis/tree/master/figurs`

Phase	Number of Parallel Shuffles 1	10	100	1000	3000
Preprocess	172140.31	26817.94	9380.99	7380.56	6219.38
Evaluate	122.84	122.84	122.84	122.84	122.84
Online	5.94	2.38	2.02	1.99	1.99
Total	172269.09	26943.16	9505.85	7505.42	6344.21

(a) *Constructor*

Phase	Number of Parallel Shuffles 1	10	100	1000	3000
Preprocess	193.75	36.59	19.15	17.57	17.50
Evaluate	0.11	0.10	0.10	0.10	0.10
Online	1.44	0.58	0.49	0.48	0.48
Total	195.30	37.27	19.74	18.15	18.08

(b) *Evaluator*

Phase	Number of Parallel Shuffles 1	10	100	1000	3000
Preprocess	172334.06	26854.53	9400.14	7398.13	6236.88
Evaluate	122.95	122.94	122.94	122.94	122.94
Online	7.38	2.96	2.51	2.47	2.47
Total	172464.39	26980.43	9525.59	7523.57	6362.29

(c) *Total*

Table A.1: Data sent: Comparison of *Constructor* and *Evaluator* in *kb*'s sent to the other party. (a) *Constructor*: *kb*'s data sent in different phases. (b) *Evaluator*: *kb*'s data sent in different phases.

	Number of Parallel Shuffles				
Phase	1	10	100	1000	3000
Preprocess	8202.12	918.74	193.32	116.73	106.89
Evaluate	100.82	10.22	1.18	0.30	0.30
Online	301.09	120.84	103.38	116.60	179.35
Total	8604.03	1049.80	297.88	233.63	286.54

(a) *Constructor*

	Number of Parallel Shuffles				
Phase	1	10	100	1000	3000
Preprocess	6619.65	823.37	187.27	119.74	118.28
Evaluate	129.49	17.91	5.43	5.44	9.58
Online	301.76	121.09	116.34	159.80	430.11
Total	7050.90	962.37	309.04	284.98	557.97

(b) *Evaluator*

	Number of Parallel Shuffles				
Phase	1	10	100	1000	3000
Preprocess	14821.77	1742.11	380.59	236.47	225.17
Evaluate	230.31	28.13	6.61	5.74	9.88
Online	602.85	241.93	219.72	276.40	609.46
Total	15654.93	2012.17	606.92	518.61	844.51

(c) Total

Table A.2: Comparison of *Constructor* and *Evaluator* in terms of time consumption in *ms* during framework calls. (a) *Constructor*: Time consumption in different phases. (b) *Evalauator*: Time consumption in different phases. (c) *Total*: Time consumption in different phases.

Primary Bibliography

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

[A2] Carmit Hazay and Yehuda Lindell. *Efficient Secure Two-Party Protocols: Techniques and Constructions*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.

[A3] 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.

[A4] 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

[B5] F. Yates R.A. Fisher. Statistical tables for biological, agricultural and medical research, 6th ed. http://hdl.handle.net/2440/10701, 1963.