Secure Distributed Poker using MPC

Christian Bobach, 20104256

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

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

TODO: abstract

Resumé

TODO: Resume

Acknowledgments

During the work of my thesis I have had the help of some great people. Both when it comes to the problem of finding the right solutions, but also when looking at the nifty little details. I would like to thank Kristoffer Arnsfelt, who has been a great help in the beginning of the process, when I was trying to find the best shuffle solution to use. He came up with the idea to look into sorting networks, because he was concerned with the security of the *Fisher-Yates* algorithm. Ni Ni Trieu has been a great help with learning to understanding the *Frigate* compiler. She has all ways been ready to answer my stupid questions fast, in a short precise way. I would like to thank Claudio Orlandi, for the well guided hand he has been giving me through the thesis. He has always been open hearted, and said what was on his mind regarding the project. Roberto Trifiletti should have my biggest thank as he proposed this exciting project to me many month ago, even before the *DUPLO* was a reality. He has supported me with all his knowledge during the project, from the early beginning to the bitter end. He has been there every time I needed someone to discuss a problem or solution with. Lastly but not leas I need to thank my wife for holding up with me, when I when on one of my coding sprees, and for her details for commas.
Thank you everyone.

Christian Bobach,
Aarhus, June 11, 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 arrangement 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 equally 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 implementation 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 for fills the right properties needed for an application as a poker game. I have studied different permutation algorithms and implemented them to compare them and see what effects they have on the underlying protocol.

1.1 Chapter Overviews

I will now give a short introduction to each chapter such that you as a reader know what to expect.

Chapter 2: In chapter 2 I cover the basics 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 shipped with the *DUPLO* project works. TODO: Introduce chapter duplo

Chapter 3: 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.

Chapter 4: 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 choices 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 simultaneous shuffles, the effect of network latency and the effect of bandwidth.

Chapter 5: TODO: introduce chapter on conclusion and proposals of further studies

TODO: Lim til f'rste kapitel

Chapter 2

DUPLO

In this chapter I will describe the *DUPLO* framework introduced in [4] and why this was chosen, to handle the communication and security of the poker game. I will shortly introduce the concepts of $2PC$ and *garbled circuits*. I will explain how the structure of the *DUPLO* protocol works, and give an introduction to what the different framework calls do. I will go over the security details of the protocol, to illustrate how this is guaranteed. Lastly, I will introduce the *Frigate* compiler which ships with the *DUPLO* framework to generate circuits for evaluation.

As it can be read in [4], the *DUPLO* framework is among the latest papers, where the efficiency of a two party computation(2PC) protocol, using garbled circuit, in a malicious setting is studied. In the *DUPLO* paper they claimed to perform better then any existing protocol. Their idea comes from the fact, that the two extreme variants of cut and chose protocols, when it is measured relative to circuit size. While they due perform well in one end of the spectrum, they due bad in the other. In the paper they propose a new approach to due cut and chose. The idea is to cut and choose sub-components of the circuit, and get an optimum somewhere in between the two extremes. The earlier protocols due cut and choose of either complete circuits or gate level. Their aim is to show, that the gate level cut and chose adds an overhead to the protocol, when the small components are soldered together again to constructing the garbled circuits for evaluation. At the same time, they try to show that when the number of sub-components goes up, there is a performance gain, when compared to cut and choose of complete garbled circuits.

As seen in section 7 on performance in [4], it is clear that the experiments done yields an optimal cut and chose strategy. This strategy differs from the earlier known possibilities. We see that the gain in terms of running time increases as the size of the circuit get bigger, which shows that the *DUPLO* protocol scales significantly better then those it is compared to.

Based on these observations and the fact that it is developed at Aarhus University, such that the people with knowledge of the protocol is close by, is the main factors for choosing to use *DUPLO* . The fact that *DUPLO* also supports the possibility of single and distinct wire opening was important when making the decision. Exactly, the opportunity to be able to due unique wire opening, is needed to handle unique opening of cards to one player, without the other player learning anything about the card.

Before continuing on the introduction of the *DUPLO* framework, I will explain the main ideas behind 2*PC* and *garbled circuits* in the next two small sections.

2PC: 2*PC* is a special case of *MPC*. I have used [3] as a reference, but many others due exist. In 2*PC* only two parties P_1 and P_2 participate in the distributed evaluation of the functionality $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$. The goal in a 2*PC* protocol is to allow 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, securely. When talking about securely evaluation, it is meant that the evaluation is to guarantee *privacy* and *correctness*. *Privacy* is to guaranteed, that no more than the output is learned from the computation. *Correctness* is the ability to guarantee, that the correct outputs is generated. When discussing *MPC*, we have to take *independence of input*, *guaranteed output delivery* and *fairness* into account, as these cannot be guaranteed en all setings. If we let m denote the number of parties in a *MPC* protocol and let t denote the threshold for the number of corrupted parties, then in the 2*PC* case $m = 2$ and $t = 1$, since each party P_i, where $i \in \{1,2\}$, trust it self. In a *MPC* setting *guaranteed output delivery* and *fairness* can be achieved for any protocol with a broadcast channel, with $t < \frac{m}{2}$. This implies, that non of these can be achieved in a 2*PC* setting, like the one used by *DUPLO* , because $1 \not< \frac{2}{2} = 1$. It is another case for *privacy*, *correctness* and *independence of input*, which can be achieved in a 2*PC* setting. This can be done, when the parties are given access to a broadcast channel and when we assume the existence of enhanced trapdoor permutations. It is important to remember, that these properties only hold for the computational setting of adversary powers.

This ensures, that the setting studied in the *DUPLO* paper can get *privacy*, *correctness* and *independence of inputs* from a 2*PC* protocol.

Garbled circuits: The *DUPLO* protocol uses, what is known as encrypted or *garbled circuits*. I use [3] as the reference for the introduction to *garbled circuits*. *Garbled circuits* is a construction, where the functionality f, is represented as a boolean circuit $\mathcal{C}$. The garbling of $\mathcal{C}$ gives the circuit the desired abilities as argued in the section on 2*PC*. We let $g : \{0,1\} \times \{0,1\} \to \{0,1\}$ denote a gate in $\mathcal{C}$. When garbling $\mathcal{C}$ it follows the garbling of all gates. Let the input wires of g be labeled w_1 and w_2, and the output wire w_3. Let k_i^0 and k_i^1 be keys generated for each wire in g, where $i = 1, \ldots, 3$, such that two keys are generated for each wire w_i. The idea is to be able to learn $k_3^{g(\alpha,\beta)}$ from k_1^α and k_2^β without revealing any of $k_3^{g(1-\alpha,\beta)}$, $k_3^{g(\alpha,1-\beta)}$ or $k_3^{g(1-\alpha,1-\beta)}$. Let g be defined by these values:

$$c_{0,0} = Enc_{k_1^0}(Enc_{k_2^0}(k_3^{g(0,0)}))$$

$$c_{0,1} = Enc_{k_1^0}(Enc_{k_2^1}(k_3^{g(0,1)}))$$
$$c_{1,0} = Enc_{k_1^1}(Enc_{k_2^0}(k_3^{g(1,0)}))$$
$$c_{1,1} = Enc_{k_1^1}(Enc_{k_2^1}(k_3^{g(1,1)}))$$

Where Enc is a private-key encryption scheme, that has indistinguishable encryption under chosen plain-text attacks. g is then represented by a random permutation of the values $c_{0,0}$, $c_{0,1}$, $c_{1,0}$ and $c_{1,1}$. Remember that the existence of a enhanced trapdoor permutation was a required assumption to get *privacy*, *correctness* and *independence of input*, in a $2PC$ setting.

The correct $k_3^{g(\alpha,\beta)}$ can then be generated by computing $Dec_{k_1^\alpha}(Dec_{k_2^\beta}(c_{j,l})$, for $j, l \in 0, 1$. Dec denote the corresponding decryption algorithm for Enc. It is required that for some message $m = Dec_k(Enc_k(m))$. If it is the case when decrypting, that more then one yields non-$\perp$, the protocol aborts, else define k_3^γ to be the only non-$\perp$ value. Because of the chosen encryption scheme k_3^γ, is the correct value with negligible probability. When this is done to generate a complete garbled circuit $\mathcal{C}$, where the description above is followed for each gate, it will results in a garbled circuit, representing f, that ensures *privacy*, *correctness* and *independance of input* for the evaluation.

2.1 The *DUPLO* framework

In this section I will introduce the different functions from the *DUPLO* framework and what they achieve. The *DUPLO* framework consist of two parties, a *Constructor* and an *Evaluator* with different roles during the protocol. The *Constructor*'s main purpose is to generates or construct the garbled circuits and send to them to the *Evaluator*. The *Evaluator* then verifies a number of these circuits, if this verification passes, the *Evaluator* trust that the remaining circuits are valid, and these are used during evaluation. If the *Evaluator* cannot verify the chosen circuits it aborts the protocol. The *Evaluator* main purpose is to evaluate these circuits.

The overall construction of the *DUPLO* framework consists of different functions. These functions guarantees the right communication is sent between the two parties. Because of the construction of the functions it is necessary to call them in a predetermined order, this is to ensure that the correct information is at the parties at the time when needed. When a framework defines a set of functions to be called in a given order, one function could have been used to handle the same functionality. In *DUPLO* the have chosen to split it in to different functions such that local computations can be done in between these framework calls. This allows for a more modular approach when sing the framework.

When we interact with the framework, and run the protocol, we need to create a *Constructor* and an *Evaluator*. When these are created they read the circuit file, specifying the functionality desired to be computed during the protocol. In our case this will be the shuffle algorithm which will be introduced in chapter 3.

Once the *Constructor* and *Evaluator* are created they run the framework function calls in parallel. If they get out of sync the protocol will abort. First the two parties call the framework function `Connect`. This sets up a connection between the two parties. In this case it is the *Constructor* hosting the service, and the *Evaluator* connects to this. When the connection has been established, they each make a call to the `Setup` function. This call initializes the commitment protocol, which will be used between the parties during the protocol. The next specified by the framework is the call to `PreprocessComponetType`. This call takes n and f as inputs, where n is the amount of garbled circuit to produce, and f is the functionality that will be evaluated. This call then generates n garbled representations of f, that can be securely evaluated.

Then a call to the `PrepareComponents` function is specified. This takes i as input, which is the amount of input authenticators to generate. These authenticators is used to securely transfer the input keys from the *Constructor* to the *Evaluator*. This function also ensures that all required output authenticators are attached. The authenticators guarantees that only one valid key will flow on each wire of the garbled components. If not the *Evaluator* will learn the *Constructors* input.

After this, the `Build` function call is done. This function takes a boolean circuit $\mathcal{C}$ as input, representing the desired functionality of the computation. The function constructs the garbled circuit, which is to be evaluated later. The `Build` call ensures that the function components specified by the $\mathcal{C}$, is soldered together, such that they compute the functionality specified. This is done in such a way that the output wires from one sub-function is soldered together the right input wire of another sub-function.

The next call is then made to `Evaluate`, which takes x_1 and x_2 as input. Here x_1 is the input to the computation from the *Constructor*, and x_2 is the input from the *Evaluator*. The call then evaluates the garbled circuit given these two inputs and yields a garbled output of the desired functionality $f(x_1, x_2)$. When the parties then hold a gabled output from the local evaluation, a call to `DecodeKeys` can be made and the output of the functionality will be revealed.

Because the evaluation of circuits, in the *DUPLO* protocol, allows for openings of distinct output wires to only one or both parties. It allows us to reveal cards to only one player and other cards to the other, without the opponent learning anything about the cards. The decision to split-up the `Evaluate` and the `DecodeKeys` functions will allow us to open different output wires in different rounds. This will help us to achieve a good round complexity when creating our poker implementation. This decision can also be used if the output of one circuit evaluation, will be used as input for another, such that the garbled output can be used as new inputs.

2.2 Security

In this section, I will introduce the security of the protocol to show that the players playing a game of poker, using an implementation with the *DUPLO* framework will have *privacy*, *correctness* and *authenticity*. With *privacy* is meant that the opponent can not learn more then supposed to. The play will be guaranteed *correctness*, meaning that if the garbled evaluation of the circuit is done without aborting, then it is guaranteed to give the right output. *DUPLO* ensures *authenticity*, such that it is not possible for a player to due evaluation of the garbled circuit, on other input then the one provided by the parties.

The proof of security for the protocol is done using the Universal Composition(UC) framework, a nice introduction to the framework can be found in [1]. The UC framework is an easy digested abstract, protocol proof technique, which allows for sequential predefined interaction between parties using actions and reactions. It has a modular approach 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, and $\mathcal{F}_{OT}$ for the one out of two oblivious transfer, used by the protocol. These functions are then used to prove *privacy*, *correctness* and *authenticity* of the protocol.

In the section on protocol details in paper [4], appendix A, they describe and analyze the protocol. Here structuring the main protocol and going into details on how *privacy*, *correctness* and *authenticity* are guaranteed through the different function calls in the protocol. The proof end up being rather complex as the main structure consist of 8 sub-functions. Where multiple of these is a combination of further sub-function. All these functions are proved to satisfy the properties mentioned above. During the analysis of these functions, they end up proving correctness of soldering and evaluation, of sub-circuits. They end up proving robustness of the key authenticator buckets, evaluation of the key authenticators, input of the *Constructor* and *Evaluator*, evaluation of sub-circuits and the output of the *Evaluator*. This culminate in the theorem, cited here as theorem 1, proving robustness of the protocol. The theorem guarantees that if the *Constructor* is corrupt and the *Evaluator* is honest, and the protocol does not abort, then the protocol completes holding the before mentioned properties, except with negligible probability. As known when using MPC protocols, when half or more of the participating parties are corrupt, we can not guarantee termination of the protocol.

In appendix B they prove the fact, that the protocol is secure against a corrupt *constructor* or *evaluator*. Since it is a $2PC$ we may assume that one of the parties are honest, as the parties trust them self's. 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 $\mathbf{C}$, and a honest *Evaluator*, denoted $\mathbf{E}$. The simulator $\mathcal{S}$ plays the role of $\mathbf{E}$ in the protocol, but is not given access to the inputs $x_{\mathbf{E}}$ of $\mathbf{E}$. Instead $\mathcal{S}$ has access to an oracle $\mathcal{O}_{x_{\mathbf{E}}}(\cdot)$ containing $x_{\mathbf{E}}$. The simulator $\mathcal{S}$ might contact the oracle $\mathcal{O}_{\S_{\mathbf{E}}}$ giving input $x_{\mathbf{C}}$ and in return learn $y_{\mathbf{C}}$, as if the evaluation of the functionality was done with $x_{\mathbf{E}}$ and $x_{\mathbf{C}}$ as input.

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

Theorem 1 *If constructor* **C** *is corrupt and evaluator* **E** *is honest, and the protocol does not abort, then the following holds with negligible probability. For each input gate with 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 with id′,* **E** *holds* $k_{id'} = k_{id'}^{y_{id}}$ *where* y_{id} *is the plain text value obtained by evaluating circuit* $\mathcal{C}$ *on* x_{id}*. The probability of the protocol aborting is independent of the inputs of* **E***.* [1]

$\mathcal{S}$ is constructed such that it first constructs $x_{\mathbf{E}} = \mathbf{0}$, as the zero input vector for **E**. It then inspects the commitment of the input gates of **C** and learns k_{id}^0 and Δ_{id}, where *id* is the gate identifier. From these k_{id}^1 is computed. By theorem 1 $k_{id} = k_{id}^{x_{id}}$ can be retrieved. This defines the input $x_{\mathbf{C}}$ for **C**. $\mathcal{S}$ then calls $\mathcal{O}_{x_{\mathbf{E}}}(\cdot)$ with input $x_{\mathbf{C}}$ and learns $y_{\mathbf{C}}$ from $\mathcal{O}(x_{\mathbf{C}})$. If $y_{\mathbf{C}} = \perp$ 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 simulation aborts with the same probability. When they due not abort the key returned to **C** is the same as **E** would have sent, except with negligible probability.

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 appendix B.2 in [4].

While I went through the proof of *DUPLO* I found typos in both section *B*.1 and *B*.2. Here they had switched around on the corrupt and honest party when they recall the task of the proof. These typos has been announced to them and a fix will be made. When reporting back to them on findings like this, I add value to their work by helping to secure a better end result.

2.3 Frigate the *DUPLO* Circuit Compiler

In this section, I will introduce how the new version of the *Frigate* circuit compiler works. First I will introduce which requirements there are to the compiler to get it up and running. Then I will describe how the programming language used to generate the circuits works. Lastly I will cover the bug I found in the compiler, how it was done and what results this had on the *DUPLO* project.

First of all, when installing the compiler some special versions of libraries are required. These laborers are not the newest updates and therefor they are required to be hold back. Following the instructions in the installation guide, which can be found via the appendix A.3, and some amount of internet search, I was able to get it up and running. The hardware I have been using to run both *Frigate* and *DUPLO* , can be found in appendix A.1. The standard version of *flex* and *bison*, which ships with system I used, is newer then the one required by *DUPLO* . This required, that an older versions was installed and kept back, such that no updates was made to no compatible versions. As the compilation

[1]This theorem is a cited from [4], with small textual modifications. It can be fond as theorem 2 in appendix A.

```
function SEED xorSeeds(SEED s1, SEED s2) {
    SEED s;

    for (int i = 0; i < n; ++i) {
        s{pos:card_size} = s1{pos:card_size} ^ s2{pos:card_size};
    }

    return s;
}
```

Figure 2.1: wir-file: An example of how *XOR* could be done in an itterativ way, in blocks of card size.

of circuits are done once, and due not effect the compilation of the *DUPLO* implementation, this is not a complete deal breaker for using *Frigate*. For future circuit compilations, this can be done using `docker` via the duplo image linked in A.4.2 where the right versions of *flex* and *bison* are installed. When I had *Frigate* up and running I could start looking into the documentation of how it worked.

The *DUPLO Frigate* compiler came shipped with the documentation for the first version and was not updated, when the functionality was extended to cover the *DUPLO* framework. This should not be expected as the functionality of the compilers input language was not changed, but it resulted in some time consuming trial and error for my site, as the documentation was not well written and specified. The documentation is very short and not that precise. I have not in resent time worked with circuit compilation in the way *Frigate* required, where a higher level abstraction language was used to generate a circuit. This resulted in some hard earned experience on very small examples.

Some of the most important and different things to take into account, learned from my time using the *Frigate* compiler is, the possibility to due wire access. It allows you to specify exactly which and how many wires should be represent in a certain value. This is bone by the following syntax `s=s1{index:size}` in the *Frigate* `wir` format, the can be seen in figure 2.1. This allows for single bit inputs to be translated into higher level representations, like variables. This gave some occasional frustrations, as it is not possible to access wires based on variable inputs. As these variable inputs cannot be predetermined by *Frigate*. This makes perfect sense, since the compiler cannot know which wires should be used for the representation. This is due to the fact that, circuits are static representations and therefore to overcome, variable wire access the compiler needs to know all possible wire accesses on compile time. It the compiler could due this, it could generate components for each of these possibilities. If this approach was chosen, it will result in a huge blow up in the complexity and number of gates in the circuit. Therefore it makes perfect sense that this feature is not implemented. A second point to remember, when working with *Frigate* is that it due not allow for more than one level functions. This result in some restrictions when programming for circuit generation, compared to normal programming languages. This restriction makes it harder to create small functions with one single specific focus. This could be the case for generating a conditional swap gate, which we will due in section 3.3, or something similar. Because of this restriction it is nor possible to generate recursive functions. Because circuits are a static representation, the problem of recursive functions cannot easily be handled, since the size of the circuit can not variate based on inputs, as it is generated ahead of time. It is another case when looking at one level functions. I due not completely understand why this restriction was made. It may be because of the way they stitch the functions together when generating the circuit. It should be possible to handle multiple level functions via some intermediate representations. As long the depth of the function calls are static. Another small problem I had, which was not stated anywhere in the documentation, is that *Frigate* only allows for assignments in the main method through function calls. This gives completely sense when looking into the *DUPLO* circuit file.

Otherwise the programming 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`. Other wise it allowed for definitions of constants, types, structures and imports just like in *C* which allows for some easy readability. The programming language format used by *Frigate* is called `.wir`.

When the desired functionality has been implemented using the `.wir` format described above used by *Frigate*, the compiler can be used to translate it in to a format the *DUPLO* framework can handle. This can be done from inside the compiled *DUPLO* framework, if the right versions of the libraries are installed. The compilation is done by running the following command:

```
./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 a bunch of different files with different file extensions. The file we are interested in is the file with the `.wir.GC_duplo` extension. This is the file the *DUPLO* framework uses as input.

In this section I will cover the bug found in the *DUPLO* updated version of the *Frigate* compiler. I will cover how it was found and which fix it resulted in. First of all the *DUPLO* framework has a functionality, that allows for evaluation of a circuit files without setting up both parties, the *Constructor* and *Evaluator*. This possibility gave the opportunity to specify the input values, from both the *Constructor* and the *Evaluator*, for the computation of the circuit. This allowed me to learn the outputs, in a fast return cycle. This possibility allowed me to study the implementations of the *Frigate* compiler and learn the programming language. This opportunity allowed me to see how the algorithms, found in chapter 3, behaved compared to expected while

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*

implementing them. First I tried to implement the *Fisher-Yates* algorithm in the `.wir` format. The *Fisher-Yates* algorithm is is described in details in section 3.2, and can be seen as algorithm 1. During the implementation, I encountered some problems, when trying to generate the seeds from the to parties to the shuffle algorithm. Because I did not have any prior experience working with the *Frigate* compiler, I was not able to determine if the problem arose from the implementation of the algorithm, or the compiler it self. At first the focus was on the implementation, as my lack of experience working with *Frigate* easily could lead to errors. After a lot of modulation and debugging, it was clear that something was wrong with the compiler. During the debugging process, I created a framework that allowed me to test the different modules of the implementation one by one. It became clear during this process that something was off when using the modulo reduction. The modulo operator `%` is specified in the documentation and should therefore be possible to use. Since the compiler I used was a modified version of the original *Frigate* compiler, the posibility was that a bug could have been introduced when adding the new *DUPLO* featurs. Therefore the original *Frigate* compiler was installed to test, if that implemetation introduced the same problems with my algorithm. This idea introduced a new problme, since the original version of *Frigate* due not support the *DUPLO* circuit format. Luckely for me the *DUPLO* framework interpeter has supports for another circuit file format known as *bristol*[2]. To be able to use this file format, I wrote a parser that took the output format from the original *Frigate* compiler, and translated that into the *bristol* file format. This gave me the possibility to evaluate two different compiled circuit files that could tested to see if the problem also was in the original version of *Frigate*. Based on these two formats I created a test framework, to see if the two circuits generated the same output. This showed that there was a difference in the results produced, especially in the test case wher the modular reduction was used. When taking a deeper look at the problem, it became clear that not all gate types was generated during compilation, using the *DUPLO* version of the *Frigate* compiler. I was lucky that the modulo reduction triggered one of these gates. It was then discovered that multiple gates was not supported as they were forgotten during implementation.

This discovery resulted in a fix of the new version of the *Frigate* compiler and a complete change in the representation format used for the circuits. Now all gates in the *DUPLO* format has two input and one output wire. Because

[2]The *bristol* format can be found at
`https://www.cs.bris.ac.uk/Research/CryptographySecurity/MPC/`

of this shift in supported gate types the amount of input and output wires was removed from the file format. The representation of gates changed, from the more readable type like XOR, to a truth table friendly type like 0110, for the XOR gate. This change in representation ensured that all 16 gate types was implemented. The 16 gate types can be seen in table 2.1. The representation of the two constant **0** and **1** are handled as special cases, because 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 implementation of the *Fisher-Yates* algorithm, I have contributed to the *DUPLO* project by reporting my findings and allowed them to fix this problem before publishing. This has helped to secure a stronger overall research product.

They have made the changes to the compiles as mentioned above, but has not updated it to support modulo with a divisor that is not the power of 2. This behavior of modulo is not mentioned anywhere in the documentation for the original or the updated compiler. When I discovered the problem of the modulo operator I used some time to research it, and see if a solution could be made to implement this functionality easily. During the research, I noticed that implementing the modulo reduction to some power of 2 is simple. While the implementation of modulo with a divisor different from 2 does not seem to be trivial. The problem was therefore left unfixed. In my implementation of *Fisher-Yates* I use a workaround to overcome the problem, which is explained in section 3.2.

In this chapter I have covered the main idea of *DUPLO* . I have argued for the security and what that will give of guarantees to my implementation. I have introduced the $Frigate$ compiler, and the problems I have encountered when using it. I have claimed my findings and added value to the *DUPLO* project in multiple ways.

In the next chapter I will introduce the algorithms, used to shuffle cards in the poker implementation.

Chapter 3

Shuffle 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.

In the first section I will introduce the poker game used during this project. I will introduce som optimizasions that can be done to reduce the amount of cards that needs to be shuffled. I intro duce another type of poker then the one studdied, to have a comparison.

The permutation algorithms studied in this project, have the purpose of shuffling card decks. Here it is important to chose a shuffle algorithm that guarantees, that the correct amount of permutations is reached. If the right amount of permutations is reched we can ensure a unifor distribution on the outcome.

The first algorithm studied in this chapter is the Fisher-Yates algorithm which was introduced in [5]. It may also be known as Knuth shuffle, which was introduced to computer science by R. Durstenfeld in [2] as algorithm 235. This algorithm uses an in place permutation approach and gives a perfect uniform random permutation. This algorithm is introduced in section 3.2 and can be seen in pseudocode as algorithm 1. When takling about an in place permutaiton algorithm, a algorithm with only one data representation to hold the values to be shuffled is meant. Another normal approch is to use two data representations where the values are moved between these representations.

The second algorithm proposed uses ideas from shuffling networks and [6] 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 3.3 and can be seen in pseudocode as algorithm 2. This algorithm yields a perfect uniform permutation.

These shuffle algorithms are optimized to fit to the poker setting, which will be introduced in the section 3.1. This is done such that it only shuffles the required cards and not the whole deck. Resulting in a smaller circuit when this is used in the benchmarking.

The implementations of the shuffle algorithms will be discussed in section 3.4, where the choises made during implementation will be explained.

In the last section 3.5 I will compare the algorithms implemented. Here I will explain why I chose the algorithm I did, to use in the benchmark of the poker game. In this section another type of shuffling networks called Bitonic shuffle network, will be introduced and discussed shortly. No implementation of this type of shuffle network was done.

I will introduce the poker game used in this thesis, and another to compare the differences. This will be used during the thesis to reflect upon how another type of poker game would have effected the outcome.

3.1 Poker the Game

I will in this section introduce the poker game. Poker is a card game played in various rounds, where the players draw cards and places 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 to use during the project. The variant chosen to use is known as $five\ card\ draw$ poker. In this project the game will be played between two parties, as described in chapter 2. 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. At last a new betting round is performed, before the cards is revealed and a winner is declared.

The $five\ card\ draw$ poker variant is played with a deck of 52 cards. This poses some requirements for our shuffling algorithms. Since there are 52 cards in the deck, this yields 52! different permutations required by our shuffle algorithm. We require a shuffle algorithm that can produce exact these permutations to represent all the possible shuffles of the card deck. Because 5 cards are dealt to each party in the first round and that they at most can change all the cards in the second round, only the first 20 card of the deck is needed per game. Therefor it is enough for the algorithm to produce a complete shuffle of these 20 cards 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.

Another variants of poker require a different amount of cards per game. One example could be if the game included tree players instead of two, then 30 cards of the complete deck would be needed. Another example could be the $Texas\ Hold'em$ variant, which is played by dealing 2 cards to each player and placing 3 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

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
```

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 deck is needed. The optimizations done when only some cards out od the total amount is used, is also known as an m out of n permutation.

From here on when talking about a poker game, the $five\ card\ draw$ poker variant will be the reference, otherwise it will be specified. This is especially interesting when looking for optimizations in the shuffle algorithms, which will be introduced in 3.4 and when they are compared in section 3.5. When coming to chapter 4 the nomber of cards dealt will have effect on both, the amount of data sent and the time used by the protocol.

The poker game which will been implemented during the project is now introduce. In the next couple of sections the shuffle algorithms will be introduced, how they were implement and a comparison will be done. Based on this comparison a choise is done on which will be used during benchmarking. During the sections the different optimizations will be explained. These optimizations will reduce the overall size of the circuits.

3.2 Fisher-Yates

The *Fisher-Yates* algorithm can be seen as 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 a given index with the value specified by the swap value of the second array. Think of the input to be shuffled as a card deck, where you take the top card of the deck and swap it with another card at a position defined by the swap value.

This implies that the algorithm takes two inputs of the same size where the one is holding the values to be permuted, *deck* with n vlause $card_i$, for $i = 0, \ldots, n$. The other holding the values for which the different $card_i$ in the *deck* is to be swapped, *seed* with n values $seed_i$. It is important that $seed_i$ is chosen correctly. The algorithm states that $seed_i$ should be chosen from interval $[i, n]$. This yealds exactly the number of permutations required, as $card_1$ has exactly n possible swap posibilities. $card_2$ has $(n-1)$ possibilities and so forth. The last $card_n$ is dertermined by all the other swaps. Since $seed_i \in [i, n]$ we have $n!$ possible permutations because i runs from 1 to n. This is exactly the desired result, as discribed in section 3.1.

If $seed_i$ contained in *seed* is not chosen form the right interval, but instead is chosen from 1 to n, for all cards, we would end up having a skew on the probability distribution of the different permutations. Because $card_i$ in this case has n possible swaps, this yields n^n distinct permutations. This introduces an error into the algorithm, as n^n is not divisible by $n!$ for $n > 2$ and can therefore not yeald the desired $n!$ permutations. This results in a non uniform probability. If $seed_i$ instead is chosen from $]i, n]$ such that the index i is not in the interval, then an error is injtroduced to the algorithm, as the empty shuffle is not possible. In other words it is not possible to get the same output as the input. This error results in $(n-1)!$ permutations, which is neither devisible by $n!$, and therefor cannot give the desired uniform propability.

As described in section 3.1 no more then a permutation on the first 20 cards is needed. This has the effect that only $\frac{52!}{32!}$ specific permutations of the total 52! permutations is needed. Optimizing the *Fisher-Yates* algorithm to due a m out of n is streaght forward. Instead of running through n swaps indicated by the size of *seed* it is enough to run through m swaps. This yealds that hte size of *seed* only need to be 20 and therefore, the for-loop seen in algorithm 1 on line 8 needs to run fewer iterations. Those giving us a full permutation on the first m indexes of *deck*. This is because of the following fact:

$$\frac{n!}{(n-m)!} = \prod_{i=n-m}^{n} i$$

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 the inputs specified in the figure yealds the first 9 cards of the shuffle 1, 52, 14, 20, 10, 37, 9, 33, 6 as output. Here it is interesting to notice, that 37 occures twice in *seed*. Because the algorithm permuts the *deck*, with the seed value 37, 37 will not end up in the output twice. The first case

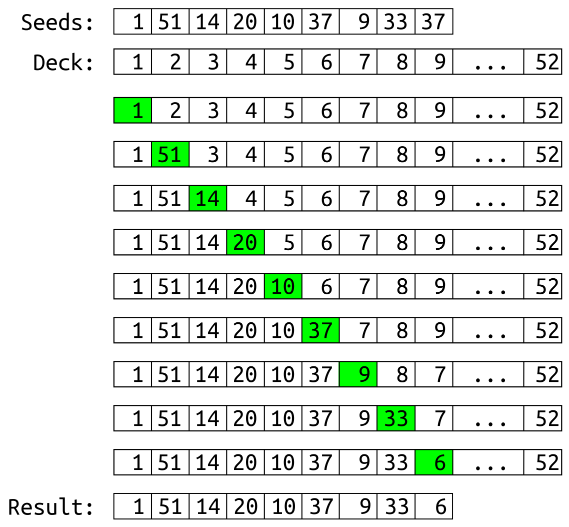


Figure 3.1: Fisher-Yates algorithm in action: In this figure a 9 out of 52 shuffle has been completed to ilustrade how the algorithm works. First 1 is swpaed with 1. Then 2 is swaped with 51. 3 with 14. 4 with 20 and so on until the first 9 numbers has completed a full permutation. Resulting in 1, 51,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

---

where the seed value 37 is used 6 and 37 is swapped in the deck. Resulting in 37 to be the sixth card in the output deck. The second time 37 is used as a seed value 6 is the swaped in as this was now located at thet spot in the deck. This illustate that it is possible for a *card* to be swapped multible times and therfor can end up on another index then specified by the first swap.

In the next section I will introduce the concept of shuffle network and the algorithm 2, which will be denoted *Conditional-swap* .

3.3 Shuffle Networks

Shuffling networks or permutation networks have a lot of resemblance with sorting networks, becuase of their structure. 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 placed. 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 those found in [6] as figure 3.

Applying such a shuffle network in the setting of a poker game is simple, because of the main structure. 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 different shuffle algorithms that

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 $i \in [1, \frac{n^2}{2}]$.

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

can be implemented using shuffle networks, such as *bubble sort*, *bitonic sort* and others. The one I have looked into and implemented, builds on ideas from [6], where they introduces the conditional swap gadget and the well known *bubble sort*. The *bubble sort* algorithm was chosen because of its simplisity and the ease of optimizing it to a m ot of n shuffle algorithm. This algorithm will be donoted *Conditional-swap* , because of the conditional swap gadget used in the network.

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

Conditional Swap: [t]

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 forth, until the last 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 input wires at the adjacent swap gadget. This is illustarated and can be seen 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 the first value on the first output wires is determined by the first layer of swap gadgets. Resulting in the first input

$card_1$ has n places to go, since the $0 \dots 0$ input strig will leave$card_1$ in place and the input string having 1 as input for the first gate in each layer will result in $card_1$ ending up on the last output wire. The second layer determines which output will be on the second output wire and so on. Continuing this way untill, reaching the last, layer where the two last outputs for the $(n-1)$ and nth wire will be determined. This gives us a shuffle algorithm with a perfect shuffle and $n!$ different permutations as desired. This is once agian due to the fact of that $n! = \prod_{i=1}^{n} i$.

If each layer of the swap gadgets are not decreasing by 1 in terms of the number of swap gadgets, as seen on line 12 in the pseudocode in algorithm 2, because i encreses by one on line 11. The algorithm suffers the same problems as *Fisher-Yates* , of producing n^n permutations, which is not devisible by the decired $n!$ permutations, as explained for the *Fisher-Yates* algorithm. This resulting in a skew of the probability on the different permutations, such that the propability of each permutation is no longer uniform.

Some optimization can also be done to this algorithm, since we only need a m out of n permutation. This can be done by letting the outer loop of algorithm 2, on line 11, run for m iterations instead of n. This yields n possible output wire positions for $card_1$, $n-1$ for $card_2$ and so forth, until $n-m$ values for $card_{n-m}$. This gives the exact amount of permutation desired for the optimized algorithm. Resulting in $\frac{n!}{(n-m)!}$ permutations, which is enough for the poker implementation used, as described in section 3.1.

In figure 3.2 a run of *Conditional-swap* algorithm can be seen. Here a 9 out of 52 variant is illustrated, since an output of 20 is hard to fit on the page. It can be seen that the input *deck* is sorted, which holds the values to be shuffled and *seed*, a bitstring indicating if two values should be swaped. The first 52 bits of the *seed* decides the value on the first output wire. In this run the first input value $card_1$ will also be the first output value. The next 51 bist from *seed* indicate, that 51 should be swapped all the way accross the network to the second output wire. This yealds that all cards 51 passed on its way, to the second output wire, will now be swapped one wire to the right, such that $card_{n-1}$ is now at output wire n. That is why the third output value 14 starts at wire index 15 and the fourth output wire with value 20 starts at wire index 21. The algorithm continiues like this until it outputs the first 9 cards, shuffled as 1, 51, 14, 20, 10, 37, 9, 33, 6.

In the next section the implementation of the two algorithms, *Fisher-Yates* and *Conditional-swap* will be described.

3.4 Circuit Implementation

In this section I will describe how the cork of the implementation of the *Fisher-Yates* and *Conditional-swap* algorithm was done. Because the *DUPLO* protocol was chosen to use in this project, the first hurdel was to generate the circuits,

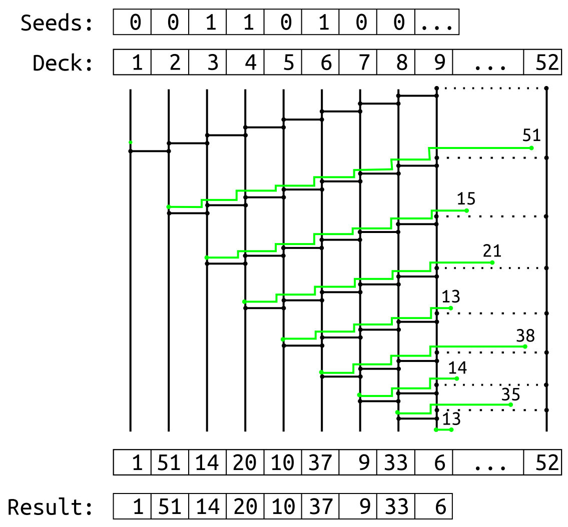


Figure 3.2: Conditional swap algorithm in action: In this figure a 9 out of 52 shuffle has been completed to illustrade how the algorithm works. Each bit in the *seed* indicates, if a gadget should swap the two incomming values. Since the size of *seed* is 423 bit long I have tried to ilustrate which wire each value is located at, by the values at the end of the line, before it is moved in a layer. This yealds the following output for the first 9 cards: 1, 51, 14, 20, 10, 37, 9, 33, 6.

---

that should handle the shuffling of the cards. As mentioned in section 2.3, implementing algorithms in boolean circuits become rather complex, when going beond small toy expamles. Luckly, the *DUPLO* project came shipped with a compiler for generating circuits with the right format. Therefore, before staring to implement the shuffle algorithms, I read up on the documentation for the compiler, this can be found by following the descriptions in appendix A.3. The documentation is from the first version of the *Frigate* compiler, which was extended during the *DUPLO* project, to generate the decired *DUPLO* format. An introduction to the *Frigate* compiler can be found in section 2.3. The codebase for *Frigate* can be found in appendix A.3.

Now going in to the details of the two algorithms implemented. The codebase for the implementaions can be found via appendix A.4.1. In the implementations the algorithms five different functions was implemented as modules, these modules could then be stiched together to give the desired functionality. Both shuffle algorithms *Fisher-Yates* and *Conditional-swap* make use of the module with the `initDeck` function.This function initializes a hardcoded representation of a sorted card deck, which is to be shuffled, when evaluating the generated circuit. This is done by a for-loop inserting the bit values, for each card, on the right wires. In the implementation 6 bit variables are used for the representation of each card, as 6 bits allow for 64 different representations, since $2^6 = 64$. This is more then enough representations to create a unique value for each *card* in the *deck*. It is importand to notice, that the variable indexing the

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.

---

start position of every *card* needs a representation with at least 9 bits, to be able to hold the correct value. Because the representation of a *card* is 6 bits and there are 52 cards in a *deck*, the representation of the *deck* has size $6 \cdot 52$, which is 312 bits. For the index starting position of each card to be able to hold the right value at least $\lceil \log_2(51 \cdot 6) \rceil = 9$ bits are required. The 51 come from the fact that the indexing starts at 0. If only 6 bits were used only wires up to position 63, could be assigned a value and not all 312. Therefor 9 bit is used for this variable.

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 cale to `initDeck` is done, shuch that the result for the two functions can be fead into the shuffle algorithm. 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. Following the structure of algorithm 2, ensure that we are ending up with an algorithm producing the right amount of permutations. This followings from the discussion made in section 3.3. Using the optimizations proposed in the same sections, yealds a clear reduction in the number of gates in the circuit. This can be seen in table 3.1, where the optimized and non-optimized versions are compared. The most interesting part is the 62% reduction in the number of non-free gates in the optimized version. Since these are the gate adding the most overhead to the *DUPLO* protocol.

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 3.2. At first each of the bit input strings are splitted up into representations of 6 bits, such that each holds 52 values, where i represent the card index from 1 to 52. These will be denoted $seed_{C_i}$ for the *Constructor*'s inputs and $seed_{E_i}$ for the *Evaluator*'s. These input reprsentations are added, such that $seed_i = seed_{C_i} + seed_{E_i}$. This ensures, that the value $seed_i$ is at most $2 \cdot (2^6 - 1)$, because of the representation. Since the addition of two 6 bit values cannot be guaranteed to fit inside another 6 bit value, a representation with an extra bit should be used to store the resulting value. One extra bit is enough since $2 \cdot 2^6 = 2^7$. The idea was to use a modulo reduction on $seed_i$ to gurantee, that $seed_i$ was inside the intervall

$[0; 52 - i]$, which is discussed in section 3.2. In the implementation we use this intervall, which is a slight deviation from the original wone, which was $[i; n]$. This is due to the fatc that, the modulo reduction implemented in $Frigate$ only supports divisors, which are powers of 2. Therefore could modulo reduction not be used. This is because $seed_i$ can not be guaranteed to have this property, since $3 = 1 + 2$ is not a power of 2 and 1 and 2 are both possible representations for eith $seed_{C_i}$ or $seed_{E_i}$. Therefore another solution was chosen. Since the input $seed_{C_i}$ and $seed_{E_i}$ is assumed to be in $[0; 52 - i]$, the solution was to subtract the boundray $I_u = (52 - i)$ of the interval from $seed_i$, if this exceets I_u. Doing it this way, yealds a $seed_i$ in $[0; 52 - 1]$. This is due to the fact, that $seed_{C_i}$ and $seed_{E_i}$ can at both atmost be I_u. This ensures that $seed_i \leq 2 \cdot I_u$. $seed_i - I_u$ is therefore 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 noteworthy to mention, that all values have had an unsigned representation until now. But since the comparison of two values need a signed representation, as stated by the documentation. $seed_i$ was converted into a signed value, by adding a 0 bit to the most significant bit of the representation. This workaround ensures that the *seed* given to the `shuffleDeck` function has the right form. This is based on an assumption that the inputs from to `correctSeed`, $seed_{C_i}$ and $seed_{E_i}$, are inside the intervall $[0; I_u]$. This is not guaranteed as 6 bits can hold the value $63 = 2^6 - 1$.

The second function used in the *Fisher-Yates* implementation is the same as in the case for the *Conditional-swap* algorithm, where the `initDeck` function is called. This is to initialize a hardcoded representation of the *deck* in the circuit. It is completly fine to hardcode this representation since the algorithm generats a uniform permutation. The last function called is the `shuffleDeck` function, which is different from the one from the *Conditional-swap* algorithm. This function consists of an outer for-loop, that runs through the cards $card_i$ of the deck and an inner loop that generates swap gadget. In algorithm 1 we only see one loop. The addition of the second loop is a way to handel the problem of circuits being a static representation, as disscused earlier in section 2.3, as it is not possible to assign a wire a value based on a variable input. The inner loop therefore generates conditional swap gadgets that ensures the posibility for the correct swaps. This yealds $52 - i$ swap gadgets in layer i, for $i = 1, \dots, 52$. The outer loop ensures the generation of layers holding the composed swap gadget from the inner loop. The composed swap gadgets are generated such that for each $card_i$, it can be swapped with any other $card_j$, where $i \leq j \leq 52$. Therefore is the composed swap gadget a composition of $52 - i$ desitinct conditional swap gadgets, as thoes used in *Conditional-swap* . This gives the desired propability, as described in section 3.1.

The *Fisher-Yates* implementation can also be uptimised as decsribed in section 3.2. This was therefor implemented and both the optimized and non-optimized version can be see in table 3.2. This gives a good overview of the improvements of the size of the circuits. When comparing the two versions we see a optimization of 40% decresae in the number of non-free XOR gates. Which is a desent amount of improvements.

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.

In this section, I have introduced the implementations of the shuffeling algorithms studied and compared them with their optimized versions. In the next section I will compare the two optimized algorithms and discuss their differencies and make a choise on which one I will use to benchmark on in section 4.2.

3.5 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 chose, which one to continiue with in the benchmarking 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.

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 $i = 1, \ldots, 52$. 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 see 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 bits 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 gadgets, the *Conditional-swap* algorithm constructs swap gadgets. Eventhoug the algorithms has different approches to how the conditional swap gadgets are generated, they end up generating the same amount. This is because *Fisher-Yates* generates 20 composed gedgets each consisting of $52 - i$ swap gadgets, for $i = 1, \ldots, 20$. 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 it is used by `shuffleDeck`. Here we see some big differences,

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.

---

as 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 3.2 this is not the case for *Fisher-Yates* , since it uses 6 bits of randomness for $seed_i$ at a time. Due to the restriction on $seed_C$, $seed_E$ and *seed* in *Fisher-Yates* , the overhead added by `correctSeed` 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 in the affect on the circuit size 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. Every circuit generated with the $Frigate$ compiler will have two non-free XOR gates. This is because of the bit constants **0** and **1**, which are implemented using AND or $NADN$ gates, as described in section 2.3.

When looking into 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 then *Fisher-Yates* . On the more important comparison the *Conditional-swap* circuit has 18% less non-free XOR gates. The amount of non-free XOR gates are an easy way to speed up the protocol, as these takes more time to process. Chosing between circuit with the same functionality, the one with the fewest non-free XOR gates will encrease the performance. Therefore, is the *Conditional-swap* algorithm the one that will be used for benchmarking of the poker game.

As a note on the comparison it shouls be mentioned, that some of the variables used in `correctedSeed` are bigger, in terms of bit size then needed. As mentioned earlier only 7 bit is needed to hold the signed values for $seed_{C_i}$, $seed_{E_i}$ and $seed_i$, but a 9 bit value was used. This that the number found in 3.3 could be redused, but since this is only a fraction of the 1873 non-free gates it will not change that *Conditional-swap* has less non-free gates then *Fisher-Yates* . The fraction between the bits used is $\frac{7}{9}$, which implies, that *Fisher-Yates* has at least 1457 non-free gates in the `correctedSeed` part of the circuit. This

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.

---

will have an enfluence in the difference between the circuits when compared in table 3.4, but will not change the fact that *Conditional-swap* has a smaler circuit. If we calculate the difference based on the assumption we get that *Conditional-swap* has at least 16% less non-free *XOR* gates.

At last I will give a short comment on another possible shuffle algorithm, that could have been used. This algorithm is another form of sorting network then *Conditional-swap* . In [6] they uses this algorithm, which is known as a *Bitonic* merge sort algorithm. Such an algorithm is constructed of subgadgets, which is known as $half-cleansers$. These $half-cleansers$ are a gadget that, given an input with one peak p, $i_1 \leq \cdots \leq p \geq \cdots \geq i_n$ guarantees that the output is half sorted, such that the highest halfs of the values are in one of the two halfs. Placing these gadget correctly on the wires in the network a sorting algorithm can be generated. If a sorting network can be generated, then can a shuffle network be generated. The only difference is the condition of the swap gadget. In a sorting network this is controled by the comparison of the value on the input wires, where it in a shuffle network is based on a random input. The *bitonic* sorting network is known to generate networks of of size $O(n \cdot log(n))$, which is better then what the *Conditional-swap* algorithm can aquire. This generates a network of $O(n^2)$ in size, but as argued earlier both 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 the amortized size benefits, we see that for some card games, it can be better to use other algorithms, then the ones studied in this project. It may be the case it will outperform *Conditional-swap* . We know that a *Bitonic* algorithm would produce a smaller circuit then *Conditional-swap* , but since it *Conditional-swap* is easy optimized and therefor produce a relative small circuit. It is not clear wheather a *bitonic* algorithm would produce a smaller circuit. Overall this observation implies that there will be a cross over at some point, where it is better to shuffle a complete deck then only parts of it. It can even be the case that it sometimes will be better to shuffle a complete deck, eventhoug not all cards are needed.

In this chapter, the poker game that is used during this project has been introduced. The shuffel algorithms and ther implementations has been introduced and discussed. We saw which effect it had to only shuffle the fraction needed of the card. A comparrison of the algorithms and a choise to use *Conditional-swap* during benchmarking was done. In the next chapter the implementation of the poker game using the *DUPLO* protocol will be introduced. This will be followed by a section on benchmarking of how the protocol performs. The results will then be discussed in terms of the applicability of real world problems.

Chapter 4

The Game

The main idea of this chapter is to introduce the different stages of the poker game. The different problems encountered during the implementation ond benchmarking will be introduced and the solution to overcome these.

In the first section 4.1, on the implementation of the poker game, a describtion of the implementation the process using *DUPLO* is done. First a discuss is done on which setting is to be used during the implementation. Then an introduction to the decissions made on how to interaction with the framework and which communication should be between the petween the parties participating is done.

In section 4.2 on benchmarking, a I introduce the testing that was done on the implementation. In this section an explanation of what and how the benchmarking was done, will be given. An introduction to the different parameters tested will and their results will be done.

At last in section 4.3, a discussion of the results will be made to try to compare them with the expected values for a existing system. The discussion will be on several different parameter, to compare the performance of *DUPLO* against existing online poker games.

In the next section I will introduce the implementation of the poker game. I will discuss, in which setting the implementation is done. I will argue for the decissions made during the implementation and why these was chosen.

4.1 Implementation

Before introducing the implemetation it is important to decide, which setting of the poker game should be studdied. Two different settings are proposed and discussed. The two settings can be seen in figure 4.1. The one to the left, is a setting where the *Constructor* and *Evaluatr* each act as players of the poker game themselves. They will play against each other, they each decide on the input to the shuffle algortihm and which and how many cards should be changed. The setting on the right is a setting where the *Constructor* and *Evaluator* act as servers where players then connects. These players will act as clients connecting to both the *Constructor* and *Evaluator*. Here the clients

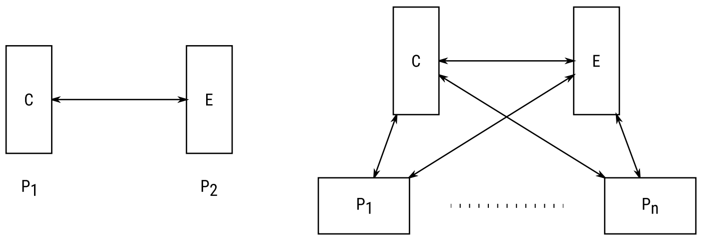


Figure 4.1: The two different settings discussed to implement. The one on the left where the *Constructor* and the *Evaluator* also are the players and the one on the right wher they act as servers where players connect and then localy construct output based on output from the servers.

---

will decide which to change as contrast to the other. The *Constructor* and *Evaluator* will still run the protocol as described. They still chose the input to the shuffle algorithm, but use inputs from the clients when opening the cards. The setting to the left is the one that will be used in this implementation mainly because of the simplicity and timelimit of the project. The other setting have some interesting features, which will be mentioned here. This setting resembles what is used at real online poker providers today. Here a player connects to a server, which handles the shuffleing and dealing of cards. Instead of connecting to one server, which is not completly trusted, the player connects to two servers which runs the *DUPLO* protocol, to ensure the propertied discussed in section 2.1 on *DUPLO* . This setting distribut the trust issiue with the setting used today. It could be that one of the serves in the *DUPLO* setting is a State authorized server in which the palyer would put its trust. Then game providers is required to use this protocol against the server to be allowed to provide games in that contry. It will allow for distributed trust issiue instead of a single piont. In this setting there are no limits on how many playes could connect to the servers. In this way the *DUPLO* protocol is not a restriction on the amount of players, as is the case for the setting where the *Constructor* and *Evaluator* acts as players themself. This server setting then requiers a way of handling authentisity between the players, the *Constructor* and *Evaluator*. This is needed to ensure that the *Constructor* and *Evaluator* does not send different outputs to the players. Because of the timelimets of the project this setting was not considered.

Now that the decision of the setting to study is made we can start to look into the actual implementation of the poker game. Taking a look at the way the *DUPLO* protocol is constructed, we see that it allows for preprocessing of the circuits before taking input for the functionality to compute. The preprocessing ensures that the right informtion is at the right party before evaluation, this is both keys and commitments of the internal protocols. The *DUPLO* protocol can be catogorized in 3 main phases. The first phase will be denoted the *Preprocess* phase which consist of the framework calls, from section 2.1, `Connect`, `Setup`, `PreprocessComponentType`, `PrepareComponets` and `Build`. This prepares the functionality of the circuit to be evaluated. The next phase will be denoted the *Evaluation* phase which calls the `Evaluate` framework function. Here the input to the circuit is given and evaluated based on this. The last phase will be denoted the *Online* phase, which consists of the `DecodeKeys` calls. It is called the *Online* phase since it is here the online comunication outside of the *DUPLO* framework takes place. The construction of the *DUPLO* protocol in this way allows for an intens *Preprocess* phase, where alot of data is processed and communications between the parties is done. This gives a posibility for a faster *Evaluation* phase and results in a small overhead when running the *Online* pahse. Inthe setting studied here this implies that the most of the time will be done in the *Preprocessing* phase, but when this phase is done the next phases will be fast.

I will now describe the implementation of the poker game. First of all we need to both implementa a *Constructor* and an *Evaluator*, since thise have different roles in the protocol. These then run the protocol in parallel with the same framework calls, as required by the framework. The parties holds different information during the phases. First the decired functionality is read in to both parties when they are created. The decired functionality in this case is the *Conditional-swap* shuffle algorithm implemented in section 3.4. The creation is done with the compiled circuit version of the algorithm. The first interaction between the two parties is then done with the `Connect` call. This setup the communication channel betwwen the parties. On the *Constructor* site this generates the server functionality which the *Evaluator* then connect to with this call. After this a call to `Setup` is done to initilize the commitment scheme used by the protocol. The call to `PreprocessComponetType` is then made on each of the subcircuits, specified in the circuit file. Based on these subfunctions, of the circuit, a number of garbled copies are generate. Then call to `PrepareComponets` is done, this takes the number of input wires in the circuit and generates the key authenticators to securly transfter the input keys and attach all output authenticators. At last, the circuit are constructed from the garbled componets by a call to `Build`. This call ensures that the garbled subcomponents are soldered together such that the functionality of the input circuit is guaranteed. No special input is given to any of the parties during this calls except of the circuit representing 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, here 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 algorithm. Since the shuffle algorithms always shuffles the same value, which are harcoded into the circuit, it only takes one input and not two as discussed in section 3.4 on the inplementaion of the circuits. Therefore it is sufficient for the parties to only 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 then used as the input to `Evaluate`, which then evaluate the garbled circuit based on this input. The function returns the values of the evaluation to a predetermined empty array.

The last phase is the *Online* phase, it is in this phase the calls to `DecodeKeys` is done. This phase is run for each round of poker game played. The amount of possible games is specified in the number of simuntainiusly shuffled decks. This is a value hardcoded in the circuit before it is compiled. Some consideration was done in the implementation of the *DUPLO* framework abouth the possibility to specify how many copies to generate of a functionality. This was not done. Therefor it is the compiled circuit holding this properti and not the parties participating in the protocol. In this *Online* phase the *Constructor* and *Evaluator* makes three calls to `DecodeKeys`. Before the first call they must agree on which outputwires should be opened to which party. This is done by generating an array containing the wire indexes to be opened to the *Constructor* and to the *Evaluator*. These are then opened by the first call to `DecodeKeys`. The return values which are in the range from 0 to 51 are then translated into card representations and displayed to the players. The translation from values $n \in \{1, \ldots, 52\}$ to a vector $card = (v, s)$, where v is the value of the card and s is the shade. This is 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 the first call to `DecodeKeys` is done the first hand is dealt. Here the *Constructor* is always dealt the first five cards from the deck, whith indexs 0 to 4. The *Evaluator* is dealt the five cards with indexes from 10 to 14. This is a more optimal way to deal the cards then by opening one at a time, as this requires the less calls to `DecodeKeys`. This way of dealing the cards does not change the propability of the cards, as the distribution of the different permutations is uniform. 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. The probability skew is probably such that a sequences of cards are more likely to repeat. This is not the case for our shuffle algorithm, since it has a uniform propability distribution. When 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 which cards to change. This is done by inputting either, 0 if no cards needs to be changed, 1 for the first card to be changed, 2 for the second and so on. If multiple cards is to be changes, then 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 must know which indexes should be opened by the second call to `DecodeKeys`. Therefore the information of which cards are to be changed is sent to the otehr party. First the amount of cards to be changed is sent. Then the indexs of the cards that should be changed are sent. This ensures that each party knows which wires should be opened in the `DecodeKeys` call. The old array holding the index of output wires to the first hand is updated to hold the new index wires. The index wires can be calculated since the cards with indexes from 5 to 9 is reserved for the *Constructor*'s second hand and the cards with indexes from 15 to 19 is for the *Evaluator*. Then the second and final hand is opened by a call to `DecodeKeys`. The cards are then translated and displayed to the parties as above. By opening the second hand as decribed, adds an overhead to the implementation, since one card may be openend in both the first and second call to `DecodeKeys`. By doing this it allows us to do a need little trick, such that no exchange of indexes are required between this and the last call to `DecodeKeys`. Thereby adding less complexity to the comunicatin rounds of the parties.

Now that the last hand have been dealt, we want to know who has the best hand. This is done by the last calle to `DecodeKeys`. Here the opening is done by swapping the inputs holding the index wires to `DecodeKeys`. This way the *Constructor* learns the output for the *Evaluator* and the other way around. Thise cards are then displayed at each party. 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. The data written is the timings for each framework call and the amount of data send, per *deck* shuffled. This data is used in the section 4.2 to discuss the effeciency of the *DUPLO* protocol in the setting of a poker game.

It can be argued that some information will leak when opening the second hand as described way, but the leak of information when communication indexes for `DecodeKeys` to the oponent is not a problem, as the opponent does not know which value is hold by the index. The propability for holding one specific card is uniform. It can also be argued that in the real world case it also known how many and which cards are changed. Therefor this communication of indexes does not leak any information.

Another note on the `DecodeKeys` framework function. When implementing the poker game I experienced problems with a call to `DecodeKeys`, when trying to do unique openings of cards to one of the parties. It was discovered that an update to the *DUPLO* protocol had only been done on one of the sides such that unique openings could not be done. This reported to the recearsh group and fixed.

In this section I have introducet the setting of the poker game and the implementation. A description of the interactin with the framework has been done and how different challenges has been solved. In the next section I will introduce the benchmarking done on this implementation and explain the data from these.

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 the *DUPLO* framework can reach runningtimes that are acceptable.

Before going into details on the benchmarking, the setup used during the testing will be introduced. To do the benchmarking a machine running both parties in the computation, was used. The hardware setup of this machine 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. In each setting I will introduce the setup of this script.

The implemntation was done such that different flags could be triggered to allow for an easy change of the setup. The flags implemented was, `-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 `-e` flag was set to $8, 8, 8$ to make use of multithreatting in all phases. This allows a faster roundtrip of the benchmarking. This setting does not simulate clients of online poker well, as these will not run a multithreatted poker game. This setup does then simulate the setting of servers communicating well, as these will try to use as many threats as possibe. Another flag that was used during all the test was the `-i` flag, which was set to 0, such that the testing could run without the need of any interaction.

When dicussing the data it will be done using the different phases described in section 3.1, *Preprocess*, *Evaluate* and *Online*. This is done to reduce the amount of information in the figures, such that only the most necesary information is precent.

The tests done was to mesure the overhed of using the *DUPLO* protocol and see how good results we could get from it. The first test done was to see how the approch of *DUPLO* to cut and chose, 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 in the different phases and the data sent. 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 created 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 3.3 was used because it has the one with the least amount of non-free *XOR* gates, as can be seen in 3.4 and therefor should be faster then *Fisher-Yates* .

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 a more fair result 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` is high, 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, to simulate

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.

a real latency. This was found to be a fair latency based on pinging different ip addresses in europa as seen in table 4.1.

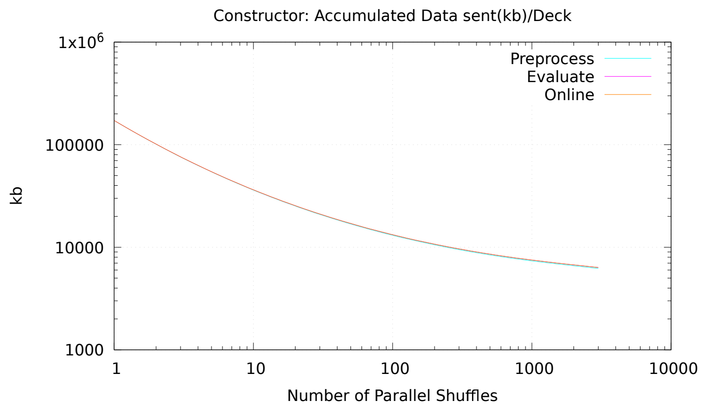
The bandwith used in this test was set to 1024Mb/s, which is in the high end of what is normaly found in denmark. A statistic[1] shows an average on the fastest network type to have a bandwith speed of 70Mb/s, this statistic is based on houshold download speeds and not server bandwith connections. Which can be assumed to be heigher when requiring higher throughput.

What we will expect to see int the results is that the more simuntanious shuffles we do the lower runtimes and lower data communication. We remember that in the *Preprocess* phase of the *DUPLO* protocol alot of communication is done to exchange keys and commitmens. Therefore we see an overhead here compared to other protocols. The expectation is that by doing many simuntanious shuffles this overhead will be spread across each deck and therefore will the cost per deck shuffled be smaller. This is the expectation in terms of both runtime and data communication. The benchmarking for the data sent can be seen in figure 4.2 and the runtime in figure 4.3.

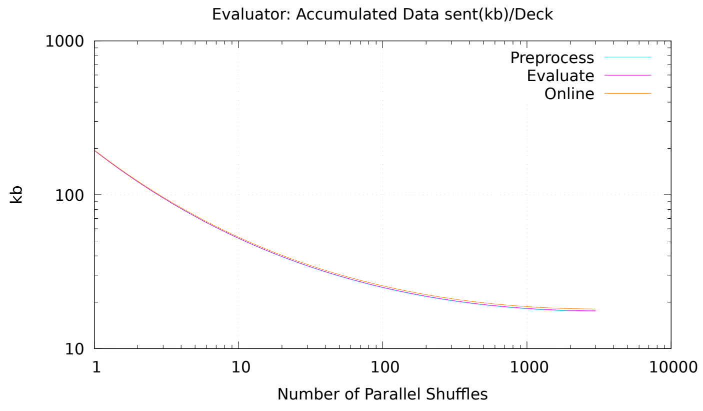
The first we will look at is the amount of data send in *kb* per shuffled deck. We see the accumulated dats sent per deck shuffled. The data is represented on a duble logaritmic scale, to be easier to see the development. It is easy to see that as more simuntainiously shuffles are done the less data is sent per deck shuffeled. This implies that the most data sent is the overhead, of setting up the protocol. As we see it is hard to distinguish the different lines on the plot. This implies as expected that the *Preprocess* phase is the one where the most data is communicated between the parties. Relative to this nearly no data is sent in the *Evaluate* and *Online* phase. Remembering that it is only the input to the functionality that is sent in the *Evaluate* pahse, which is 830 bits for each shuffle. In the *Online* pahse we call `DecodeKeys` three times and therefore is it only these keys that are sent. For the *Preprocess* phase the information of the garbling, soldering and authentication is to be sent, which requires more communication.

In figure 4.2b we see that the line is flattening out indicating that there are

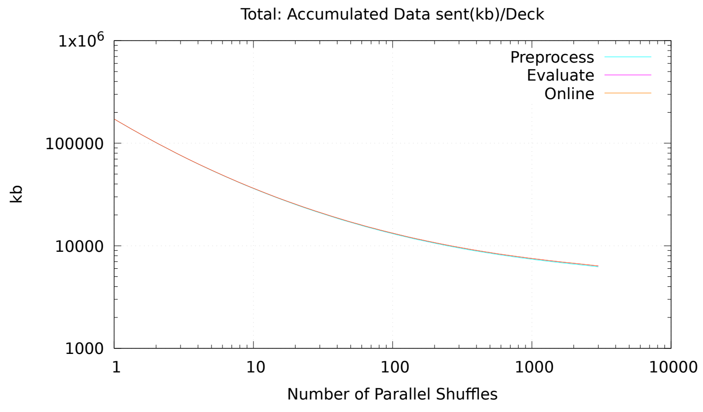
[1]A statistic of bandwith speed in danmark is found from 2015 here `https://www.statista.com/statistics/593964/average-internet-download-speed-by-connection-type-in-denmark/` It is fair to assume that the bandwith rate has gone up since 2015.



(a)



(b)



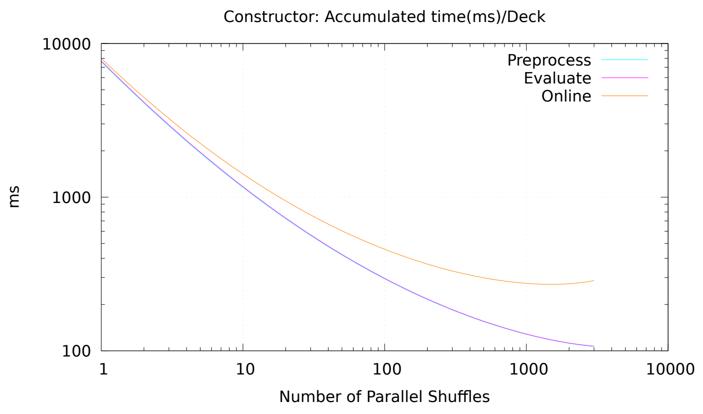
(c)

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.

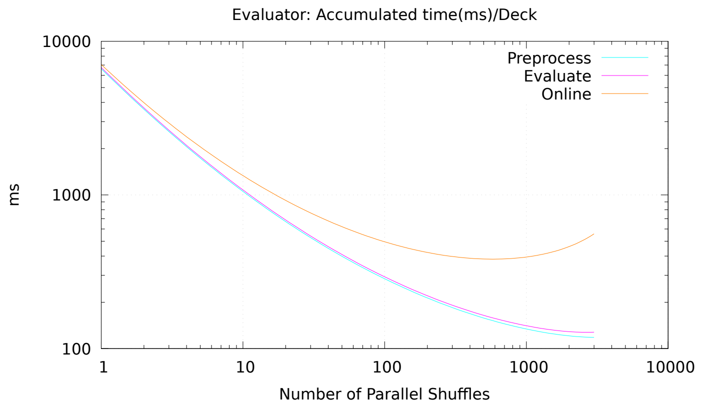
not much more to gain in shuffeling more deck on the *Evaluator* site. If we then look at figure 4.2a we see a different tendency where the plot is still decreasing, indicating that some gain can still be done on the *Constructor* site of the protocol. This may indicate that doing more then 3000 simuntainus shuffles, can bring the amount of data sent per shuffle futher down when looking at the total amount of data sent in the protocol, which can be seen in 4.2c. When consulting the graph of the total amount of data sent by the parties we see that it is still decreasing, indicating that mor is still to be gained in terms of the data transmission. Looking at the scale on the *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 using *DUPLO* down.

In this experiment we see excatly what we expected as described above. The amount of *kb* sent would decrease as more shuffles were done because of the overhead of using *DUPLO* . Even going beond the 3000 shuffles seems to give a decrese in terms of data sent. The gain of going beyond the 3000 mark is not nearly as significant as the gain of doing the first 100.

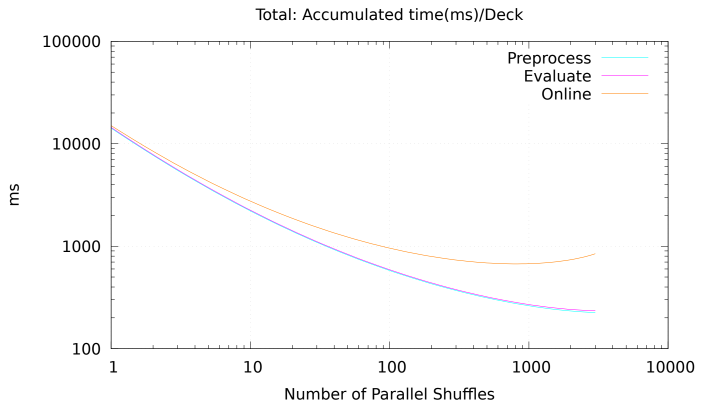
In the next section we will look a the running time used of the framework calls per shuffled deck. This can be seen in figure 4.3. The plots are the accumulated running times on a duble logarithmic scale. In figure 4.3a we see the time used in the different phases for the *Constructor*, in 4.3b the *Evaluator* and in 4.3c sumation of these two. On the *Constructor* side we see that the *Preprocess* phase accumulates the most of the time. We also see that the time use in this phase is decreasing an approching 100 ms per deck shuffled, when shuffeling 3000 decks simuntainiously. At the same time we see that the *Evaluate* phase does not add any significant time. It is a different senario when looking at the *Online* phase. Here we see that the space between the *Evaluate* and *Online* graphs encreases significantly, when approching 3000 shuffled deck. This tells us that the *Online* phase uses more time per shuffled dech, when the amount of shuffles encreases. This it not as expected and we will look into that in the next section. For now we can conclude that from the perspective of the *Constructor*, in terms of accumulated running time per shuffled deck, there exists an optimal number of shuffles around 1500. This optimum will propably variate if tested on oter hardware, as the hardware used here are at it lipits when doing 3000 shuffles. If we the look at the *Evaluator* in figure 4.3b we see the same tendency as for the *Constructor*. The *Preprocess* is the one consuming the most of the time and here approching 150ms per shuffled deack. For the *Evaluate* phase we see a small encreas in time used per shuffle when pasing the 1000 mark. This may be because of the encreas in the size of the input to the algorithm. The total time used on these two phases seems to be slightly decreasing. Indicating these can still gain some from doing more shuffles. When we turn oir attention to the *Online* phase, we once again see an increase in the runningtime. For the *Evaluator* the encrease is more significant than for the *Constructor* and the optimum is around 500 shuffles. This is significantly earlier then for the *Constructor*. When we then combine the running times in figure 4.3c, we see the same tendencies as for the *Constructor* and *Evlaluator*,



(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.

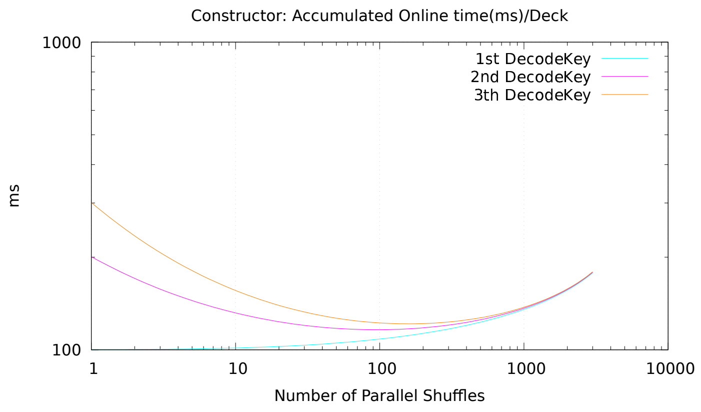
but with an optimum around 1500.

This was not the result we had expected. As we descused erlier we expected the running times to decrese through the complete graph. In the next section we will look into and try to come up with an explanation of why we this results.

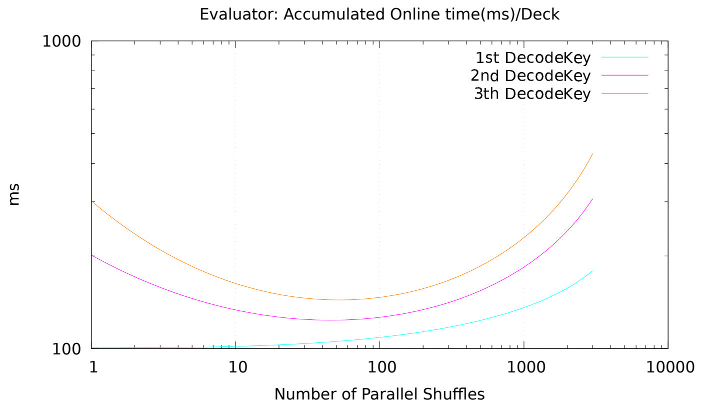
We will now take a closer look at the *Online* phase, to see there are som e ovious reasons why we get the results of an encreasing *Online* phase. First off all we start by remember that the *Online* phase has three calls to the `DecodeKeys` framework function. In figure 4.4 we see the accumulated times used on these `DecodeKeys` calls. The tim eused by the *Constructor* can be seen in figure 4.4a. Here we see a encreas in the time used on the first `DecodeKeys` call, while the two other calls decreases as expected. This can imply that somthing happens in the first `DecodeKeys` call, that we did not expect. If we then turn our attention at the *Evaluator*, in figure 4.4b, we see a graph that looks different from the *Constructor*'s. The main reason is because the most of the work done in the `DecodeKeys` call is done by the *Evaluator*. Here we see an increase in the time used by the first call, while the second and third call firs decreases and then encrease again when going towards the 3000 mark. When consulting figure 4.4c for the combined plot 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 other calls 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 in appendix A.4.3, table A.1, we see that approximatly 2.5kb is sent, during the *Online* phase. Sending this amount of data on a network, with a bandwith of 1Gb/s, takes 2.5ms. 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, since we see a fine caching for the *Constructor* in the second and third call to `DecodeKeys`. The fact that graph looks as it does for the *Constructo* may indicate that some form of caching is taking place. We cannot say the same about the *Evaluator*, it seems like some chaching could take plase as the second and third `DecodeKeys` call starts by decrease. What then happens when passing the 100 shuffle mark is hard to say. The best guess is that the cach may be full and therefore an encrease in time is happening, because it has to get the data for a slower memory.

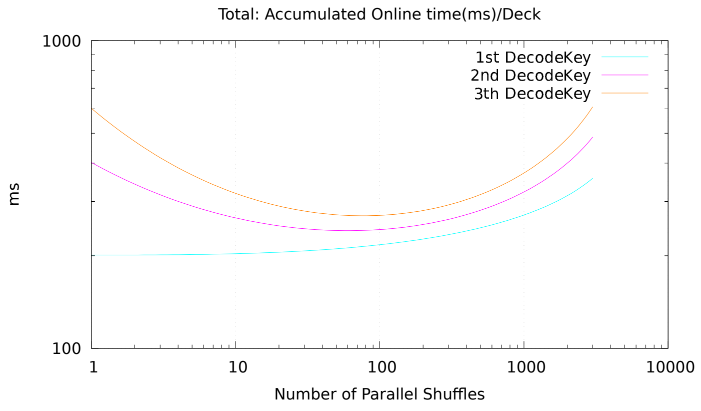
One thing that could support this hypothesis is that it seems like the timings done by each of the `DecodeKeys` calls seemt to be more fluctating when shuffling many decks simuntainious, in a small test I did. This supports the idea that it might be I/O wait that causes the decrease in performance. There simply is a higher risk of some call to wait when shuffling 3000 decks compared to shuffling 100. When we then take a look at the time used by each call we see that these taks around 100ms, if a call the has to wait for just 1ms, we see an encrease in



(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.

time by 1%.

To test this hypothesis of a full chach a test run with the `-d` flag, for the ram only mode, was performed. This did not show any change in the runningtime other then the uncernety of the test results. Another approch to cover the reason, could be to take a deeper look at the framework and test the internal structure of the `DecodeKeys` call, to understand what causes this development in the time consumption. This has not been done because of the time limets to this project.

In the next sections I will look into the other things that might affect the protocol. Here it will be the latency on the network and the bandwith. First off we will start by looking at how the latency affects the *DUPLO* protocol, this can be seen in figure 4.5. The expectations is to see a decrease in performance when the latensy go up, in other words, we expect the running time to go up when the time used on the network goes up. The testing was done with the 1000 simuntainious shuffle circuit, as this was the constructed circuit closes to the optimum discovered above. The bandwith was still at the 1Gb/s mark, to complete the test faster.

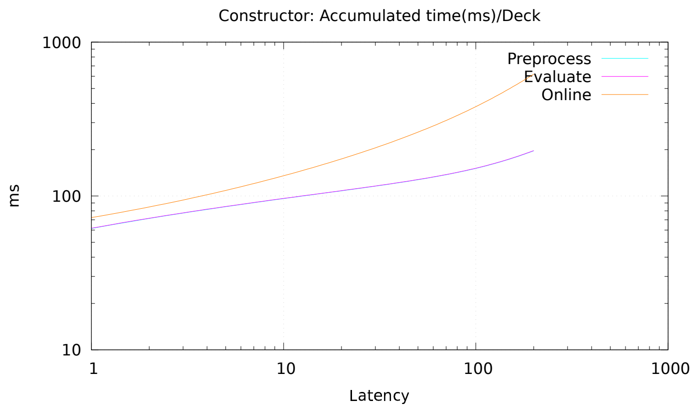
In figure 4.5c, the overall impression is that the protocol handles delay in a constant encreasing way. When looking at the *Constructor* in figure 4.5a we see that the *Online* phase is the one effected the most by the network latency, the same is the case for the *Evaluator* but not as severly. As argued before the *Online* phase uses a significant amount of time on the network. Therefor an encrease in the delay on the network would affect this part of the protocol the most. When comparing these graphs with table 4.1, it seems the protocol is performing fine for the latencies found there.

I will now turn the focus towards the benchmarking of the protocol against the bandwith. To performe this test the 1000 shuffle circuit was used onece again. This time the latency was turned down to 0 to speed up the process. The expectations of the test is to see a decrease in running time when the bandwith goes up. It is worth mentioning that no tests are done with bandwiths slower then 50Mb/s, because the test machine chrashed when trying to performe test below this treshold. If consolidating the statistic[2] a bandwith of 50ms is within the once used.

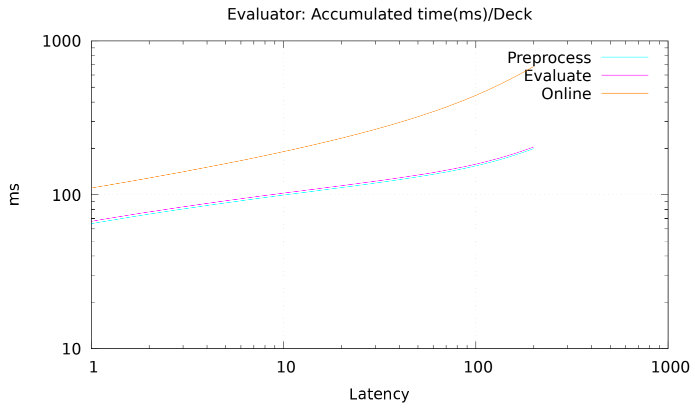
The results of the bandwith test can be found in figure 4.6. Here we see some fine constant decreasing graphs, both when consulting the graph for the *Constructor*, *Evaluator* and the total. We see that the *Online* phase levs the *Evaluate* phase a bit to come closer again. This is most propably do to the uncertanty of the testing environmet. Overall this folows the expectations we had.

In this section I have tried to cover the tests done an argue for how they wer performed. I have discussed the results we have seen and try to argue for

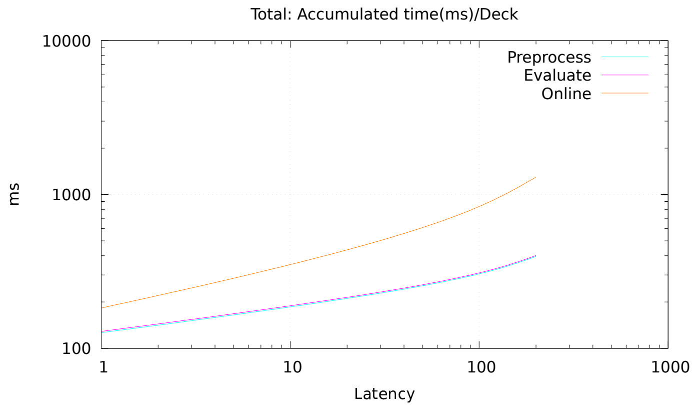
[2]See link in footnote on page 35.



(a)

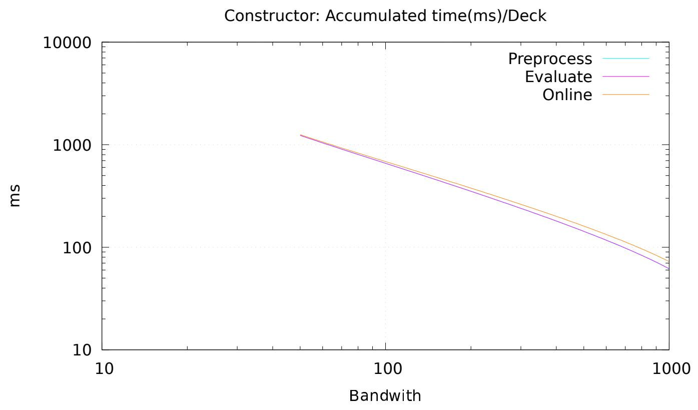


(b)

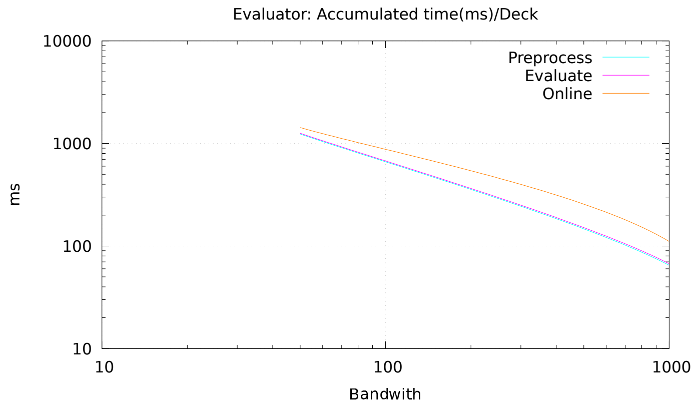


(c)

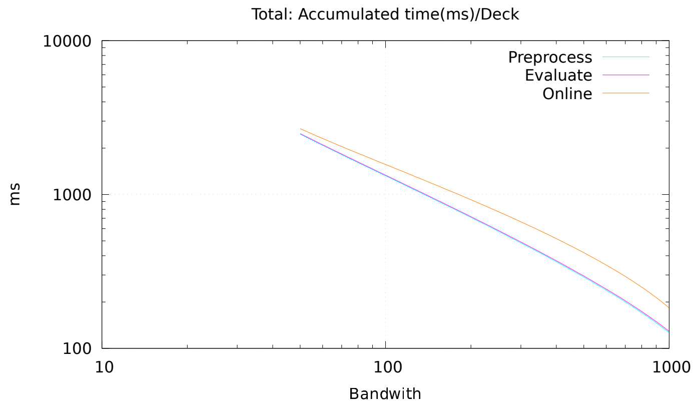
Figure 4.5: Delay: Comparison of *Constructor* and *Evaluator* in *ms*'s used when 1000 shuffles are done with different latency on the network. (a) *Constructor*: Accumulated time per deck shuffled on a duble logarithmic scale. (b) *Evaluator*: Accumulated time per deck shuffled on a duble logaritmic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

why they are a they are. If the results was not as expected I have tried to cover these areas with new results within the timelimmit of the project. In the next section I will try to hold the results from this section up against whant can be expected in of runningtimes of real online poker.

4.3 Discussion

TODO: ahead of time generation

In the tabel we se an average latency of 47ms. Consulting the graph on the total running time we se that, such a latency would result in a running time around 500ms per deck shuffled.

Chapter 5

Conclution

TODO: future work

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 3.5.

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 3.2. 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`

	Number of Parallel Shuffles				
Phase	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*

	Number of Parallel Shuffles				
Phase	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*

	Number of Parallel Shuffles				
Phase	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.

Phase	Number of Parallel Shuffles 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*

Phase	Number of Parallel Shuffles 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*

Phase	Number of Parallel Shuffles 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.

Bibliography

[1] Ronald Cramer, Ivan Bjerre Damgrd, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing.* Cambridge University Press, New York, NY, USA, 1st edition, 2015.

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

[3] 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.

[4] 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`.

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

[6] 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.