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

This project is a study in, how secure two party protocol($2PC$) perform in the application of a poker game. As online poker is played today, the player can not now if the cards are dealt fairly. There exists protocols that can handle this. These are called secure multiparty computations(MPC). In this project a study of these will be done to see how they perform.

In this project the *DUPLO* protocol is used to implement a two party five card draw poker game. In the setting studied the two parties, of the protocol, act as players in the poker game. In this setting the *DUPLO* protocol guarantees both *privacy* and *correctness*. These properties are reached in *DUPLO* by using *cut & chose*, *commitment schemes* and *garbled circuits*. Because of the construction *DUPLO* adds an overhead to the evaluation of the circuit. This overhead is reduced as much as possible by the construction of *DUPLO* .

To handle the card shuffling of the game, two different shuffle algorithms are studied. These are compared to find the one performing best in the protocol. Both algorithms are optimized, to the two party poker setting, such an even better performance can be reached.

The poker implementation is benchmarked to measure the performance of *DUPLO* . Because of the *DUPLO* construction, a test is done to see when the pest performance can be expected. This is done by doing different amounts of simultaneous shuffles in the circuit. The expectation is when more simultaneous shuffles are done, the performance of the protocol will be better, per shuffle. The results show a distribution of the overhead that is decreasing, when doing more simultaneous shuffles, but the cost of decoding the output is increasing. An optimum is then found from the test. The increase in the cost of decoding, is assumed to be the result of a full cache. This optimum was then tested to see the performance when different latency and bandwidth was used. The expectation is to see a linear development. The results is as expected. When a network with low latency is used the performance goes up. When a higher bandwidth is used the performance is better.

When the results are combined a good performance is achieved by the implementation. Translating this into the setting of online poker as it is known today, no distinct conclusion can be made. There are to many unknown variables in the benchmarking. The tendency of the results indicates, that a good solution could be created.

Resumé

Dette projekt er et studie af hvordan to-parts beregnings protokoller yder når de anvendes i et online poker spil. Online poker spillere kan på nuvrende tidspunkt ikke vide sig sikre på at kortene bilver delt retfærdigt. der eksistere protokoller der kan sikre dette. Disse protokoller kaldes sikre flere-parts beregnings protokoller.

Dette projekt er et studie for at se, hvordan disse protokoller yder. I dette projekt bruges *DUPLO* protokollen til at implementere et "five card draw" poker spil, med to spillere. I denne opsætninge agere de to parter, af protokollen oså som spillere i pokerspillet. I denne opsætning kan *DUPLO* protokollen garantere både privatliv og korrekthed. Disse egenskaber i *DUPLO* er opnået ved brug af følgende teknologier *del og vælg*, *kommitment systemer* og *krypterede kredsløb*. På grund af måden *DUPLO* er kostrueret tilføjs en margin til evalueringen af kredsløbet. Denne margin reduceres så meget som muligt i *DUPLO*.

Til at håndtere blandingen af kort, studeres to forskellige blanding algoritmer. Disse er sammenlignet for at finde den mest effektive, til bruge i pokerspillet. Begge algoritmer er optimenret så en bedre kørselstid kan opnåes i to-parts beregningen.

Poker implementering er tested for at måle kørselstiden ad DUPLO. På grund af DUPLO's konstruktion er en test lavet for at se hvornår den præstere optimalt. Dett gøres ved at bruge kredsløb hvor flere kortæst blandes samtidig. Det forventes at når flere blanding fårestages samtidig bliver ydeevnen af DUPLO bedre og bedre jo flere samtidige blandinger der laves. Resultaterne viser at jo flere kortspil der blades jo laver bliver den tilføjet marginen per kortspil blandet, men samtidig stiger omkostningerne ved at bilve tildelt kort. På bagrund af dette er et optimalt antal blandinger fundet. Stigningen i omkostningerne til uddeling af kort er forventet at komme fra et fyld hukommelses lager. Det fundne optimum bliver herefter tested i forskellige omstændigheder for at se hvordan DUPLO yder. Her bliver både påvirkningen af forsinkelsen og båndbredten på netwærket testet. Forventningen er at se en lineær uvikling. Det er også disse resultater vi får. Når forsinkelsen på netværket går ned, går ydeevnen op. Når båndbreden går op går ydeeven også op.

Da disse resultater foreligger bliver de kombineret til at, konkludere at implementationen præstere godt. Oversættes resultaterne til hvad der kan forventes af online poker udbydere, er det svært at sammenligne, da der er for mange ukendte variabler i testningen, men tendensen ser ud til at god løsning kan kostrueres med DUPLO.

Acknowledgments

During the work of this project, 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 understand 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 months 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 least I need to thank my wife for holding up with me, when I went on one of my coding sprees, and for her details for commas.

Thank you everyone.

Christian Bobach,
Aarhus, June 13, 2017.

Contents

Chapter 1

Introduction

In this chapter I will introduce the thoughts about this project. I will give an introduction to the ideas behind the project and why this is interesting. In the following an overall description of the project will be given.

First of all, we can all put ourself in the place of players of online poker, where it is not possible to know if the cards are dealt fairly or the provider collaborate with one of the opponents to win. The players puts their trust in the provider to guarantee, that the cards are dealt fairly. This problem can be overcomed by using a secure protocol, that can give this guarantee. This can be done by secure multiparty computation(MPC) protocols.

Secure multiparty computations are a construction, that allow multi parties to collaborate on evaluating a function. The protocol is designed such that the evaluation is guaranteed to have the right output, with negligible probability. The idea behind using a secure multiparty computation protocol is that, if we allow the players of poker to collaboratively shuffle the cards used in the game, then they are guaranteed that the cards are dealt fairly.

In this project a secure two party computation($2PC$) protocol will be used. This will only allow for two parties collaborating in shuffling the cards. The protocol chosen is the $DUPLO$ protocol. This was chosen because of the functionality of allowing one of the parties to learn some part of the output of the function, while the other learn the other part. This gives the possibility to deal some cards to one party and other cards to the other party.

The intent of this project is to see how, an implementation of poker using the $DUPLO$ protocol perform against what is known today. An implementation of a poker game using the $DUPLO$ protocol is therefore made. During the implementation phase, different shuffle algorithms are studied to find the one performing best. The shuffle algorithms are optimized to the poker setting used. The poker game that will be implemented is the five card draw version. This is done, because it is one of the most used when played between two players. This allows for some optimizations of the shuffle algorithms, as only a certain amount of cards are used during a game.

As only two parties are participating in the poker game, one player may expect the other to try to cheat. This could be by tying to affect the shuffle to deal better cards to guarantee a win. As $DUPLO$ is proven to be secure in the

malicious setting, where the opponent may deviate for the protocol, we are on the safe site.

The poker implementation is then used to benchmark the performance of the *DUPLO* protocol. This is done to get some numbers to hold up against, what can be expected by online poker games. Based on this benchmarking, a conclusion will be done.

In the following section, an overview of the structure of this thesis will be given. Before doing this, a timeline of the project will be introduced to give an understanding of some of the problems, that was covered during the project. The main reason for some of the problems is, because the *DUPLO* project was still ongoing, when I started this thesis. The main structure and functionality was in place, but was not of a production quality. Therefore, some bugs were uncovered in the *DUPLO* project. The *DUPLO* project and paper were first published late in my project, but I have been in touch with them during the whole process of the project. The main reason for this has been to get support when things didnot work as expected and be able to give the feedback on my findings.

We can now tur our attention to the outline of the thesis.

1.1 Chapter Overviews

In this section a short description of each chapter will be given. This is done, such that you as a reader know what to expect and where the different contents can be found. The chapters follow in this order, first the chapter on *DUPLO* , the Shuffle Algorithms, The Game and lastly, the Conclusion.

Chapter 2: In chapter 2 I cover the basics of *DUPLO* .
TODO: Introduce chapter duplo

Chapter 3: The main focus of this chapter is to compare different shuffle algorithms, such that the one assumed to perform fasts in the implementation, can be chosen to use. First an introduction to why these are needed, are given. Then an introduction to the version of poker that will be used in the implementation of the poker game. This is done, such that the requirements of the shuffle algorithms Can be analyzed. In this part a proposal to optimizations of the shuffles are introduced.

The first algorithm studied is the *Fisher-Yates* algorithm. This is studied in great depth to understand the ideas behind the construction.

The second algorithm studied is the *Conditional-swap* algorithm. This is also studied in great depth to understand the constructions. This algorithm is created based on ideas from different concepts. The main idea comes from shuffle network which is introduced here.

A description is given on how these algorithms are implemented and optimized to perform even better. Based on these implementations, a comparison

of the algorithms are done. This comparison makes the basis of the choice of, which algorithm will be used during the benchmarking.

Chapter 4: The main focus of this chapter is to generate the basis for the conclusion done in chapter 5. First a discussion of the setting of the poker game is done, such that the implementation can be made to fit the setting. A description of the implementation of the poker game is given. This is done by explaining how the interaction with the *DUPLO* framework is done and what is handled in between. It is also here an introduction to the different choices during the implementation process will be given.

The implementation is then used during the benchmarking in this chapter. This is a thorough test of how the *DUPLO* protocol performs in the implementation. Here these are done to cover, when the protocol performs optimal, in terms of simultaneous shuffles, network latency and bandwidth.

Based on the results of the benchmarking, a discussing is made. This compares the results with the expectation and conclude on the application of the *DUPLO* protocol.

Chapter 5: In this chapter the conclusion of the project is done. Is *DUPLO* and secure two party protocols mature enough to work in the real world. Here both the setting studied during the project and a more real world version will be discussed.

During the project multiple corners had been cut, because of the time limit. These are presented in this chapter in a section with proposed future projects.

In this chapter a introduction to the project and the cumming chapters were given. In the next chapter an introduction to the *DUPLO* framework will be given. The reasons of why this was chosen and the security will be discussed. Some of the concepts, which is used by *DUPLO* will be introduced, as is the case for the *Frigate* circuits compiler that came shipped with *DUPLO* .

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 $2PC$ and *garbled circuits* in the next two small sections.

2PC: $2PC$ is a special case of MPC. I have used [3] as a reference, but many others due exist. In $2PC$ 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 $2PC$ 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 settings. 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 $2PC$ 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 $2PC$ 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 $2PC$ 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 $2PC$ 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 $2PC$. 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 *independence of input* for the evaluation.

TODO: Garbled circuits can be evaluated by the garbler, there for use commitment scheme

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 **C** running the protocol can not distinguish between talking to **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* $k_{id'} = k_{id'}^{y_i d}$ *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 $k_{id} = k_{id}^{x_i d}$ 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, an introduce to the *DUPLO* version of the *Frigate* circuit compiler will be done. First an introduction to the requirements for the compiler, to work will be given. Then a describe of how the programming language used to generate the circuits works, and lastly a coverage of the bug found in the compiler. Here an explenation of how this was done and what affect that had on the *DUPLO* project will be given.

First of all a reson will be given of why a circuit compiler is needed. Because the *DUPLO* framework uses *garbled circuits* to represent its functionality, we need a way to give the functionality on a circuit form. This could be done by writing the circuit gate by gate. This is in no way feaseble, as when functions

[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 iterative way, in blocks of card size.

---

get the least complex, their respective circuits will have a great complexity. Small simple functions can be implemented in circuits, but not if the function should do some complex. One of the reasons for chosing $DUPLO$ was the ability to have a compiler translating a higher level programming language to its respective circuit representation. When using a compiler is is posible to go from one abstration level to another.

First we need to install the compiler. To do this some instalation of libraries are required. These libraries are not the newest version and are therefore not easily installed. Following the instructions in the installation guide will get the libaries installed. The hardware I have been using to run both $Frigate$ and $DUPLO$, can be found in appendix A.1. The packaged required by $Frigate$ is $flex$ and $bison$. The specific versions required by $Frigate$ is older then the one in my package management system. They were therefore required to be hold back, such that no updates was installed and brok the $Frigate$ compiler. To ensure a consistency on the system a `docker` image was created, such that the correct versions was allways installe. This can be found via the link it the docker image in A.4.2. When $Frigate$ was up and running it was time to look into the documentation.

The documentation for the $DUPLO$ $Frigate$ compiler was from the first version. This was not updated, when the functionality was extended to cover the $DUPLO$ framework. This is what could be expected as the functionality of the compilers input language was not changed. Because the documentation was weak at pints it resulted in some time consuming trial and error. It allways takes some time to get familiare with a new programming language. This resulted in some hard earned experience on very small examples. An example is shown in figure 2.1. In this example the XOR is done on different parts of the inputs and stored in different sections of the output. A more detaild description of the programming language will follow in the next section.

Some of the most important and different things to take into account is, the possibility to do wire access. It allows you to specify exactly which and how many wires should be represent in a certain value. This is done by the following syntax `s=s1{index:size}` in the $Frigate$ `wir` format, as shown in

figure 2.1. This allows for single bit inputs to be translated into higher level representations, like variables. The `index` indicate the wire position to start from and the `size` indicate ho many wires from `index` should be included in the representation. Here it is important to notice that it is not possible to access wires based on variable inputs. As the variable input cannot be predetermined by the compiler. Which makes perfect sense because circuits are static representations.

A second point to remember, when working with *Frigate* is that it do only allow one level functions. This puts some restrictions on the programming language, as it makes it harder to create small functions with a single specific focus. This restriction may be because of the way the functions are stitched together when generating the circuit. When we look at the functions another thing to remember, is that *Frigate* only aupports assignments of variables in the main method through function calls. This gives completely sense when looking into the *DUPLO* circuit file, as the main function do not contain any gates.

Otherwise the `wir` programming language resembles the well known *C* language. The compiler requires you to specify how many parties the functionality is used by, by the decleration of `#parties n`, where `n` is the number of parties. At the same time, it requires that the size of the input and output for each party is specified using `input i size_i` and `#output i size_o`, here `i` is the party and `size` is the size. The programming language allows for definitions of constants, types, structures and imports like *C*.

When the desired functionality has been implemented using the `wir` format, the compiler can be used to translate it into the *DUPLO* format. This is be done from inside the compiled *DUPLO* framework, by running the following command:

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

The `-db` flag ensures that the *DUPLO* format is generated. This call to the compiler generates a lot of different files, with different extensions. The file required by *DUPLO* is the `.wir.GC_duplo` file.

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 was done to the *DUPLO* project based on it. First of all, the *DUPLO* framework has a functionality, that allows for evaluation of a circuit files without setting up both parties. This opportunity give the posibility to specify the input values, for the computation of the circuit. This gave the ability to test the functionality of the circuits in a fast return cycle. This helped when learning the `wir` programming language. First I tried to implement the *Fisher-Yates* shuffle algorithm, which will be introduced in chapter 3. During the implementation, an anormaly was encountered, when trying to use the modulo operater. The modulo operator `%` is specified in the documentation and should therefore be possible to use. It could easily have been the problem that the implementation of the algorithm

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*

was wrong. Therefore at first the focus was debug the implementation. After a while it was clear that the compiler was the reason to the error. Since the compiler used, was a modified version of the original *Frigate* compiler, the hinch was that a bug could have been introduced when adding the new *DUPLO* functionality. The original *Frigate* compiler was therefore installed to test, if that implemetation had the same problem. Since the original version of *Frigate* did not support the *DUPLO* circuit format, a parser was generated to translate that format into one *DUPLO* could read. It is the case that *DUPLO* supports two different file formats. One that uses circuit representated as on big circuit and one where the representation is splited into smaler functions. I was therefore not forced to make a new implementation of the *DUPLO* features. The gate format used is known as *bristol*[2]. In this format a gate lookes like this `n_i n_o [i] o g`, where `n_i` is the number of input wires to the gate, `n_o` the number of output wires, `[i]` a list of length `n_i` specifying the input wires indexes ins the circuit, `o` specifying the ouput wire index in of the gate and lastly, `g` specifying the gate type. An exor gate could then look liker `2 1 1 2 3 XOR`. This was the original format used to rperesent gates in *DUPLO* . This was late changed, because of the bug discovered. To be able to use the file format with the full circuit representation, I wrote a parser that took the output format from the original *Frigate* compiler, and translated that into the *bristol* format. The original compiler format for a gate looked like `g o [i]`. Where the gate ty[e `g` was represented as the truth values from table 2.1, where xor would be represented like `0110 3 1 2`. After the parser was implemented a test run showed that there was a difference between the *DUPLO* and the original version of the *Frigate* compiler. Especially when the modulo reducion was used. This showed that not all gate types was implemented in the *DUPLO* version. The modulo reduction triggered one of the gates not implemented and therefore coursed an error in the circuit.

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. After this all gates in the *DUPLO* format was changed from the bristol format. The new format used a cobination of the olde *Frigate* format and the *bristol*, such that it looked like `[i] o g`. The specification of `n_i` anf `n_o` was removed and the `g` was changed from `XOR` to `0110`. This change in representation off gates only allow for gate types with two input wires. This allowed for the 16 gate types shown in table 2.1. The representation of binary constants in the circuit

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

for the **0**, is `1 1 3 0110`, and **1**, `1 1 3 1001`.

In this way by going into details and debugging the implementation of the *Fisher-Yates* algorithm, I contributed to the *DUPLO* project by reporting the findings and thereby adding value to their reaserch project by securing a more stable product.

While a fix was done to the *Frigate* compiler, such that the gate types are now all supported. The modulo operater does not work for devisors with other values then thoes that are powers of two. This behavior of modulo is specified in the documentation and is a lefower from the original *Frigate* project. Some time was used to research if a fix could be made, such that this functionality could be implement. While it seems to be rather easy to do the implementation of modulo with divisors of powers of two, it seemes not to be the case for all other devisors. The modolo operator was therefore left unfixed. In the 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 some optimizations that can be done to reduce the amount of cards that needs to be shuffled. I will also intro another type of poker then the one studied, to compare their differences.

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 reached we can ensure a uniform 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 pseudo code as algorithm 1. When talking about an in place permutation algorithm, a algorithm with only one data representation to hold the values to be shuffled is meant. Another normal approach 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 pseudo code 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 choices 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. This variant of poker was chosen because it is one of the most used poker games when played between only two players.

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

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
```

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 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 odd 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 number 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 choice 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 specifying 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 values $card_i$, 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 yields exactly the number of permutations required, as $card_1$ has exactly n possible swap possibilities. $card_2$ has $(n-1)$ possibilities and so forth. The last $card_n$ is determined 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 described 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 yield 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 introduced 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 divisible by $n!$, and therefor cannot give the desired uniform probability.

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 straight forward. Instead of running through n swaps indicated by the size of *seed* it is enough to run through m swaps. This yields that the 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 yields the first 9 cards of the shuffle 1, 52, 14, 20, 10, 37, 9, 33, 6 as output. Here it is interesting to

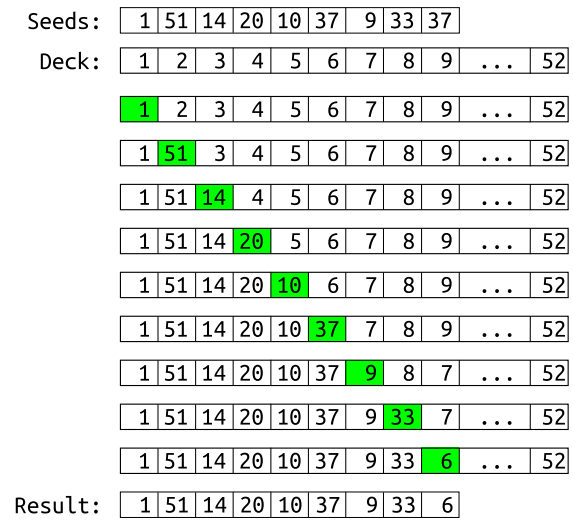


Figure 3.1: Fisher-Yates algorithm in action: In this figure a 9 out of 52 shuffle has been completed to illustrate how the algorithm works. First 1 is swapped with 1. Then 2 is swapped with 51. 3 with 14. 4 with 20 and so on until the first 9 numbers has completed a full permutation. Resulting in 1,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

---

notice, that 37 occurs twice in *seed*. Because the algorithm permutes the *deck*, with the seed value 37, 37 will not end up in the output twice. The first case 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 swapped in as this was now located at that spot in the deck. This illustrates that it is possible for a *card* to be swapped multiple times and therefor 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, because 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

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
```

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 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 simplicity and the ease of optimizing it to a m out of n shuffle algorithm. This algorithm will be denoted *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 illustrated 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 string 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 until, 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 again 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 pseudo code in algorithm 2, because i increases by one on line 11. The algorithm suffers the same problems as *Fisher-Yates* , of producing n^n permutations, which is not divisible by the desired $n!$ permutations, as explained for the *Fisher-Yates* algorithm. This resulting in a skew of the probability on the different permutations, such that the probability 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 bit string indicating if two values should be swapped. 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 bits from *seed* indicate, that 51 should be swapped all the way across the network to the second output wire. This yields 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 continues 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

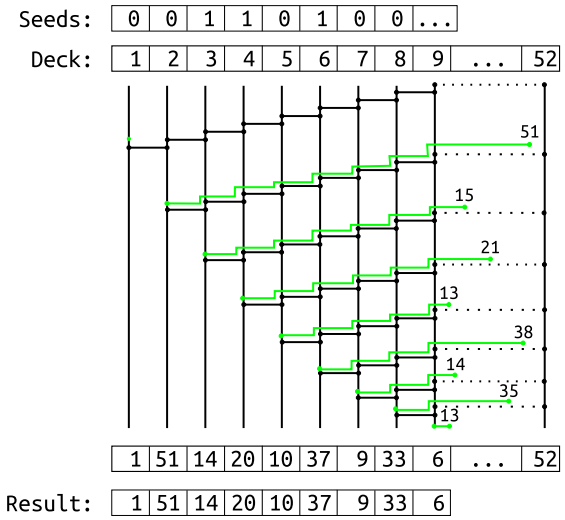


Figure 3.2: Conditional swap algorithm in action: In this figure a 9 out of 52 shuffle has been completed to illustrate how the algorithm works. Each bit in the *seed* indicates, if a gadget should swap the two incoming values. Since the size of *seed* is 423 bit long I have tried to illustrate which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

---

was chosen to use in this project, the first hurdle was to generate the circuits, that should handle the shuffling of the cards. As mentioned in section 2.3, implementing algorithms in boolean circuits become rather complex, when going beyond small toy examples. Luckily, 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 desired *DUPLO* format. An introduction to the *Frigate* compiler can be found in section 2.3. The code base for *Frigate* can be found in appendix A.3.

Now going in to the details of the two algorithms implemented. The code base for the implementations 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 stitched 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 hard coded 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

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 algorithm. The comparison is done on the amount of each gate type in the compiled circuit.

value for each *card* in the *deck*. It is important to notice, that the variable indexing the 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 structure of the *Conditional-swap* algorithm, we see that the first function used is the `xorSeed`. This function handles the XOR of the *seed* received from the two parties. This is a straightforward implementation using the build-in XOR function `^`. After this the call to `initDeck` is done, such that the result for the two functions can be fed 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, yields 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 representations 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 guarantee, that $seed_i$ was inside the interval $[0; 52 - i]$, which is discussed in section 3.2. In the implementation we use this interval, which is a slight deviation from the original one, which was $[i; n]$. This is due to the fact 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 either $seed_{C_i}$ 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 boundary $I_u = (52 - i)$ of the interval from $seed_i$, if this exceeds I_u. Doing it this way, yields a $seed_i$ in $[0; 52 - 1]$. This is due to the fact, that $seed_{C_i}$ and $seed_{E_i}$ can at both at most 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 implementation this was done by introducing an if statement 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 interval $[0; I_u]$. 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 hard coded representation of the *deck* in the circuit. It is completely fine to hard code this representation since the algorithm generates 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 handle the problem of circuits being a static representation, as discussed 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 possibility for the correct swaps. This yields $52 - i$ swap gadgets in layer i, for $i = 1, \ldots, 52$. 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 $52 - i$ distinct conditional swap gadgets, as those used in *Conditional-swap* . This gives the desired probability, as described in section 3.1.

The *Fisher-Yates* implementation can also be optimized as described 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% decrease in the number of non-free XOR gates. Which is a descent 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 algorithm. 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 shuffling algorithms studied and compared them with their optimized versions. In the next section I will compare the two optimized algorithms and discuss their differences and make a choice 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 upon their gate composition and based on that chose, which one to continue 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 therefore not yield 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 abstract 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. Even though the algorithms has different approaches to how the conditional swap gadgets are generated, they end up generating the same amount. This is because *Fisher-Yates* generates 20 composed gadgets each consisting of $52 - i$ swap gadgets, for $i = 1, \ldots, 20$. Yielding 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 greater 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. Choosing between circuit with the same functionality, the one with the fewest non-free XOR gates will increase 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 should 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 reduced, 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: Comparison 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 influence in the difference between the circuits when compared in table 3.4, but will not change the fact that *Conditional-swap* has a smaller 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 sub-gadgets, 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 half's of the values are in one of the two half's. 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 controlled 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 acquire. 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 whether 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, even though not all cards are needed.

In this chapter, the poker game that is used during this project has been introduced. The shuffle algorithms and their implementations has been introduced and discussed. We saw which effect it had to only shuffle the fraction needed of the card. A comparison of the algorithms and a choice 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 and benchmarking will be introduced and solutions to overcome these will be presented.

In the first section 4.1, on the implementation of the poker game, a description of the implementation 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 decisions made on how to interaction with the framework and which communication there should be between the parties participating.

In section 4.2 on benchmarking, I introduce the testing that is 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 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 decisions made during the implementation and why these was chosen.

4.1 Implementation

Before introducing the implementation it is important to decide, which setting of the poker game should be studied. 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 *Evaluator* each act as players of the poker game themselves. They will play against each other, they will each decide their input to the shuffle algorithm 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

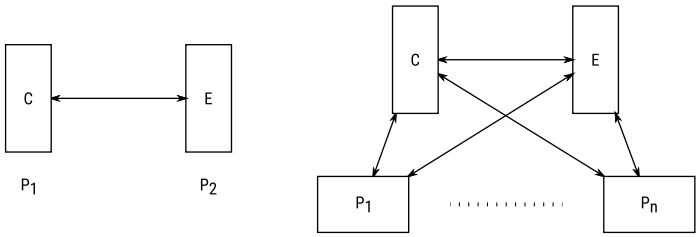


Figure 4.1: The two different settings discussed to implement. The one on the left where the *Constructor* and the *Evaluator* also are the players. The one on the right where they act as servers where players connect to and then locally construct the output of the computation based on the output recieved 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 time limit 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 shuffling and dealing of cards. Instead of connecting to one server, which is not completely 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 distribute the trust issue 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 player would put its trust. Then game providers is required to use this protocol against the server to be allowed to provide games in that country. It will allow for distributed trust issue instead of a single point. In this setting there are no limits on how many players 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* act as players themself. This server setting then requires a way of handling authenticity between the players, the *Constructor* and *Evaluator*. This is needed to ensure, that the *Constructor* and *Evaluator* do not send different outputs to the players. Because of the time limit 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 information is at the right party before evaluation, this is both keys and commitments of the internal protocols. The *DUPLO* protocol can be categorized in three main phases. The first phase will be denoted the *Preprocess* phase, which consists of the framework calls, from section 2.1, `Connect`, `Setup`, `PreprocessComponentType`, `PrepareComponents`

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 communication outside of the *DUPLO* framework takes place. The construction of the *DUPLO* protocol in this way allows for an intense *Preprocess* phase, where a lot of data is processed and communications between the parties is done. This gives a possibility for a faster *Evaluation* phase and results in a small overhead when running the *Online* phase. In the 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.

When implementing the poker game, I will not do any implementation of betting nor declare a winner. This choice was done because this does not have anything to do with the *DUPLO* protocol and would consume a lot of time from the project.

In the next section a describtion of the implementation of the poker game is done. First of all, we need to both implement a *Constructor* and an *Evaluator*, since these 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 hold different information during the phases. First the desired functionality is read into both parties when they are created. The desired 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 sets up the communication channel between the parties. On the *Constructor* site this generates the server functionality which the *Evaluator* then connect to. After this a call to `Setup` is done to initialize the commitment scheme used by the protocol. The call to `PreprocessComponetType` is then made on each of the sub-circuits, specified in the circuit file. Based on these sub-functions, of the circuit, a number of garbled copies are generate. Then call to `PrepareComponents` is done. This takes the number of input wires in the circuit and generates the key authenticators to securely transfer the input keys and attach all output authenticators. At last, the circuit is constructed from the garbled components by a call to `Build`. This call ensures, that the garbled sub-components 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 call, except of the circuit representing the functionality to be computed. Therefor this will be denoted the *Preprocess* phase, as these calls can be 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 hard coded into the circuit, it only takes one input and not two as discussed in section 3.4 on the implementation

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 are specified in the number of simultaneous shuffled decks. This is a value hard coded in the circuit before it is compiled. Some consideration was done in the implementation of the *DUPLO* framework about the possibility to specify how many copies to generate of a functionality. This was not done. It is therefore the compiled circuit holding this property 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 output wires 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 yields no collisions on (v, s) since 13 and 4 has no common multiples 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, with index's 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 less calls to `DecodeKeys`. This way of dealing the cards do not change the probability of the cards, as the distribution of the different permutations is uniform. As known for real world card games one or two cards are normally dealt at a time. This is probably done because the shuffle algorithm does 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 probability 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 separating 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 other party. First the amount of cards to be changed is sent. Then the index's 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 described, adds an overhead to the implementation, since one card may be opened 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 communication 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 call 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. These cards are then displayed at each party. Opening the opponents card this way, we ensure that the opponent does not learn the cards which were discarded.

At last, when all the decks are played, the statistics are written 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 efficiency 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 opponent is not a problem, as the opponent does not know which value is hold by the index. The probability for holding one specific card is uniform. It can also be argued, that in the real world case, it is also known how many and which cards are changed, therefore 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 was reported to the research group and fixed.

In this section I have introduced the setting of the poker game and the implementation. A description of the interaction 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 implementation using the *DUPLO* framework can reach running times 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 stable environment, such that no network 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 implementation 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 specifying the ip address of the *Constructor*, `-p_const` for specifying the port the *Constructor* is listening on and lastly the `-d` flag for ram only mode, where the computation is done without writing to disk.

The `-e` flag was set to $8, 8, 8$ to make use of multi threading in all phases. This allows a faster round trip of the benchmarking. This setting does not simulate clients of online poker well, as these will not run a multi threaded poker game. This setup does then simulate the setting of servers communicating well, as these will try to use as many threats as possible. 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 discussing 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 necessary information is present.

The tests done were to measure the overhead of using the *DUPLO* protocol and to see how good results we could get from it. The first test was to see how the approach of *DUPLO* to cut and chose, effected the poker implementation. This allows us to see, if we reach a optimum of the number of simultaneous shuffles. To be able do this test different circuits was generated and compiled. Each of these circuits was then evaluated to measure the timings in the different phases and the amount of data sent. Because the *DUPLO* framework does not allow for soldering of *DUPLO* format circuits, the circuit file used should contain all simultaneous shuffle of decks. Therefore different variants of the *Conditional-swap* algorithm are created where 1, 10, 100, 1000 and 3000 simultaneous decks was shuffled. The 3000 was chosen as a maximum since the memory limit was exceeded on the hardware, when more was tested. These variants of circuits are all compiled following the instructions in appendix A.3. The *Conditional-swap* implementations from section 3.3 was used, because it has the one with the

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 Europe.

least amount of non-free XOR gates, as can be seen in 3.4 and therefore should be faster then *Fisher-Yates* .

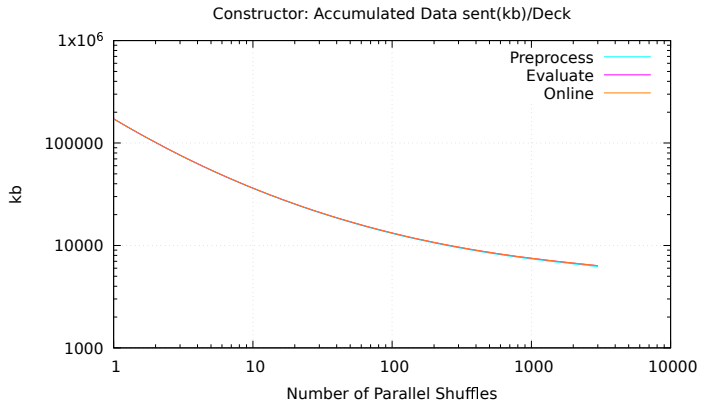
A bash script was setup to allow for automatic testing of all these different circuits, this is explained in A.4.3. This script ensured that ten timings was done for each of the circuits to guarantee a more fair result when taking the average. All the timings were logged and can be found via appendix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

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

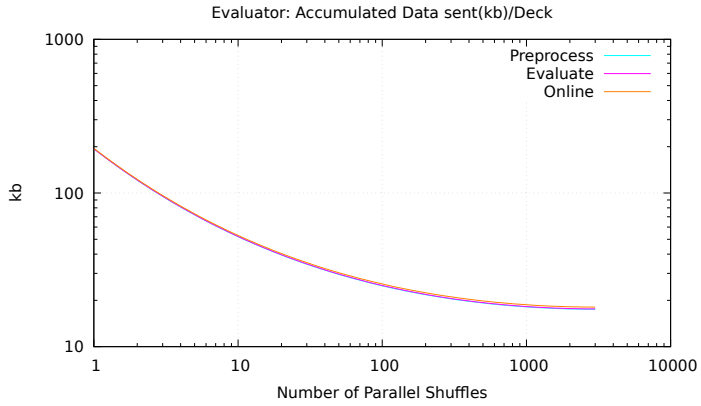
What we will expect to see in the results is that the more simultaneous shuffles we do, the faster run times and lower data communication per shuffle. We remember that in the *Preprocess* phase of the *DUPLO* protocol a lot of communication is done to exchange keys and commitments. Therefore, we see an overhead here compared to other protocols. The expectation is, that by doing many simultaneous 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 run time and data communication. The benchmarking for the data sent can be seen in figure 4.2 and the run time 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 data sent per deck shuffled. The data is represented on a double logarithmic scale, to be easier to see the development. It is easy to see, that as more simultaneous shuffles are done, the less data is sent per deck shuffled. This imply that the most data sent, is because of the overhead of

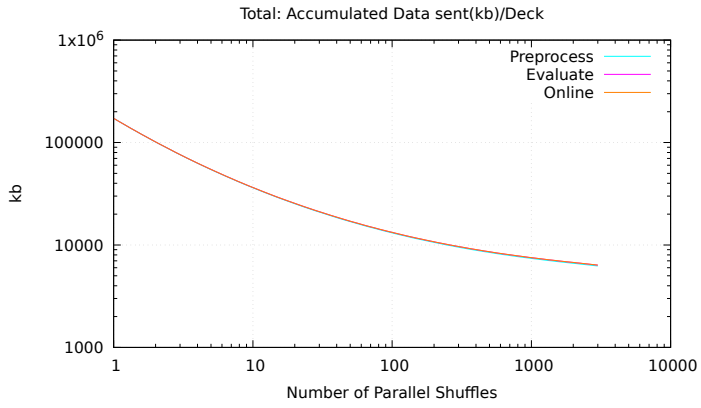
[1] A statistic of bandwidth speed in Denmark is found from 2015 here `https://www.statista.com/statistics/593964/average-internet-download-speed-by-connection-type-in-denmark/` It is fair to assume that the bandwidth rate has gone up since 2015.



(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 double logarithmic scale. (b) *Evaluator*: Accumulated data sent per deck shuffled on a double logarithmic scale. (c) *Total*: Accumulated data sent per deck shuffled on a double logarithmic scale.

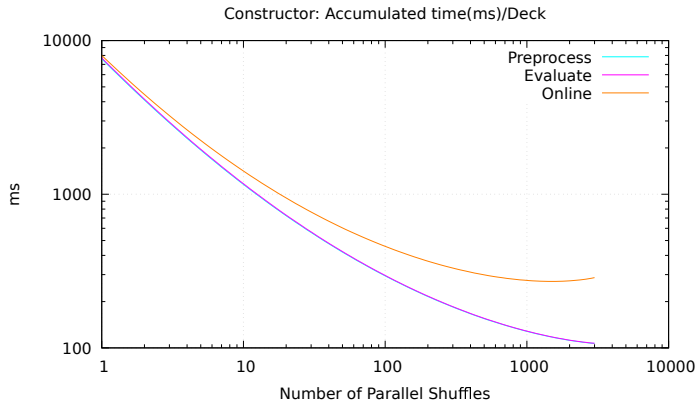
setting up the protocol. As we see it is hard to distinguish the different graphs in the figure. This is because, as expected, that the *Preprocess* phase is the phase where the most data is sent between the parties. Relative to this nearly no data is sent in the *Evaluate* and *Online* phase. Remembering that it is only the input to the functionality that is sent in the *Evaluate* phase, which is 830 bits for each shuffle. In the *Online* phase 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 not much more to gain in shuffling 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 simultaneous shuffles, can bring the amount of data sent per shuffle further 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 more 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 sends the most data and therefore the one that require the most simultaneous shuffles to bring the overhead of using *DUPLO* down.

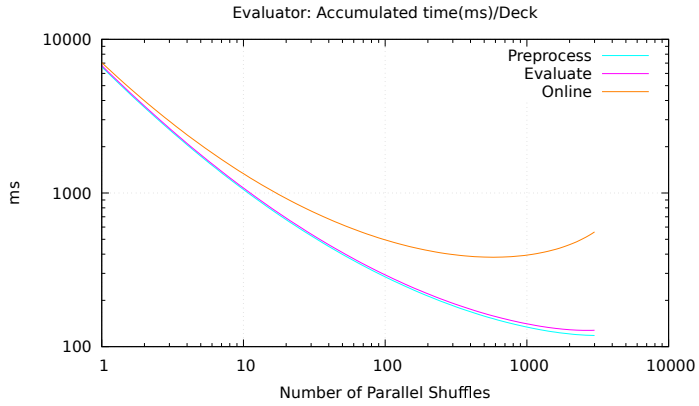
In this experiment we see exactly what we expected as described above. The amount of *kb* sent would decrease as more shuffles were done, because of the overhead of using *DUPLO* . Even going beyond the 3000 shuffles seems to give a decrease in terms of data sent. The gain of going beyond the 3000 mark 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 double 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 summation 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 approaching 100 ms per deck shuffled, when shuffling 3000 decks simultaneous. At the same time we see that the *Evaluate* phase does not add any significant time. It is a different scenario when looking at the *Online* phase. Here we see that the space between the *Evaluate* and *Online* graphs increase significantly, when approaching 3000 shuffled deck. This tells us that the *Online* phase uses more time per shuffled deck, when the amount of shuffles increases. 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 probably variate if tested on other hardware, as the hardware used here, are at it limits when doing 3000 shuffles.

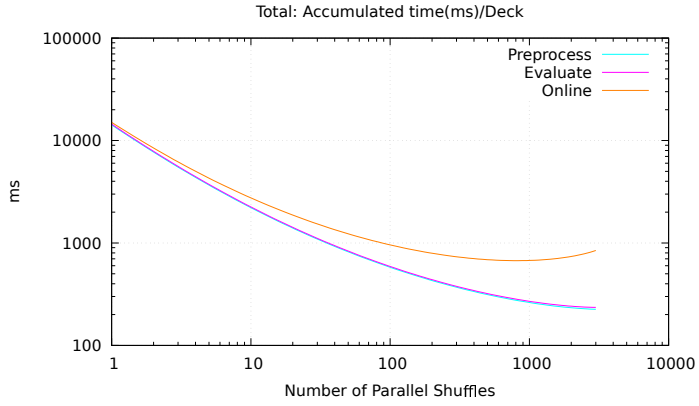
If we the look at the *Evaluator* in figure 4.3b, we see the same tendency as



(a)



(b)



(c)

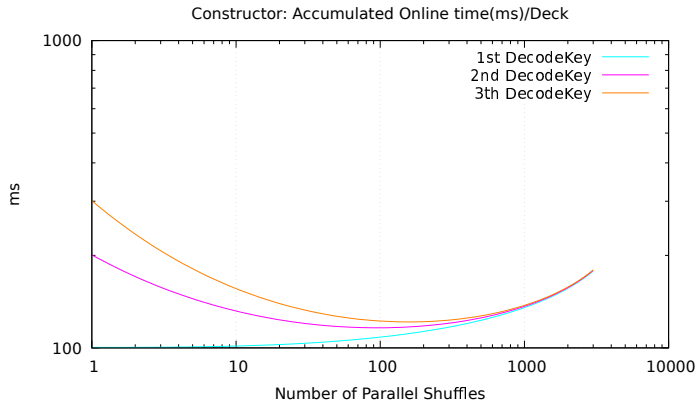
Figure 4.3: Time: Comparison of *Constructor* and *Evaluator* in *ms*'s used. (a) *Constructor*: Accumulated time per deck shuffled on a double logarithmic scale. (b) *Evaluator*: Accumulated time per deck shuffled on a double logarithmic scale. (c) *Total*: Accumulated time per deck shuffled on a double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

for the *Constructor*. The *Preprocess* is the one consuming the most of the time and here approaching 150ms per shuffled deck. For the *Evaluate* phase we see a small increase in time used per shuffle when passing the 1000 mark. This may be because of the increase in the size of the input to the algorithm. The total time used on these two phases seems to be slightly decreasing. Indicating that these phases can still gain some from doing more shuffles. When we turn our attention to the *Online* phase, we once again see an increase in the running time. For the *Evaluator* the increase 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 *Evaluator*, but with an optimum around 1500.

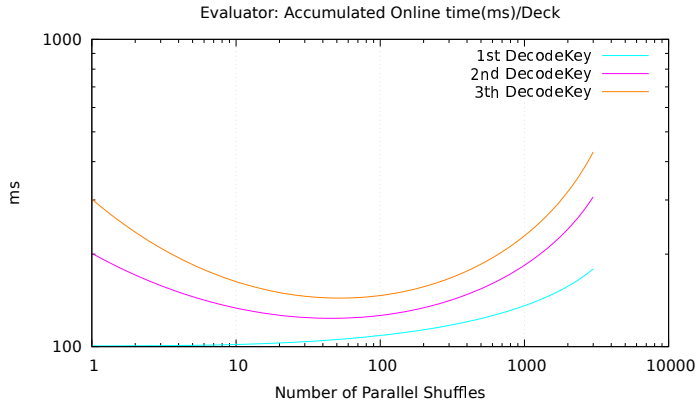
This was not the result we had expected. As we discussed earlier, we expected the running times to decrease through the complete graph. In the next section we will look into the result and try to come up with an explanation of these.

We will now take a closer look at the *Online* phase, to see there are some obvious reasons, why we get the results of an increasing *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 time used by the *Constructor* can be seen in figure 4.4a. Here we see a increase in the time used on the first `DecodeKeys` call, while the two other calls decreases as expected. This can imply that something 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 first decreases and then increase again when going towards the 3000 shuffles. When consulting figure 4.4c for the combined plot of *Constructor* and *Evaluator*, we see that the increase in time used by the first `DecodeKeys` call happened before 100 shuffles, while the increase by the other calls happens after 500 shuffles.

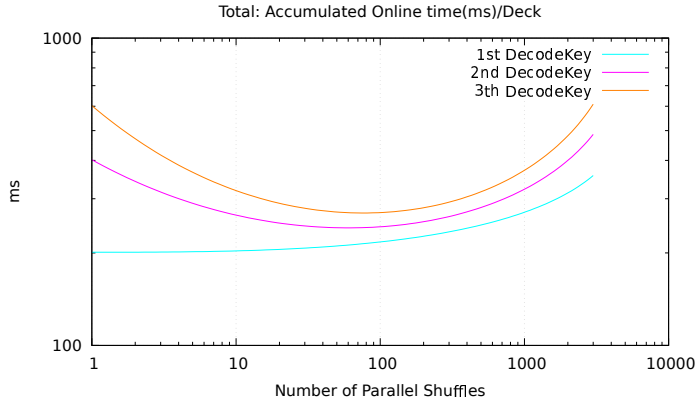
The increase in time spent on the `DecodeKeys` calls is probably from the fact, that the implementation tries to cache as much as possible. From figure 4.2 we see, that in the *Online* phase nearly no data is send. By consulting the data in appendix A.4.3, table A.1, we see that approximately 2.5kb is sent during the *Online* phase. Sending this amount of data on a network, with a bandwidth 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 anything from this, beside the fact, that this can be seen as a constant. Therefore, 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 the graph looks as it does for the *Constructor* may indicate, that some form of caching is taking place. We cannot say the



(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 double logarithmic scale. (b) *Evaluator*: Accumulated time per deck shuffled on a double logarithmic scale. (c) *Total*: Accumulated time per deck shuffled on a double logarithmic scale.

same about the *Evaluator*. It seems like some caching could take place 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 cache may be full and therefore an increase in time is happening, because it has to get the data for a slower memory.

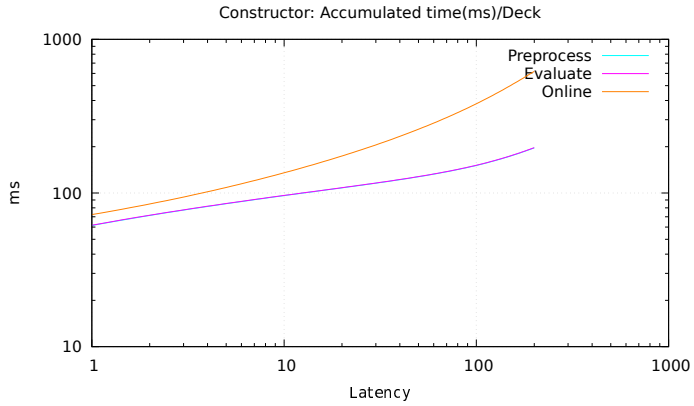
One thing that could support this hypothesis is, that it seems like the timings done by each of the `DecodeKeys` calls, seems to be more fluctuating when shuffling many decks simultaneous. This was indicated in a small test I did. This supports the idea that it might be I/O wait, that causes the decrease 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 takes around 100ms. If a call has to wait for 1ms, we will see an increase in time by 1%. The problem can also come from the fact, that when a machine run on high load, the performance go down. When running the 3000 shuffles test, it consumed the most of the memory on the test machine.

To test this hypothesis of a full cache a test run with the `-d` flag, for the ram only mode, was performed. This did not show any change in the running time other then the uncertainty of the test results. Another approach 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 limits to this project.

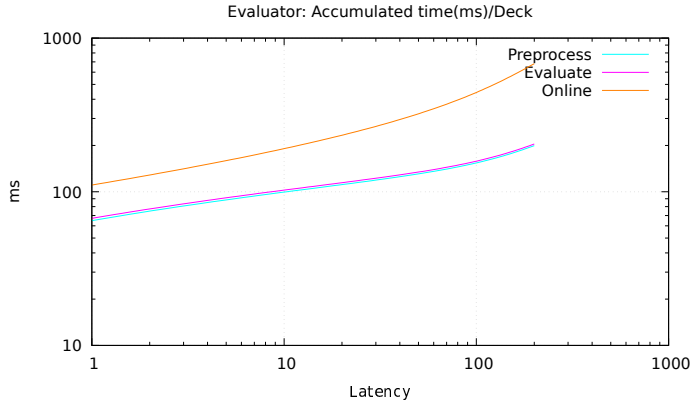
In the next sections I will look into the other things that might affect the protocol. Here it will be the latency on the network and the bandwidth. First off, we will start by looking at how the latency affects the *DUPLO* protocol. This can be seen in figure 4.5. The expectations are to see a decrease in performance when the latency 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 simultaneous shuffle circuit, as this was the constructed circuit closes to the optimum discovered above. The bandwidth 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 increasing 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 severely. As argued before the *Online* phase uses a significant amount of time on the network. Therefore an increase 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 latency's found there.

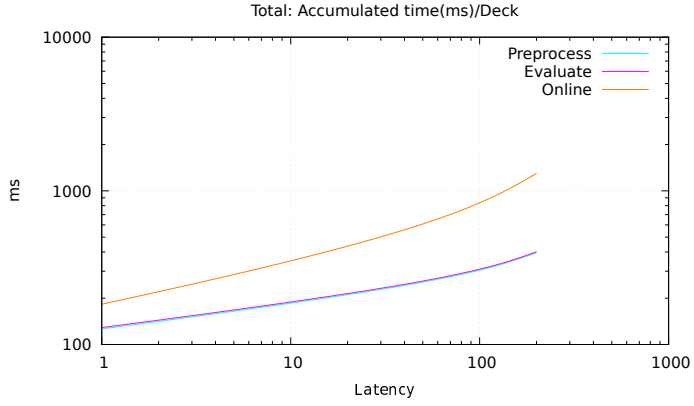
I will now turn the focus towards the benchmarking of the protocol against the bandwidth. To perform this test, the 1000 shuffle circuit was used once again. This time the latency was turned down to 0 to speed up the process. The expectations of the test, is to see a decrease in running time when the bandwidth



(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 double logarithmic scale. (b) *Evaluator*: Accumulated time per deck shuffled on a double logarithmic scale. (c) *Total*: Accumulated time per deck shuffled on a double logarithmic scale.

goes up. It is worth mentioning, that no tests are done with bandwidths slower then 50Mb/s, because the test machine crashed when trying to perform test below this threshold. If consolidating the statistic[2] a bandwidth of 50ms is within the once used.

The results of the bandwidth 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 leaves the *Evaluate* phase a bit to come closer again. This is most probably do to the uncertainty of the testing environment. Overall this follows the expectations we had.

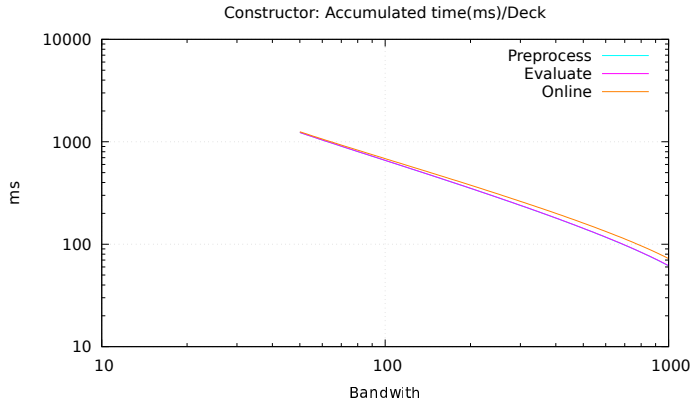
In this section I have tried to cover the tests done and argue for how they were performed. I have discussed the results we have seen and tryed to argue for why they are as they are. If the results are not as expected I have tried to cover these areas with new results within the time limit of the project. In the next section I will try to hold the results from this section up against what can be expected in terms of running times in online poker.

4.3 Discussion

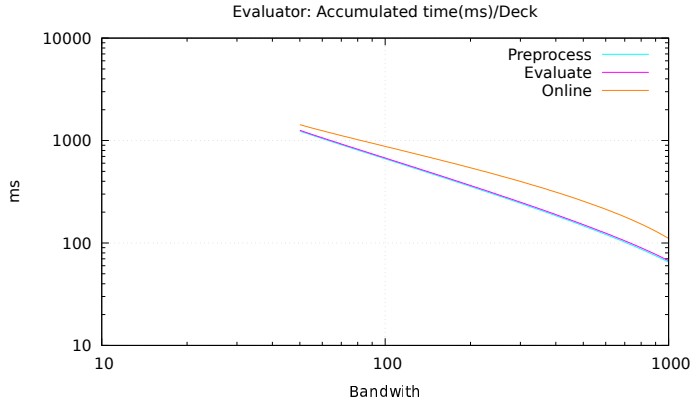
In this section the benchmarking done in section 4.2 will be compared to what can be expected in the real world. I will discuss both settings of the poker game introduced in section 4.1. This will be done because some of the settings used when testing, fits best in one setting and others in the other.

The first setting that will be discussed, is the setting implemented, where the *Constructor* and *Evaluator* both play the part of server and client. Before comparing this setting to the real world an discussion of the different parameters used will be done. If we start by looking at the latency used in figure 4.3. This was done using 50ms delay, compared to table 4.1 it seems to be a fair latency between two player in the real world. The bandwidth in this setting was set to 1024Gb/s, as argued above this is too high when compared to statistics[3]. Correcting the results according to this will imply, that more time would be added to the running time. This can be done by combining the results from figure 4.3 and figure 4.6. If we use the timing of the50Mb/s, then when doing 1000 shuffles we would have to add around 1 second to the running time per deck shuffled. This will yield a total running time of around 1.2 second per deck for the *Constructor* and 1.5 for the *Evaluator*. Here it is important to remember, that the time used in the *Preprocess* phase, can be done ahead of time. It may also be fair to argue, that for two player to play 1000 hands may be to the high site, as the preparation of the protocol can be done in a cycle, such that the preprocessing is done, when they do not play. By reducing the amount of shuffles the running time per deck will go up, but the total time may go down. We can then argue, that the hardware used, is to the extreme

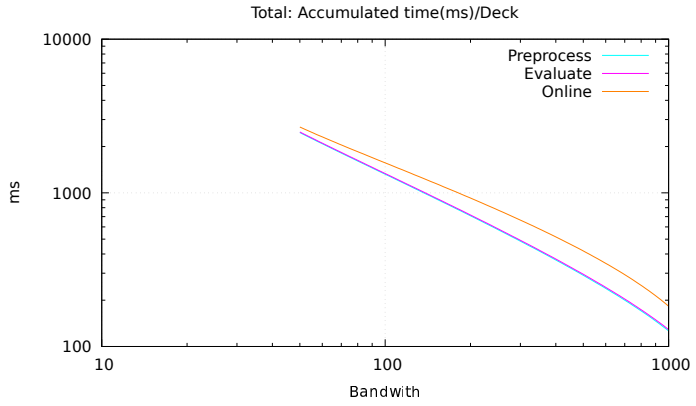
[2]See link in footnote 1 on page 37.
[3]Same as in footnote 1 on page 37



(a)



(b)



(c)

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

end when comparing to ordinary machines. This may imply that the optimal number of simultaneous shuffles will be reduced and therefore fit better with the discussion about 1000 shuffles being too many. This may yield an optimum around 500 shuffles. From the test done, it is not possible to conclude on the exact time of such a game, but an estimate would be below 1.5 seconds per shuffle, in total 1000 seconds which is around 13 minute. Consolidating the graph the *Preprocess* phase uses approximately the half of this time, in this case around 7 minutes. By doing this preprocess the players are then able to play 500 games of poker, with a running time of 6 minutes in total. This yields a running time of just below 75ms per game. Here we remember that the tests are done automatically without any change of cards, but when two players are playing against each other the decision of choosing cards would be significantly longer then this. Therefore, we may argue that performance of the poker implementation is good, if the time for the preprocessing is not too high. In this setting we argue that on a consumer computer it will take around 7 minutes to preprocess 500 decks. Two players of this type, must be assumed to leave their computer for more then 7 minutes a day, and therefore may the protocol actually do well in this setting.

In the other setting where the *Constructor* and *Evaluator* plays the role of servers, but not clients themself, we may argue that we will see some other results. Once again the latency used during the test seems to be fair. The bandwidth used during the test may also be fair in this setting since servers prioritize a higher bandwidth to handle a big load. In this setting we may argue that 1000 shuffles is to few. The 1000 shuffles takes around 1.2 seconds for the *Constructor* and 1.5 for the *Evaluator* per deck shuffled as argued above. Yielding 20 minutes for the *Constructor* and 25 minutes for the *Evaluator*. We may argue, that a poker provider use more then 1000 decks in 25 minutes, and the fact that no games can be played while the servers is in the *Preprocess* phase. This problem can be handled by having two sets of servers, one preprocessing enough games to last the time it takes for the other set of servers to do their preprocessing. Then we may argue, that servers of poker providers have more computation power then the hardware used during the test and therefore can handle more simultaneous shuffles and may be able to produce a greater amount in the same time. In this setting, it is hard to argue whether the protocol can handle the task or not. In this setting we do not have the ability to do the preprocess when no game is ongoing, but the phase where the clients was dealt the cards are still acceptable, as it will be hidden in the choice of cards to be changed as argued above.

All in all, we seem to be able to draw the conclusion that the poker game implemented using *DUPLO* in this project, can actually handle real world scenarios. The biggest concern is to find the time to do the preprocessing, but is a setting used where the time to the preprocessing can be found, then is the running time in the other phases acceptable.

In the next chapter, I will do a summary of the project and reflect the conclusion done in this section. I will introduce some future work, that could be done, to see how things stack up in different scenarios and why I got the

results I did.

Chapter 5

Conclusion

In this chapter, I will try to do a summary of the project and conclude on the results found in chapter 4.

In this project I have studied the application of using the *DUPLO* framework and protocol in the setting of a poker game. This was done to see how using a two party computation protocol will affect the overall running time and see if the protocols are mature enough to be used in real world situation. To do this the *DUPLO* protocol was chosen, which is a secure two party computation protocol using garbled circuits. This was chosen, because it has the possibility to do unique single wire opening, the existence of a circuit compiler and the fact that is developed at Aarhus University. Because of the security of *DUPLO* we are guaranteed to be able to provide privacy, correctness and independence of input. Privacy is the ability of the protocol, to ensure that nothing more than the output is leaned. Correctness it the ability of the protocol to guarantee, that the right output is provided to the parties. Independence of input is the ability of the protocol to guarantee, that the input of one of the parties cannot depend on the input of the others, because these properties is given from the protocol, we can start looking at the decision that should be made before the implementation could begin.

First a decision was made on the setting of how the protocol should be used. This could either be a setting where the two parties of the protocol acted as both server and client, or one where they only acted as server. The setting where the both acted as server and client, was the one chosen mainly because of its simplicity and the time limit of the project. The other setting is proposed as a future work below for others to look into.

A decision of which type of poker game to implement also had to be done. Here the five card draw variant was chosen. This was done mainly, because it is the variant of poker used in the most games when only two players participate. When only two parties participate it allowed for some optimizations of the shuffle algorithm, such as a smaller circuit could be produced for evaluation in the protocol.

To be able to use the *DUPLO* protocol in the poker implementation, we needed to generate the functionality that should be used. In this case it was the shuffling of the cards, that should be handled by *DUPLO* . We looked into

different algorithms to compare their circuit size, to be able to choose the one giving the fast's run time of the protocol. A compiler was used to generate the circuits, because the complexity of circuits is high, when the complexity of the functionality implemented gets higher then simple. This allowed for a higher level of abstraction. The *Fisher-Yates* and *Conditional-swap* algorithms was then implemented, both in a non-optimized and optimized version. This was done to see the affect of only shuffling the cards necessary, when only two players participated in the poker game. Based on the comparison of the shuffle algorithms the optimized version of *Conditional-swap* was the one chosen.

The implementation of the game could then start. The *DUPLO* protocol stated the order of the framework calls. The main purpose of the implementation was to generate the inputs to the protocol at the right time. Because of the way the *DUPLO* framework had splitted the functions, three different phases were made. The first phase was based on the functionality of the circuits where this was garbled, such that it could be securely evaluated. This is denoted the preprocessing phase, as it can be done only based on the circuit generated. The second phase was the evaluation phase, where the input to the functionality was given and the garbled circuit evaluated. The last phase was the online phas. This was the phase where per player would be shown their hand, chose which cards to change and see the opponents cards. No betting or declaration of a winner was implemented, because of the time limit to the project. The betting functionality is proposed as future work, as it could be interesting to see if this could be done using another secure protocol. When the implementation was in place the testing could begin.

During the benchmarking different parameters were tested to see how the *DUPLO* protocol handled these settings. Thie was done because the *DUPLO* protocol should perform better when working with big circuits. Different variants of the *Conditional-swap* shuffle was generated where different amount of shuffled decks was generated. This was done to see how big the benefit was of doing many simultaneous shuffles. The expected result, was that it would perform better and better, but this was not the case. We saw that there existed an optimum for the protocol. This is probably because of the internal optimizations of the protocol, where it tries to cache as much as possible and when the limit of the cache is exceeded the performance decreases. The optimal number of shuffles was then used to test how the protocol reacted to different amounts of latency on the network. The expectation was that it would have a linear slowdown in performance, when the latency on the network went up. This was also the results we got when performing the benchmark. The optimal circuit was then also used to test how the bandwidth affected the protocol. Here the expectations was, that we would see a linear increase in performance when the bandwidth went up. One again it performed as expected. When the benchmarks are done the running times from these were then compared to what could be expected of a real world poker game.

When looking at the numbers for the settings studied in the project, we may conclude, that the poker implementation using the *DUPLO* protocol can achieve running times that are good. When discussing the benchmarks the numbers of playing 500 hands of poker between to players yields 7 minutes

used on preprocessing and evaluation of the circuit and less then 75ms for each game played. The 75ms is negligible when comparing to the time used by the player to decide which cards to keep and which to change. Furthermore, the time is determined by the network latency, which is 50ms in this estimate of time. Therefore all in all the performance achieved is good, but the most online poker games are not played in this setting. Therefore an argument was done to try to compare it to that setting. Here a down period for preprocessing the shuffles cannot be tolerated. This can be handled by letting two systems run in parallel. Here one will preprocess shuffles while the other deals cards. This setup is only feasible if the time it takes for a server to preprocess the shuffles, takes less time the for the players connected to the server dealing cards, to use all the preprocessed decks. This have not been studied, but as argued in the discussion on benchmarking the online timings are feasible enough for the protocol to work in this setting. The remaing questing is, if the latency, bandwidth and hardware of online poker providers allows them to preprocess shuffles fast enough. This cannot be concluded, before tested in such a setting, but the performance of secure two party computation are closing the gab to be feasible in real world applications.

In the next section I will come up with some proposals to further work that could be done based on this project and the *DUPLO* protocol.

5.1 Proposed future work

Here I will propose some interesting aspects I have touched during the project, but have not had the time to look into. Each of the proposals should be possible to cover in the time of a project. Some of them might not be for a thesis, but could be used as a preproject to get into the *DUPLO* protocol and the working mechanisms of a *MPC*. I will give each project a short paragraph on why this could be interesting and how this could be done.

DecodeKeys running time coverage: In this project the development of the running times of the `DecodeKeys` call could be studied. It could be interesting to get a deeper understanding of why we see the development in the graphs in figure 4.4, as we do. This could be done by going in to the framework and time the different calls made inside of the `DecodeKeys` function. It could be interesting to see if the hypothesis of the full cache is the reason or something else. It could also be interesting to see the differences at the two parties. Why do the timings differentiate in such a way they do.

***DUPLO* in a server setting:** It could be interesting in this project to study how the *DUPLO* protocol would perform in the setting, where the *Constructor* and *Evaluator* act as servers and not clients at the same time. How would the authenticity and correctness of the inputs be guaranteed. Would the addition of these properties add an extra overhead to the *DUPLO* protocol making it perform significantly worse, or will it handle it well.

Five card draw vs. Texas Hold'em: This project could be interesting to see especially in the server setting, where more then two player could participate. Overall it could be interesting to see how much the difference in the optimization of the circuit will affect the running times. In a two party setting only 9 cards is used in Texas Hold'em compared to the 20 in five card draw.

Addition of betting: Betting is a big part of online poker, therefore could it be interesting to add this to the implementation. This could be done by using smart contract in block chain. by doing it this way the money will only be transferred if a condition is satisfied. Some but not all cryptocurrency protocols allows for these contracts, but I think it should be possible to do.

Appendix A

Code base

In this appendix references will presented to the different code bases used during the thesis. An URL to the repositories on GitHub will be presented together with a short description of where the most interesting 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 simultaneous shuffles. Therefore another setup 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 simultaneous shuffles, which are the highest done in the testing phase. When going up to 4000, this setup ran out of memory 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 clearly illustrates, 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/duplo-mains` is where the actual implementations of the *Constructor* and *Evaluator* 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 on docker hub[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 easy way to let the container running the *Constructor*, expose the port on which the container running the *Evaluator* needs to connect. When running two instances of these docker containers the *Constructor* and *Evaluator* are 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 sub-repository 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 access is done for example. It is here all functionalities that are implemented in the language are listed and how they are used. This documentation is not well written 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 hub[4] and running it, in docker version `17.05.0-ce` with the followin command;

[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
```

This will start a container where it is possible to compiler a `.wir` file using the container. For this to work the `.wir` file has to be located in the `host/dir` directory. If this is the case, the following command 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* generated file will have the extension `.wir.GC_duplo`. This can then be fed 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 functionality will run in the default *DUPLO* environment.

A.4 Poker

In this section the GitHub repositories to the different phases will be linked. A short description to where the interesting parts are, will be presented.

A.4.1 Circuit implementation

The different circuit `.wir` files can be found 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 played. 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 slower 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. This hard wires the card values to their respective wires in the circuit. The `corrected_seed.wir` file is used by the *Fisher-Yates* algorithm to ensure, that the seed fed to the shuffle algorithm is in the correct intervals as explained in section 3.2. The `xor_seed.wir` file is used by the *Conditional-swap* algorithm to 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 implementation

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 folder their shared functionality is found in the `poker-mains.h`.

Back in the main directory of the repository, the docker files are found for generating the docker instance 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 are used to start the docker containers correct, such that they 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 containers 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 access 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` flag is required together with the `-f` flag. This flag needs to reflect the number of simultaneous 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 repository 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.

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

(a) *Constructor*

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

(b) *Evaluator*

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

(c) Total

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