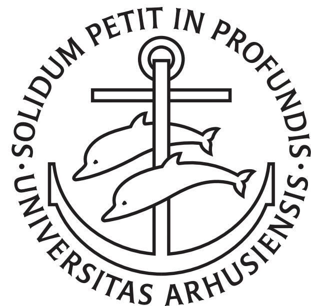
The Hitchhiker's Guide to Garbled Circuits

Garbled Circuits and Their Applications to Maliciously Secure Two-Party Protocols

Tore Kasper Frederiksen

PhD Dissertation



Department of Computer Science
Aarhus University
Denmark

The Hitchhiker's Guide to Garbled Circuits

Garbled Circuits and Their Applications to Maliciously Secure Two-Party Protocols

A Dissertation
Presented to the Faculty of Science and Technology
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Tore Kasper Frederiksen
November 11, 2015

Abstract

In the last few years garbled circuits have been elevated from being merely a component in Yao's protocol for secure two-party computation, to a cryptographic primitive in its own right. In this thesis we expand on the area of garbled circuits and their applications even further.

We take our departure in the formalism of garbling circuits as garbling schemes specified by the security properties privacy, obliviousness, and authenticity, as introduced by Bellare *et al.* in their seminal papers from 2012 and 2013 presented at CCS, Asiacrypt, and IEEE Security and Privacy. We augment their definitions slightly to better suit the use of garbling schemes in maliciously secure cryptographic protocols. We then give the following main results:

- We construct more efficient garbling schemes specifically under the assumption that the values floating on the wires of the circuit are known to the evaluator. As a highlight of our result, in one of our constructions only one ciphertext per gate needs to be communicated and XOR gates never require any cryptographic operations. We argue that these schemes have practical usage, as they can for example be directly applied in the protocol of Jawurek *et al.* at CCS 2013 for zero-knowledge arguments of knowledge for unstructured languages (in NP), making their protocol even more efficient. In addition to making a step forward towards more practical zero-knowledge arguments, we believe that our contribution is also interesting from a conceptual point of view: in particular our garbling schemes achieve authenticity, but not privacy nor obliviousness, therefore representing the first natural separation between those notions.

- We present two protocols for maliciously secure two-party function evaluation, based on cut-and-choose of garbled circuits. The first of our protocols uses a "majority rules" approach to decide on the output of the computation, in case of divergence on the output of individual garbled circuits. In this protocol we furthermore introduce a novel construction in order to verify consistency of the garbled circuit constructor's input in a parallel and maliciously secure setting. Our second protocol uses the recent idea of "forge-and-lose", which eliminates around a factor 3 of garbled circuits that needs to be constructed and evaluated, compared to the majority rules approach. We do this by introducing a new way to realize the forge-and-lose approach, which avoids an auxiliary secure two-party function evaluation protocol, does not rely on any specific number theoretic assumptions, and parallelizes well in a same instruction, multiple data (SIMD) framework. We prove our second protocol universally composable-secure against a static and malicious adversary, assuming access to oblivious transfer, commitment, and coin-tossing functionalities in the non-programmable random oracle model. Finally, we construct, and benchmark, a SIMD implementation of both our protocols using a GPU as a massive SIMD device. The findings compare favorably with all previous general implementations of maliciously secure two-party function evaluation.

Resumé

I de seneste par år er forkludrede kredsløb blevet hævet fra bare at være en komponent i Yao's protokol til sikre to-parts beregninger, til at være et kryptografisk primitiv i sig selv. I denne afhandling videreudvikler vi området af forkludrede kredsløb og dets anvendelser.

Vi tager vores udgangspunkt i forkludrede kredsløb som forkludrings systemer fastsat af sikkerheds egenskaberne privathed, uvidenhed og ægthed, som introduceret af Bellare *et al.* i deres skelsættende artikler fra 2012 og 2013 præsenteret ved CCS, Asiacrypt og IEEE Security and Privacy. Vi forøger deres definitioner en smule, for bedre at understøtte brugen af forkludrede systemer i ondskabsfuldt sikre kryptografiske protokoller. Vi giver derefter de følgende hovedresultater:

- Vi konstruerer mere effektive forkludrings systemer, under den specifikke antagelse at værdierne der driver på ledningerne i kredsløbet er kendt af evaluatoren. Som et højdepunkt af vores resultat er det i en af vores konstruktioner kun nødvendigt at overføre en enkelt ciffertekst per gate, samtidig med at XOR gates aldrig kræver nogle kryptografiske beregninger. Vi argumenterer for at disse systemer har en praktisk anvendelse, da de for eksempel kan benyttes direkte i protokollen af Jawurek *et al.* fra CCS 2013 til nul-videns argumenter af viden for ustrukturerede sprog (i NP), til at gøre denne mere effektiv. Udover at nå et skridt nærmere mod praktiske nul-videns argumenter, mener vi, at vores bidrag også er spændende fra et konceptuelt synspunkt: vores forkludrings systemer opnår især ægthed, men ikke privathed eller uvidenhed, og repræsenterer derfor den første naturlige deling mellem disse koncepter.

- Vi præsenterer to protokoller til ondskabsfuldt sikre to-parts funktions udregning baseret på snit-og-vælg af forkludrede kredsløb. Den første af vores protokoller benytter en "flertallet bestemmer" tilgang for at beslutte resultatet af beregningen, i tilfælde af uoverensstemmelse mellem resultaterne fra de individuelle forkludrede kredsløb. I denne protokol introducerer vi desuden en ny konstruktion til at bekræfte overensstemmelse af konstruktøren af de forkludrede kredsløbs input i et parallelt og ondskabsfuldt sikkert miljø. Vores anden protokol benytter sig af den nylige ide af "forfalsk-og-tab", der fjerner omkring en tredjedel af de forkludrede kredsløb der er nødvendige at fremstille og udregne, sammenlignet med flertallet bestemmer tilgangen. Vi gør dette ved at præsentere en ny måde at gennemføre forfalsk-og-tab tilgangen, der undgår at udføre en ekstra sikker to-parts funktions udregning, der ikke afhænger af nogle specifikke tal teoretiske antagelser, samt paralleliserer godt i en samme instruktion, adskillige data (SIMD) struktur. Vi beviser vores anden protokol universelt sammensættelig-sikker mod en stillestående og ondskabsfuld modstander, antaget tilgang til uvidende overførsel, binding og mønt-kastnings funktionaliteter i den ikke-programmerbare tilfældige orakel model. Til slut fremstiller vi og benchmarker en SIMD implementering af begge vores protokoller ved brug af en GPU som et massiv SIMD

apparat. Vores fund sidestiller sig favorabelt med alle tidligere generelle implementeringer af ondskabsfuldt sikre to-parts funktions udregning.

Acknowledgments

I would like to thank my main advisor Jesper Buus Nielsen. First of all for letting me write my master's thesis under his supervision, and then encouraging me to continue that work as his PhD student. It has been great working with him during the past years. I would also like to thank my co-advisor, Ivan Bjerre Damgård, for co-advising me and making the crypto group at AU an extremely enjoyable place. On the same note I would like to thank everyone who has been part of Aarhus crypto group the last three years: You have all made doing a PhD at AU a great experience. I would also like to thank the people at University of Bristol's cryptography and security group for making me feel welcome during my 6 month research visit there.

I would like to thank all my other co-authors: Thomas Pelle Jakobsen, Marcel Keller, Peter Sebastian Nordholt, Claudio Orlandi, Emmanuela Orsini, Peter Scholl, and Roberto Trifiletti.

Furthermore, I owe great thanks and appreciate to the people who I have visited or done research with, which have not (yet!) yielded published papers: Carsten Baum, Payman Mohassel, and abhi shelat.

I would furthermore like to thank the people who have given useful feedback to all the papers I have co-authored: First of all to the various anonymous reviewers, but in particular to the Rasmus Winther (Lauritsen|Zakarias), Benny Pinkas, Mike Rosulek, and Nigel Smart, for useful discussions throughout my PhD.

Finally I would like to thank my family, friends, and room mates for believing in me and my girlfriend Ina Thegen for not running away screaming sometime during the past three years.

Tore Kasper Frederiksen,
Aarhus, November 11, 2015.

Dedicated to
Lise Røjlby Frederiksen,
Aase Røjlby Frederiksen, and
Tage Bent Frederiksen

Contents

Abstract i

Resumé iii

Acknowledgments v

Contents vii

I Overview 1

1 Introduction 3
1.1 Contributions . . . 10
1.2 Outline . . . 19

2 Preliminaries 21
2.1 Notation . . . 21
2.2 Universal Composability . . . 22
2.3 Cryptographic Building Blocks . . . 24
2.4 Garbling . . . 29

II Privacy-Free Garbling 43

3 Privacy-Free Garbling 45
3.1 Defining Our Garbling Scheme . . . 46
3.2 Our Privacy-Free Garbling Schemes . . . 55
3.3 Privacy-Free FleXOR . . . 66
3.4 Efficiency Improvements . . . 69
3.5 Zero-Knowledge Arguments of Knowledge from Garbled Circuits . . . 70

IIISecure Computation Using Garbled Circuits 73

4 Improved Cut-and-Choose of Garbled Circuits 75
4.1 Our Protocols . . . 76
4.2 Building Blocks . . . 80
4.3 The Full Protocol . . . 88
4.4 Proof of Security . . . 93

4.5 Parallel Approach 113
4.6 Implementation 114

Bibliography **127**

Appendix **139**

Overview of Variables **141**

Acronyms **143**

Index **145**

Part I

Overview

Chapter 1

Introduction

Imagine, if you will, the setting of a digital national election, where you want to cast a vote to a specific politician. However, you don't want to simply tell the government who you wish to vote for; you want to keep your vote *private*. Furthermore, you want to be able to verify that your vote has in fact been counted towards the final result; you want the tallying of votes to be *robust*. A solution to the problem is Secure Multi-Party Computation (MPC). All participating parties run a protocol together which computes some function you have all agreed on (in this case the tally of votes). The field was introduced by Andrew Yao in the 80's [Yao82] where the function computed was integer comparison. However, since then many different types of functions have been considered and protocols for both the computation of specific and arbitrary functions have been constructed.

In 2008 the first successful implementation of a MPC protocol in a practical environment was done [BCD+09]. MPC was used to implement a double auction for Danish farmers and the Danish sugar beet processing monopoly, Danisco, in order to decide on a market clearing price for units of sugar beet production contracts. The implementation was done based on three parties, assuming only a minority of these acted dishonestly. It was furthermore assumed that even a dishonest party followed the protocol. The computation required approximately half an hour of computation in their setting (based on 1229 farmers). Based on this result it is interesting to study how one might achieve even more efficient and more secure implementations of MPC. In this thesis, we will focus on the specific case where only two parties have input to the protocol, known as Secure Two-Party Computation (2PC), and wish to protect against a corrupt party who might deviate from the prescribed protocol in any way. Even more specifically we will consider the case of Secure Function Evaluation (SFE), where the computation is "one shot" and the function to compute is arbitrary. That is, the parties decide on the function to be computed, give input and learn their output. Thus it is not possible to have partial output throughout the execution and thus change the functionality reactively based this. Even though it might sound like a quite restrictive setting, it is very relevant in practice such as in the setting of computing set intersection, oblivious encryption, zero-knowledge arguments, and so on. Furthermore, it provides a simpler "interface" for protocol specifications and relies on the weak assumption that the majority (i.e. half or more) of the participating parties might be dishonest and act in any way they see fit.

Garbled Circuits

A Garbled Circuit (GC) [Yao82, Yao86, BMR90] is a cryptographic tool that allows one to evaluate "encrypted" circuits on "encrypted" inputs. Here a circuit is defined as a Directed

Acyclic Graph (DAG) where the nodes are called *gates* and the edges are called *wires*. A gate takes as input a constant amount of (usually two) wires and has one wire as output, called the *(gate) input wires* and *(gate) output wire* respectively. The amount of wires going into a gate is called the *fan-in*. The wire going out of a gate can be replicated to be used as input to several other gates, the replication is called *fan-out*.[1] The wires contain values from a set of potential values and a gate implements a specific function mapping the values of the input wires to a value from the same set, onto the output wire.

In the garbled circuit all these values are encrypted; meaning that the wires have a *plain value* (also called the *semantic value*) and a *label* (also called the wire-*key*) which is the encryption of the plain value. This makes it possible, given the label of the *circuit input wires* in a circuit (those which are not output wires of some gate), to obliviously evaluate the entire circuit, learning the label of the *circuit output wires* (those which are not input wires to a gate). Given a map, or another form of auxiliary information, these labels can be translated to their corresponding plain values, thus making it possible to evaluate a circuit without knowledge of the values floating on the internal wires of the circuit.

The generic construction of such kind of garbled circuits can be described as follows for fan-in 2 Boolean gates: Call the left input wire l, the right input wire r, and the output wire o. Any wire, w, has two possible key values: K_w^0 and K_w^1 which are (ideally) independent random values. Here K_w^0 represents the bit 0 and K_w^1 represents the bit 1. Concretely we assume they are κ bit strings, $K_w^0, K_w^1 \in \{0,1\}^\kappa$. If the bit on a given wire is 0, then the wire w will have value K_w^0, otherwise it will have value K_w^1. However, since the values K_w^0 and K_w^1 are random and independent, this implies that knowing the value K_w^a for some $a \in \{0,1\}$ of a wire does not give away which bit it represents. Of course each wire in a circuit generally does not have a static value and thus most wires will have different values depending on the input to the circuit when evaluating it.

For each gate in the garbled circuit we associate a *garbled computation table*. This table is used to find the correct value of the output wire of the gate. Assume the functionality of a given gate is defined by the function $G : \{0,1\} \times \{0,1\} \to \{0,1\}$. A bit more concretely $G(a,b) = c$ with $a, b, c \in \{0,1\}$, where a is the semantic value of the left input wire, b is the semantic value of the right input wire, and c is the semantic value of the output wire. For example, if a given gate is an OR gate then $c = 1$ for all values of a and b, except if both a and b equal 0. The garbled computation table is then a random permutation of $\mathrm{E}_{K_l^a}\left(\mathrm{E}_{K_r^b}\left(K_o^c\right)\right) = \mathrm{E}_{K_l^a}\left(\mathrm{E}_{K_r^b}\left(K_o^{G(a,b)}\right)\right)$ for all four possible input pairs, using some encryption function, $\mathrm{E}_{\mathrm{key}}(\mathrm{message})$. That is, the entries in the garbled computation table consists of "double encryptions" of the output wire's labels, where the keys for each double encryption correspond to exactly one combination of the input wires' labels.

However, we still need a way to know which of the four entries is the correct one to decrypt. For this several different approaches exist. One such approach is the usage of *permutation bits* [Rog91, BMR90]. The idea is to associate a single *permutation bit*, $\pi_i \in \{0,1\}$, with each wire, i, in the circuit. The value on this wire is then defined such that $\mathrm{lsb}\left(K_i^b\right) = e_i$ where $e_i = \pi_i \oplus b$ with b being the bit the wire should represent when we use $\mathrm{lsb}(\cdot)$ to denote the function returning the Least Significant Bit (LSB) of a bit string. We call e_i the *external value*. The entries in the garbled computation table are then sorted according to the external values. That is, if the external value on both the left and right wire is 0 then the message in the *first* entry of the table will be encrypted under these two wire keys. If instead the external value on

[1] In the literature the fan-out might instead be the amount of output wires with potentially distinct values going out of the gate.

the right wire is 1, then the key in the *second* entry of the table will be encrypted under the left and right wire keys. Thus, we view the external values as a binary number, specifying an entry in the garbled computation table. More formally, if the left input bit is a and the right input bit is b, then the key $K_o^{G(a,b)}$ is encrypted in entry $e_l = \pi_l \oplus a,\ e_r = \pi_r \oplus b$ in the garbled computation table. This means that given the keys of the input wires the evaluator can decide exactly which entry he needs to decrypt, without learning anything about the bits the input wires represent. This follows since $e_l = \text{lsb}\left(K_l^a\right)$, $e_r = \text{lsb}\left(K_r^b\right)$.

One fundamental optimization of garbled circuits, making it possible to evaluate XOR gates for free is called *free-XOR* and was introduced by Kolesnikov and Schneider [KS08]. Free here means that no garbled computation table needs to be constructed or transmitted, and no encryption needs to be done in order to evaluate such a XOR gate. The only thing we need to do is to put a constraint on the way the wire keys are constructed. The constraint is very simple, assuming that we wish to construct the keys for wire i, then it must be the case that

$$K_i^1 = K_i^0 \oplus \Delta \ ,$$

where Δ is a *global key* (or *global difference*), used in the keys for all wires in the garbled circuit. Now, in order to compute a XOR gate we simply XOR the values of the two input wires of the gate, that is:

$$K_o = K_l \oplus K_r \ .$$

Regarding the external values, we need the constraint that $\text{lsb}(\Delta) = 1$ and thus notice that

$$\text{lsb}\left(K_o\right) = \text{lsb}\left(K_l\right) \oplus \text{lsb}\left(K_r\right) \ .$$

Finally, see that a row of the garbled computation table can be eliminated using the *garbled row reduction* approach of [NPS99, PSSW09]: Remember that a single Boolean gate takes up four entries of κ bits. However, we can simply define one of the output keys to be the ciphertext of the 0-string encrypted under a pair of input keys. This key pair is the one where both the external values are 0, i.e. $e_l = 0$ and $e_r = 0$. In short, we must define one of the output keys, and implicitly its external value, as follows:

$$K_o^{G(\pi_l \oplus 0, \pi_r \oplus 0)} = \text{E}_{K_l^{\pi_l \oplus 0}}\left(\text{E}_{K_r^{\pi_r \oplus 0}}\left(0^\kappa\right)\right) \ .$$

Depending on the type of gate, this again uniquely specifies the permutation bit of the output wire from this calculation:

$$\pi_o = e_o \oplus G\left(\pi_l \oplus 0, \pi_r \oplus 0\right) \ .$$

The three remaining entries in the garbled computation table are then the appropriate encryptions of the output key or the output key XOR'ed with the global difference Δ.

The descriptions above is simply an outline of *one* type of garbled circuits. This type, and several others, have been used in a number of different contexts within the area of cryptography, some of which we will consider in this thesis. We will give a survey of the "evolution" of garbled circuits in Chapter 2 along with formal definitions.

Secure Two-Party Computation

2PC is the area of cryptography concerned with two mutually distrusting parties who wish to securely compute an arbitrary functionality with private outputs, based on their independent and private inputs. A bit more formally call one party Alice, let her be denoted by A. Call the

other party Bob, let him be denoted by B. We now say that A has a string of input values x_{A} and B the string of input values x_{B}. We let n_{A} and n_{B} denote the length of x_{A} and x_{B} respectively. Furthermore, we let x denote the concatenation of x_{A} and x_{B} and use n to denote the length of x, meaning that $n = n_{\mathsf{A}} + n_{\mathsf{B}}$. We assume that A and B wish to compute a function f taking x as input and returning a string, y, as output and let m denote the length of y. Who should learn the output y can differ and in the most general case there might be a specific output for A and one for B, i.e., $f(x) = y = (y_{\mathsf{A}}, y_{\mathsf{B}}) = (f_A(x), f_B(x))$ where A should only learn y_{A} and B only y_{B}. In the most general setting the computation might be reactive, so that the parties may learn some plain outputs through the execution and then supply more input, dependent on these. However, in this thesis we only consider the case where the computation is non-reactive. That is, the setting of SFE.

Now, we want to compute the function f *securely*. As a minimum, we can assume this means that the computation should be *correct* and *private*. Correctness means that what is computed is actually the function agreed upon, using the private inputs chosen by A and B respectively. Privacy means that the information learned by A, respectively B, should only be the information specified by their respective parts of the output string y, that is y_{A} for A and y_{B} for B.

2PC was introduced in 1982 by Andrew Yao [Yao82], specifically for the *semi-honest* case, meaning that a dishonest party will follow the protocol, and only try to break security afterwards, using his or her transcript. Yao showed how to construct a semi-honestly secure protocol using garbled circuits, where one party (*the constructor*), say A, garbles a Boolean circuit computing the desired functionality.[2] The garbled circuit is then sent to B. Then, for A's own input, she sends to B the keys whose semantic meaning matches her input. For B's input, they use an Oblivious Transfer (OT) protocol such that B learns one key for each input wire whose semantic meaning matches his input, without A learning B's input. Now, given one key for each input wire, B can then evaluate the whole garbled circuit, gate by gate, when he reaches the output wires, he uses some auxiliary information to learn the semantic values of the output keys. We notice that in this case, and in fact also in the rest of this thesis, B will be the only party learning output from a garbled circuit. Still, if output for A is also needed then any of several general approaches can be used [LP15, SS11]. For a more thorough description of Yao's scheme see [LP09].

Using Garbled Circuits for Practical Secure Computation

A major reason why Yao's original protocol is only secure against a semi-honest adversary is that B cannot be sure that the garbled circuit he receives from A has been garbled correctly. However, we often desire security in the *malicious* setting, where a dishonest party tries to break the security during the execution. That is, he might not follow the described protocol. The first step towards getting security against such an adversary is to ensure that the garbling has been done correctly. One might do this by using a *cut-and-choose* approach [Pin03]: Instead of sending one garbled circuit, A sends several independently garbled versions of the circuit to B. B then randomly selects some of these, called the *check circuits*, where all the keys on the input wires are revealed to B, allowing him to verify that the check circuits do indeed compute the correct function f. If enough circuits are checked and all found to be correct, a large fraction of the remaining circuits, called the *evaluation circuits*, will also be correct except with extremely small probability.

[2] This is a historical oversimplification, which we will remedy with more details in Section 2.4.

Doing cut-and-choose on several garbled circuits introduces some other issues that have to be dealt with in order to obtain malicious security. First, there may still be a few incorrect circuits among the evaluation circuits. Also, since there are now many circuits, we must add some mechanism to ensure *input consistency* amongst the circuits: A and B must give the same input to all the circuits. Information about A's input can leak to B if he gets to evaluate the circuit on different inputs. But information might also leak to A, depending on the function the circuit is computing, if she gives inconsistent input. Another, more subtle, issue that arises, not just in the cut-and-choose case, but the general setting of OT where the receiver's choice depends on his private input, is a *selective failure attack*: Since A will supply B with the keys in correspondence with his input bits through an OT, A can simply input garbage for one of the keys, e.g, the 0-key in the first OT. If B aborts the protocol, because he cannot evaluate a garbled circuit when one of the keys is garbage, A will know that the first bit of his input is 0! On the other hand, if he does not abort the protocol then his first bit must be 1! This type of attack was first pointed out by [KS06, MF06] independently of each other.

When using cut-and-choose of garbled circuits to implement two-party SFE, in most cases the majority of the execution time is linearly dependent on the amount of garbled circuits being constructed and sent. We call the amount of circuits that needs to be constructed the *replication factor*. Most literature up to 2013, is based on an analysis saying that if the fraction of check circuits is high enough and if the check circuits are all found to be correct, then a *majority* of the remaining evaluation circuits will also be correct except with exponential small probability in the total amount of circuits constructed. This allows B to compute the correct output as the majority of the outputs from the evaluation circuits.

Recent results [Bra13, HKE13, Lin13] manage to ensure that A only succeeds in cheating if *all* check circuits are honestly constructed and *all* evaluation circuits are maliciously constructed. This happens with noticeably smaller probability than only the majority of the evaluation circuits are honestly constructed, thus making it possible to run a protocol with noticeably less garbled circuits in order to achieve the same probability of failure. The main idea of [Bra13] and [Lin13] (called "forge-and-lose") is to make sure that if A cheats, that is, if two evaluation circuits yield different results, then this enables B to learn A's private input x_{A}. He can then use x_{A} to compute locally, unencrypted, the correct output $f(x)$.

In [NO09] Nielsen and Orlandi introduced the notion of cut-and-choose at the gate level. The individual garbled gates remaining for evaluation would then be "soldered" together into maliciously secure fault tolerant "buckets", each computing a Boolean gate. These buckets would then be soldered together to form a circuit computing the desired functionality. They called this approach Large Efficient Garbled-circuit Optimization (LEGO). LEGO gave an asymptotic increase in efficiency of $\Theta(\log(|f|))$ where $|f|$ is the amount of gates in the circuit to compute. However, their approach relies on computationally heavy, group based operations (as the protocol required homomorphic commitments) for each gate in the circuit. The LEGO idea of combining elements, checked through cut-and-choose, into fault tolerant buckets was extended to consider whole garbled circuits in [LR14, HKK$^+$14]. Specifically this means that if two parties wish to compute the same function, n times, they will get an asymptotic improvement of $\Theta(\log(n))$, compared to executing n instances of a standard cut-and-choose of garbled circuits protocol.

If one is willing to accept a weaker kind of "malicious" security one can get significant improvements by using the *dual execution* approach, where only a single garbled circuit is constructed and evaluated by *each* of the parties [MF06, HKE12, MR13].

Work on a general version of garbled circuits for the multi party case (where more than two parties might be involved), was started in [BMR90] and continued in [LPSY15] for the malicious

setting. The overall idea of these works is to have each wire key consist of the concatenation of a seed associated with each party. Each party then uses a Pseudorandom Generator (PRG) to expand their respective seeds for each wire, into a one-time pad. Each entry of a garbled gate will then be the XOR of each pad (from its input wires) and the key of the output wire (which will again be the concatenation of a seed for each party).

For the three party case, the recent work of Choi *et al.* [CKMZ14] considers an efficient instantiation, by having two out of the three parties emulating the constructor of an instantiation of a modified version of Yao's protocol.

Other Approaches to Maliciously Secure Computation

Several other approaches to maliciously secure computation exist based on other constructions. Some of the first and most famous approaches include the GMW [GMW87] and the BGW [BGW88] protocols. Both these papers present, first semi-honestly secure protocols for secure computation with any number of parties, next protocols for secure computation with any number of parties which are maliciously secure. The GMW protocol considers dishonest majority (where half or more of the parties act dishonestly), in particular including the two-party setting. Each non-XOR gate is computed by a small interactive protocol based on one-out-of-four OTs. To make the protocol secure in the malicious setting, a compiler is presented which uses commitments, coin-tossing, and Zero Knowledge (ZK) proofs as subprotocols. A bit more specifically the parties commit to their inputs, and then execute an augmented coin-tossing protocol. The output of the coin-tossing is a uniformly random string, given to one party, along with a commitment to this string, which is given to the other party. The parties then run the semi-honestly secure protocol, using the random string and prove in zero-knowledge that they have done so correctly. The idea of using secret sharing is also used in the BGW protocol. However, this protocol works on arithmetic circuits in finite fields, and not just bits. Furthermore, the BGW protocol considers perfect security and requires an honest majority.[3] Their point of departure for the semi-honest case is to use Shamir's secret sharing scheme with a threshold $t < n/2$ where n is the amount of parties in the protocol. Using this scheme the parties share their inputs. Since Shamir's secret sharing scheme is linear, addition can be computed locally. For multiplications, the parties multiply their shares together locally to achieve a secret sharing of the product, but where the polynomial representing the shares is of degree $2t$. They then execute a subprotocol to reduce the degree of the polynomial representing their shares to one of degree at most t. For the malicious case a Verifiable Secret Sharing (VSS) scheme is used instead, with threshold $t < n/3$.

Another approach to secure computation is the idea of "MPC-in-the-head" from [IKOS07, IPS08] which combines an "outer" protocol that is maliciously secure against a dishonest minority, with an "inner" protocol which is semi-honestly secure against a dishonest majority. The execution of the outer protocol is simulated by using the inner protocol to compute each message communicated. The final protocol is then maliciously secure against a dishonest majority.

Yet another paradigm for secure computation is based on computation of secret shared values, where Message Authentication Code (MAC)s are used to verify correct computation. Within this family several distinct approaches exist: One approach works only on the computation of Boolean circuits. This approach, known as TinyOT, was introduced in [NNOB12], considering the case of only two parties. Later, in [LOS14, BLN$^+$15] it was generalized to the multi-party setting, remaining secure against a dishonest majority. This approach relies on a MAC on each

[3] At least two-thirds honest parties if considering the malicious case.

bit in the computation. If we wish to do arithmetic computation instead then other approaches can be used. One such approach is based on using secretly shared random multiplication triples. In this setting the MAC size is proportional to the message length (assuming it is at least the security parameter). A lot of work has been done in this setting lately, such as [BDOZ11, DPSZ12, DKL$^+$13, KSS13]. The protocols presented in the latter of these papers are called SPDZ protocols. Damgård and Zakarias [DZ13] (with work continued in [DLT14]) managed to port the amortized constant MAC size on each message bit of SPDZ protocols to computation of Boolean circuits. Their approach was to "pack" together several message bits and encode the packed bits using a linear error correcting code. A MAC on the codeword was then computed. If the code has constant rate then it is easy to see that each message bit has a constant size MAC in the amortized sense. This family of protocols is called MiniMAC.

A common feature of the secret sharing with MAC protocols above is the possibility of doing a computationally heavy preprocessing phase, where correlated randomness is constructed independently of both the input of the parties and the functionality they wish to compute. This random raw material is then used in a computationally light online phase. The difference in the time it takes to construct the preprocessing material and executing the online phase can be several orders of magnitude. In particular for SPDZ type protocols, where the preprocessing phase relies on semi/somewhat homomorphic encryption, and MiniMAC, where no explicit protocol for efficient preprocessing has previously been given. For TinyOT it is possible to do the preprocessing more efficiently since it relies on OTs, which can be done very efficiently using an OT extension.

Other Uses of Garbled Circuits

Besides SFE of Boolean circuits, garbled circuits can be used in many other cryptographic settings. For example to construct efficient garbled Random Access Memory (RAM) programs [LO13, GHL$^+$14], where the model of computation is random memory access and computation of instructions rather than a Boolean circuit.

Another example is the work of Kamara and Wei [KW13] who construct special purpose garbled circuits based on a specific type of searchable encryption schemes. These special purposes include branching programs and deterministic finite automata. Garbled circuits have also been used to construct Key Dependent Message (KDM) secure encryption [BHHI10]. More exotic types (based on more heavy cryptographic primitives) can be used to construct Non-Interactive Zero Knowledge (NIZK) proofs [AIKW15].

Garbled circuits can also be used as a component in the construction of one-time programs [GKR08, JKSS10]. That is, a function which can be non-interactively evaluated on a single input only.

Another setting where garbled circuits have been used is verifiable computation [GGP10] where a client is allowed an expensive preprocessing phase to be executed with a server. At a later point in time, the client is resource constrained, but gets some input which she needs to perform some computation on. She then prepares some information based on this input which she sends to the server. The server is then able to execute a function on the input and return the output, along with some proof of correct work, which the client can efficiently verify. That is, unlike one-time programs the same function is evaluated on several inputs and the evaluator does not learn the output. The multi-client setting of this is considered in [CKKC13].

Garbled circuits have also been used to construct efficient Private Function Evaluation (PFE), which is like two-party SFE, but where only one party knows the function to be evaluated [KM11, MS13].

It is also possible to use garbled circuits as a component in secure multi-party protocols computing a specific functionality, rather than supporting computation of arbitrary functionalities. An example of this is the computation of secure set intersection, such as in [PSSZ15]. This might be desirable as these specific protocols can be more efficient than their more generic counterparts.

1.1 Contributions

In the following we briefly outline the results and constructions which this thesis is comprised of.

Privacy-Free Garbling - Chapter 3

Different applications of garbled circuits often use different properties of the garbling scheme: In some applications we need garbled circuits to protect the *privacy* of encrypted inputs and intermediate values, in others we might also need the evaluator to remain *oblivious* of the output of the garbled circuits, while in yet others we might need *authenticity* of the output, by ensuring that even a malicious evaluator cannot tamper with the robustness of the garbled circuit. In their foundational work Bellare *et al.* [BHR12b] formally defined the different security properties that different applications require from garbled circuits, showed separations between them, and showed a "general" garbling scheme satisfying all of the above properties. This raises a natural question:

Can we construct garbling schemes tailored to specific applications,
which are more efficient than general schemes?

In Chapter 3 we answer this question in the affirmative. We will present garbling schemes which only satisfy *authenticity*, but not *privacy* or *obliviousness* (using the terminology of Bellare *et al.*). In general the garbled circuit evaluator cannot learn the values associated with the internal wires during the evaluation of the garbled circuit. This implies that the evaluation of each garbled gate must be *oblivious* (it must be the same for each input combination). We give up on this property and construct schemes where the evaluator learns the values associated whith each wire in the circuit, and explicitly uses this knowledge to perform *non-oblivious* garbled gate evaluation. This allows us to significantly reduce the size of a garbled circuit and the computational overhead for the circuit constructor. We show that this does not have any impact on *authenticity*, i.e., the only thing that a malicious evaluator can do with a garbled input and a garbled circuit is to use them in the intended way, that is to evaluate the garbled circuit on the garbled input and produce the (correct) garbled output.

Technical Overview

In a nutshell, our garbling schemes work as follows: Consider a NAND gate, with associated input keys K_l^0, K_l^1 and K_r^0, K_r^1 for the left and right wire respectively, and output keys K_o^0, K_o^1. The circuit constructor needs to provide the evaluator with a cryptographic gadget that, on input K_l^a, K_r^b, outputs the corresponding output key $K_o^{a \bar{\wedge} b}$. Remembering that our goal is not privacy, but only authenticity, we see that the evaluator is allowed to learn a and b, but even a corrupted evaluator should not learn $K_o^{1-(a \bar{\wedge} b)}$. In particular, this means that the evaluator should learn K_o^0 if and only if (iff) he holds both K_l^1 and K_r^1. This can be ensured by encrypting K_o^0 under *both* K_l^1 and K_r^1.

On the other hand, it is enough that one of the inputs' semantic value is 0 for the output to be K_o^1, so it "should be enough" to hold K_l^0 or K_r^0 to learn K_o^1. In general garbled circuits, we do not want the evaluator to learn which of the three possible combinations of input keys he holds (nor the output of the gate) and therefore we encrypt K_o^1 under all the three possibilities in the same way as we encrypt the 0-key. But if the evaluator is allowed to know which bits his keys correspond to, we can simply encrypt K_o^1 separately under K_l^0 and K_r^0, thus saving one encryption.

Note that we can instead derive K_o^0 from the input keys as $K_o^0 = \mathsf{KDF}\left(K_l^1, K_r^1\right)$, where KDF is a Key Derivation Function (KDF) and therefore we can use the row reduction approach to remove one ciphertext from the garbled computation table. We now have two-choices:

- If we want to be compatible with the free-XOR technique the value K_o^1 is already determined by K_o^0 and the global difference Δ, and thus no more row-reduction is possible.
- Alternatively we can decide to give up on free-XOR and derive K_o^1 as $K_o^1 = \mathsf{KDF}\left(K_l^0\right)$, thus removing yet another ciphertext from the garbled computation table, that now contains only the ciphertext $P = K_o^1 \oplus \mathsf{KDF}\left(K_r^0\right)$.

Cheap XOR. If we choose the second path, we need an efficient way of garbling the XOR gates: we do so by defining the output keys K_o^0 and K_o^1 as $K_o^0 = K_l^0 \oplus K_r^0$ and $K_o^1 = K_l^0 \oplus K_r^1$ respectively. Of course, it might be that at evaluation time the evaluator holds K_l^1 instead of K_l^0, and thus we provide him with an "advice" to compute the correct output key in this case. It turns out that it suffices to reveal the value $P = K_l^0 \oplus K_r^0 \oplus K_l^1 \oplus K_r^1$. Due to the symmetry of the XOR gate, now the evaluator can always derive the correct output key. Note that now XOR gates do not require any cryptographic operation but only the communication of a κ-bit string, and therefore are "almost" for free.

The paranoid reader might now worry on whether revealing the XOR of all input keys affects the security of our scheme, and the impatient reader might not want to wait for the formal proof, which appears in Chapter 3: Intuitively revealing P does not represent a problem because, if it did, then the free-XOR technique would be insecure as well: In (general) free-XOR the value P is always 0, as $K_l^0 \oplus K_l^1 = K_r^0 \oplus K_r^1$, and therefore known to the adversary already.

Privacy-Free fleXOR. Finally, we combine our technique with the recent fleXOR garbling scheme [KMR14]. A central concept in fleXOR is to look, for each wire, at the XOR between the two keys associated with that wire (which we call the *offset* of that wire). While in free-XOR schemes the offset is a constant for the whole circuit (therefore fixing half of the keys in the circuit), in fleXOR wires are ordered in a way to maximize the number of offsets which are the same, while at the same time leaving the circuit garbler the ability to choose freely the output keys for the non-XOR gates.

The fleXOR wire ordering induces a partitioning of the wires for each XOR gate. In particular, each XOR gates is assigned a parameter d which denotes how many input wires have offset *different* from the output wire, we call it a d-XOR gate. Then a 0-XOR gate can be garbled exactly like in free-XOR, while for d-XORs (with $2 \geq d > 0$) the garbler sends d ciphertexts to the evaluator, which are used to "adjust" the offsets of those input wires. In the privacy-free case, exploiting non-oblivious gate evaluation, we can simply reveal the XOR of the offsets instead, like in our cheap XOR scheme. So, while the original fleXOR requires the garbler and the evaluator to perform two and one calls to the KDF respectively, we do not require any cryptographic operations for fleXOR gates.

Garbling XORs. To conclude this technical introduction, we would like to present the reader with a recap of the different ways in which XOR gates are garbled in this part of the thesis. Like before, let K_l, K_l^1, K_r^0, K_r^1, and K_o^0, K_o^1 be the keys for the left, right and output wire, and let Δ_l, Δ_r and Δ_o be their differences, the offsets associated to the wires. Now, the "baseline" garbling of a XOR gate is done as follows: the garbler sets $K_o^0 = K_l^0 \oplus K_r^0$, then computes and sends to the evaluator the following values:

$$T_l = \Delta_l \oplus \Delta_o \text{ and } T_r = \Delta_r \oplus \Delta_o \ .$$

Given input keys K_l^a, K_r^b, the evaluator retrieves

$$K_o^{a \oplus b} = K_l^a \oplus K_r^b \oplus (a \cdot T_l) \oplus (b \cdot T_r) \ .$$

The baseline garbling transmits 2 ciphertexts, but in most cases we can do better.

Cheap XOR: In this case the garbler can freely choose Δ_o, which is set to be equal to Δ_l (so that $K_o^1 = K_l^1 \oplus K_r^0$) and therefore we do not need to communicate T_l, saving one ciphertexts w.r.t. the baseline.

Free-XOR: Here it holds that $\Delta_l = \Delta_r = \Delta_o$, therefore both $T_l = T_r = 0$ and no ciphertexts need to be transfered.

FleXOR: A XOR gate is garbled like in the baseline garbling when $d = 2$, like in cheap XOR scheme when $d = 1$ and like free-XOR when $d = 0$.

This subsection has been an introduction to results based on the paper [FNO15]. We will go into further details in Chapter 3.

Cut-and-Choose Using the Graphics Processing Unit (GPU) - Chapter 4

Doing cut-and-choose of garbled circuits to construct maliciously secure protocols for SFE requires solutions to the issues of "input consistency", "selective failure", and a way to decide on the "correct output". In the recent years many different solutions to these issues have been considered in the literature, such as in [LP15, LP12, SS11, SS13, Lin13, HKE13, Bra13, MR13, AMPR14]. In this part of the thesis we present a new way of ensuring input consistency of the constructor's input and a new way of implementing the "forge-and-lose" approach, both in manners which are particularly suitable for computation in a Same Instruction, Multiple Data (SIMD) environment. We make other design choices which are fitting for this setting and finally present implementations of two protocols based on these chooses, using an ordinary consumer GPU.

Technical Overview

Computationally our protocols rely solely on symmetric primitives, except for a few OTs (linear in computational security parameter). Furthermore, our protocols are of constant round complexity and, assuming access to enough cores, computationally bounded only by the number of layers in the circuit to be computed (and the time it takes to compute one block of a hash function), assuming a reasonable circuit and choice of parameters. Using a NVIDIA GPU as our SIMD device, we make several experiments and we show that our approaches to two party SFE are among the fastest documented assuming a "practical", yet malicious, setting.

Input consistency. To ensure consistency of A's input we augment the functionality to be computed such that A's input, apart from being used in the computation of the "real" function, is also fed into a universal hash function decided by B. The output of both the real functionality and the hash digest of A's input is then learned by B. We can now safely let B abort if any of these digests diverge, without leaking anything to A about his input. Intuitively, this approach is secure if the output of the augmented functionality does not leak information about A's input and A cannot find two inputs whose digests collide. We achieve the first property by having A give some auxiliary random input, which one-time pads the output of the hash function. The second property will follow directly from the nature of a universal hash function if A is committed to her input choices, before learning the specific hash function used.

Our initial way to construct such an augmentation was as follows [FN13]: Assume that the functionality we wish to compute is defined as $f(x) = y$ where $y = y_\mathsf{B}$ and $x = x_\mathsf{A} \| x_\mathsf{B}$ when we let $\|$ denote concatenation. We then define a new function f' as $f'(x') = y'$ where $x'_\mathsf{A} = x_\mathsf{A} \| \alpha$, $x'_\mathsf{B} = x_\mathsf{B} \| \beta$, $x' = x'_\mathsf{A} \| x'_\mathsf{B}$ and $y' = y \| \tau$. The values α and β are uniformly random bit strings, picked by A and B respectively. Specifically α consists of s bits and β of $n_\mathsf{A} + s - 1$ bits. Letting s be the *statistical security parameter*.[4] The extra output τ is the hash digest of the universal hash function and consists of s bits. The computation of τ is done as follows: First we define a matrix $M^{\mathsf{In}} \in \{0,1\}^{s \times n_\mathsf{A}}$ where the i'th row is the first n_A bits of the string β, bitwise shifted i bits to the left. Specifically the j'th bit of the i'th row is the $i+j-1$'th bit of the binary vector β. Using brackets to denote entries in matrices and vectors this means that $M^{\mathsf{In}}[i,j] = \beta[i+j-1]$. The computation of τ is then defined as $\tau = (M^{\mathsf{In}} \cdot x_\mathsf{A}) \oplus \alpha$, assuming all binary vectors are in column form. Now see that f' partially computes the same function as f, but requires s extra random bits of input from A and $n_\mathsf{A} + s - 1$ extra random bits from B. The s extra output bits will work as digest bits and can be used to check that A is consistent with her inputs to the circuits, by verifying that they are the same in all the garbled circuits which B evaluates.

Later, [SS13] observed that it is in fact not needed to have B specify the specific hash function as part of his input. Instead he can send A the vector β defining the matrix M^{In} and have her garble the circuit to compute the hash function. Evaluation of the garbled universal hash function can now be done using only XOR operations, which we can do for "free" with the free-XOR approach. Thus the binary string $\beta \in \{0,1\}^{n_\mathsf{A}+s-1}$ defines a family of universal hash functions mapping n_A bits to s bits. In turn, a sampling of such a function is simply a random choice of the string β. We will call a specific function from this family $\mathbb{H}^{\mathsf{In}}$ and call the actual circuit augmentation for $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ where a will be A's auxiliary input sampled from $\{0,1\}^s$.

A remaining problem with the construction is that A can now try to find a collision for the hash function before she gives her input keys to B. This is a significant problem as collisions for universal hash functions can be easy to find. However, if we let A commit to her input and *then* let B reveal his choice of hash function to her, then she will only have exponentially small probability in the size of the digest of finding a collision.

It should be noted that in [SS13] another universal hash function was used, which required $2s + \log_2(s)$ auxiliary random bits of input from A, unlike only s in our first approach. For our second protocol [FJN14] we use the observation made in [SS13] and prove that it is sufficient to compute the function $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$.

Selective failure. In [LP15] the authors show how to prevent selective failure attacks on the evaluators input in the cut-and-choose setting, using a circuit extension which increases

[4]We note that when we talk about statistical security we mean that even an unbounded adversary cannot break the security with more than exponentially small probability.

the amount of input bits of the evaluator from n_B to $\max(4 \cdot n_\mathsf{B}, 8 \cdot s)$. This is the solution we are using in both our protocols. More specifically, what happens is that the evaluator chooses a random binary matrix $M^\mathsf{Sec} \in \{0,1\}^{n_\mathsf{B} \times \max(4 \cdot n_\mathsf{B}, 8 \cdot s)}$ and a random binary vector $\bar{x_\mathsf{B}} \in \{0,1\}^{\max(4 \cdot n_\mathsf{B}, 8 \cdot n_\mathsf{B})}$ but under the constraint that $M^\mathsf{Sec} \cdot \bar{x_\mathsf{B}} = x_\mathsf{B}$ where x_B is the "true" input of the evaluator to the functionality f. Now, instead of computing $f(x_\mathsf{A} \| x_\mathsf{B}) = y$ we compute $\bar{f}(x_\mathsf{A} \| \bar{x_\mathsf{B}}) = f(x_\mathsf{A} \| M^\mathsf{Sec} \cdot \bar{x_\mathsf{B}}) = y$. Thus this new functionality computes exactly the same output as the original. Still, the idea of this approach is that if a selective failure attack is done, the augmented function will not leak any useful information as the entire vector $\bar{x_\mathsf{B}}$ is random, so learning a single bit of information of this vector will only give the adversary an exponentially small (in s) amount of information of the constructor's true input. This follows from the fact that the other bits of $\bar{x_\mathsf{B}}$ will be used to hide each of the actual bits of x_B. The details and a full proof of security of this approach can be found in [LP15].[5]

Other solutions include *committing OT*, where B gets, besides one output message, a commitment to *both* the messages A gave as input to OT. These commitments can the be opened for the check circuits to verify that A has used the "correct" messages as input to the OTs. This approach is used in [MF06, KS06, SS11].

Output Decision

Our first protocol (from [FN13]) does not support forge-and-lose (the concept had not been introduced at the time of writing), and thus this protocol relies on a majority of the garbled circuits remaining for evaluation being correct. B basically evaluates and decodes all garbled circuits remaining after cut-and-choose and defines his output to be the output decoded in most of these garbled circuits.

Our second protocol uses the forge-and-lose approach from [Bra13, Lin13] to decide the correct output. Namely, if the output of different garbled circuits diverge we make it possible for B to recover A's plain input x_A and in turn let B evaluate the functionality in plain. To do this [Lin13] relies on a secondary secure computation "inside" the protocol in order for B to learn x_A. While being independent of the size of the original circuit, the secondary secure computation still introduces a significant overhead. Furthermore, [Lin13] requires a number of modular exponentiations linear in the input size and reliance on the Decisional Diffie-Hellman (DDH) assumption in order to ensure A inputs the correct x_A to the secondary computation. In [Bra13] B gets to recover A's input by having her commit to her input keys and the keys on the circuit output wires, using trapdoor commitments where the trapdoor can be efficiently computed if two different circuit output keys are achieved for the same output wire, but in different garbled circuits.

Our main theoretical contribution in this part of the thesis arrives in the way we allow B to find the correct output based on the evaluation of the garbled circuits. We do so without using an extra secure computation as in [Lin13] or computationally heavy trapdoor commitments as in [Bra13]. Furthermore, we do not require any specific computational assumption.

Let ℓ be the replication factor, i.e. the amount of circuits A constructs. We assume without loss of generality (wlog) that ℓ is even. Now, let $\ell/2$ be the amount of check circuits. Also, let $K^0_{i,j}$ and $K^1_{i,j}$ be the 0-, respectively 1-keys for the i'th output wire in the j'th garbled circuit.

[5] It should be mentioned that shelat and Shen [SS13] present an improvement on this approach, where the increase in size of the evaluator's input is smaller. However, their approach is a bit more complex (in particular taken into account that it needs to be embedded in a Boolean circuit) so for simplicity we choose to implement the approach of [LP15].

We use the free-XOR technique [KS08] and let A garble each circuit with a different Δ. That is, for circuit j and all wires g A chooses the wire keys such that $K^0_{g,j} \oplus K^1_{g,j} = \Delta_j$.

We select the replication factor ℓ such that, except with probability 2^{-s}, if A was not caught cheating during the cut-and-choose phase, *at least one* of the evaluation circuits that remain is correct. Our strategy is then to make sure that for each pair of garbled circuit evaluations whose semantic output differ, B will be able to efficiently compute the global differences for one of those circuits. Knowing the global difference used to garble a circuit allows B to learn the input x_{A} that A submitted for that circuit.

Using polynomials to compute global differences. We now explain how we let B compute the global differences for those circuits whose output diverge. As part of the garbling phase A associates a polynomial of degree at most $\ell/2$ with each output wire in the circuit. Denote these P_i for $i \in [m]$. Now for each garbled circuit and each output wire A associates a point on the corresponding polynomial, say j. We abuse notation and view the polynomial as a function i.e., we let value of point $j \in [\ell]$ on polynomial $i \in [m]$ be denoted by $P_i(j)$. Thus we have a polynomial associated with each output wire and for each of these polynomials we have a point associated with each garbled circuit. Next, for the 0-key on each output wire in each circuit, $K^0_{i,j}$ and point $P_i(j)$ we associate the *link* $L_{i,j}$. The link is a gadget that returns $K^0_{i,j}$ iff it is given $P_i(j)$ and returns $P_i(j)$ iff it is given $K^0_{i,j}$. Thus, it makes a one-to-one correspondence between a circuit output 0-key and the polynomial point associated with this specific key's wire. A sends all these links to B.

Next notice that after the cut-and-choose step B will learn all the 0-keys on the output wires for $\ell/2$ garbled circuits and in turn be able to learn $\ell/2$ points on all the polynomials (by using the output 0-keys, $K^0_{i,j}$ along with the corresponding links $L_{i,j}$). Thus by learning just one more 0-key for a given output wire, B will be able to learn one more point on one of the polynomials. This will give him a total of $\ell/2+1$ points on a polynomial of degree at most $\ell/2$ and he will thus be able to do polynomial interpolation and learn all the points on that polynomial. Knowing all the points on a given polynomial will make it possible for B to learn the 0-key on a given output wire in *all* the garbled circuits (by using the points, $P_i(j)$ along with the links $L_{i,j}$).

A bit more concretely, suppose that B ends up with two garbled circuits where there is an output wire that outputs the 0-key in one of the circuits and the 1-key in the other. Say, wlog that this is the case for output wire 1 in circuit 1 and circuit 2, such that B learns both $K^0_{1,1}$ and $K^1_{1,2} = K^0_{1,2} \oplus \Delta_2$. Using the link $L_{1,1}$ B can compute the point $P_1(1)$. This gives him a total of $\ell/2+1$ points on the polynomial P_1 (remember that he already knew $\ell/2$ points from the check part of the cut-and-choose phase). Using these points he can then do polynomial interpolation, which in turn will make it possible for him to easily compute the point $P_1(2)$. He can now use link $L_{1,2}$ to learn $K^1_{1,2}$. Since he already knew $K^0_{1,2}$ and we use free-XOR he can compute $\Delta_2 = K^1_{1,2} \oplus K^0_{1,2}$. Using Δ_2 he can completely degarble the second garbled circuit, and in turn, also learn A's plain input to this circuit.

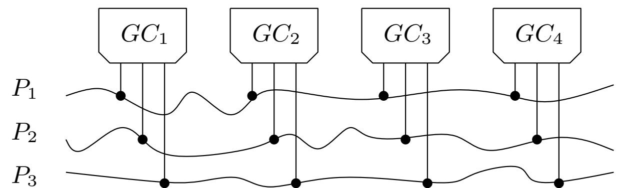


Figure 1.1: Four garbled circuits with three output wires. Each 0-key on these output wires is linked to a particular point on a polynomial of degree at most 2.

The approach of using the links to learn the global differences generalizes to each output wire

in two circuits where one has the 0-key and other the 1-key. Thus, assuming we have enough output wires with different values, it is possible to find the global differences for all garbled circuits, and in turn A's plain input to all garbled circuits.

We want to emphasize, however, that B does not use the global differences to completely open up any garbled circuit, as this would be too expensive (it would essentially count as an extra garbling towards the computational complexity). Instead B only opens up the input layer to learn the semantic value of the input keys of A to the circuit. He then evaluates the universal hash function on A's input in plain and compare it with the semantic meaning of the garbled evaluation of the universal hash function. Then B discards the circuits where the plain and semantic meaning of the output of the universal hash function is discrepant, which cannot lead to selective errors, as the correctness of the augmented part of the circuit is known by A already.

Now, for all the remaining circuits, the inputs of A are the same, except with a small probability 2^{-s} that a collision for the universal hash function occurred. This is so as the circuits computed the hash function correctly, and the hash function was chosen by B after A committed to her inputs. Hence B can safely abort if there are different inputs left, and otherwise just evaluate f in plain on his own input and the unique input of A.

Ensuring correct degree of polynomials. We made a few assumptions in the strategy described above. That is, it only works if the polynomials have degree at most $\ell/2$ *and* we have enough output wires that diverge in value to make sure B can learn the global differences for each of the evaluation circuits. Since A might be malicious, and thus can deviate from the prescribed protocol in an arbitrary manner, we cannot be sure that she constructs the polynomials of degree at most $\ell/2$. In particular she might construct these to be degree ℓ and thus make it impossible for B to use polynomial interpolation to find the global differences. We could use techniques like Kate *et al.* [KZG10] to ensure correct degree of the polynomials, but this would be inefficient and impose certain number theoretic assumptions. We instead introduce an additional cut-and-choose phase in which A prepares more polynomials than actually needed, and then B randomly chooses half of the polynomials as *check* polynomials, which A then reveals and B verifies to have degree at most $\ell/2$. We say that a polynomial of degree greater than $\ell/2$ is *inconsistent* and if it has degree at most $\ell/2$ we say it is *consistent*. Thus for B to accept the check polynomials they must all be consistent. If all the check polynomials are consistent then a large quantity of the remaining polynomials are guaranteed to be consistent. In fact, we select half of the polynomials as check polynomials, and therefore get that a majority of the remaining circuits are guaranteed to be correct except with negligible probability in the amount of challenge polynomials, for the same reasons as in the analysis of circuit cut-and-choose in [LP15].

Ensuring consistency of the polynomials with cut-and-choose is efficient and does not rely on any specific number theoretic assumptions. However, it leaves us with the issue that some of the polynomials that are not checked might still be inconsistent and thus not usable for interpolation in our context, along with the problem that we still need enough output wires which diverge in value to make sure B can learn the global differences for each of the evaluation circuits.

To cope with these problems we introduce another circuit augmentation, which is a universal hash function, taking the output of the original computation as input. We then let the the wire keys used in the links be the wire keys of the output of this new universal hash function instead of the wire keys of the actual output of the garbled circuit. Intuitively, if there is any divergence on one bit in the original output then the digests of the universal hash function will diverge in many bits. Thus the hash function ensures that we have enough output wires which will differ in their semantic values, since the output of the hash function is

uniformly distributed. Thus, if a garbled circuit is maliciously constructed then the hashed value of the malicious output will differ in many bits compared with the hashed value of the correct output. This ensures, except with negligible probability in the amount of output bits of the augmentation, that we will have enough output wires with different semantic values to learn the global differences for all the evaluation circuits. In Chapter 4 we prove that with only a majority of the polynomials guaranteed to be consistent, a hash function with at least $\lceil 4.82s + 4.82 \rceil$ output bits is sufficient in order to guarantee, except with probability 2^{-s}, that B will be able to learn all the global differences of the evaluation circuits if he ends up with two or more evaluation circuits where the semantic value on least one output wire diverge.

The augmentation on the output wires is done in almost the same way as the augmentation to ensure input consistency of A's input. That is, we exploit the free-XOR technique and let B send the specification of the hash function to A in clear. Specifically we have B choose two random binary strings $\beta_a \in_R \{0,1\}^{\lceil m+4.82s+3.82 \rceil}$ and $\beta_b \in_R \{0,1\}^{\lceil 4.82s+4.82 \rceil}$. Using the first string we define a $\lceil 4.82s + 4.82 \rceil \times m$ binary matrix M^{Out} such that the j'th bit of the i'th row is the $i+j-1$'th bit of β_a. That is, $M^{\mathsf{Out}}[i,j] = \beta_a[i+j-1]$. The augmentation should then compute $\left(M^{\mathsf{Out}} \cdot y\right) \oplus \beta_b$, assuming y is a binary vector in column form. Thus the binary strings $\beta_a \in \{0,1\}^{\lceil m+4.82s+3.82 \rceil}$ and $\beta_b \in \{0,1\}^{\lceil 4.82s+4.82 \rceil}$ define a family of universal hash functions mapping m bits to $\lceil 4.82s + 4.82 \rceil$ bits. In turn a sampling of such a function is simply a random choice of the strings β_a and β_b. We will use $\mathbb{H}^{\mathsf{Out}}$ to denote a specific function from this family.

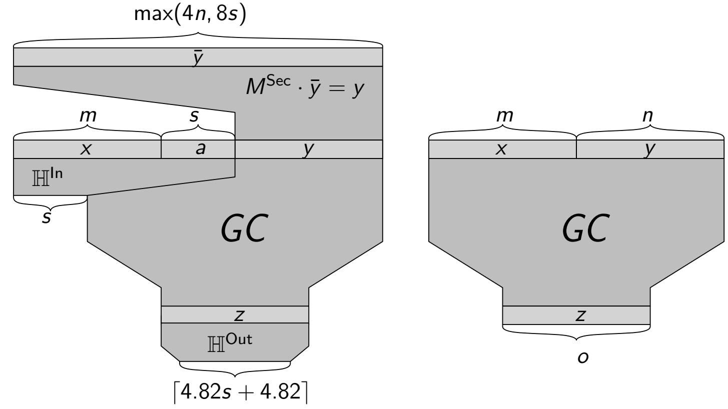


Figure 1.2: Illustration comparing the original circuit (right) and the augmented circuit (left). Inputs and outputs are shown in light grey, whereas actual computation is shown with a darker grey.

An illustration of the final circuit with all augmentations is given in Fig. 1.2.

OTs. In general, OT is an expensive primitive, and if the evaluator has a large input to the circuit this can contribute significantly to the execution time of the whole protocol. However, the amount of "actual" OTs we need to complete can be significantly reduced by using an OT extension: Beaver showed in [Bea96] that given a number of OTs it is possible to "extend" these to give a polynomial number of random OTs which can easily be changed to specific OTs. Thus making it possible to do a few OTs once, and extend these almost indefinitely. The idea of an OT extension has been optimized even further in [IKNP03] and [NNOB12] and lately even further in [ALSZ13] for semi-honest adversaries and [ALSZ15, KOS15] for malicious adversaries. Our protocol uses a slightly modified version of the OT extension presented in [NNOB12] as the recent, and more efficient, protocols of [ALSZ15, KOS15] were published after our work.

This subsection has been an introduction to results based on the papers [FN13] and [FJN14]. We will go into details in Chapter 4.

Other Results

Besides the results outlined above, the PhD programme involved the creation of several other published (and to be published) results, which unfortunately did not make it into this thesis because of space constraints. We briefly outline these results in the sequel.

MiniLEGO

We continue work on the idea of the LEGO construction [NO09] where cut-and-choose is done of individual gates, instead of circuits. A major bottleneck of [NO09] is the need of Pedersen commitments (which use computationally expensive exponentiations) in order to “solder” individual garbled gates together. Pedersen commitments were used because they have an additive homomorphic property which is essential for the soldering to work. In MiniLEGO we manage to adapt the LEGO paradigm to work using XOR homomorphic commitments. As an added bonus to this, we also manage to make the LEGO construction support free-XOR, which was not the case in [NO09]. As part of the result we also introduce a concrete way of constructing XOR homomorphic commitments which does not rely on expensive computation of exponentiations (or other computationally expensive operations.[6]) This construction relies on linear error correcting codes with certain features and black box access to OT. This outline is based on our results in the paper [FJN+13].

TinyLEGO

Even though MiniLEGO improved greatly on the original LEGO construction, trying to practically instantiate it showed that it will (except for extremely large circuits) not be able to compare favorably in practical performance with cut-and-choose based protocols. A major reason for this is the constraints on the error correcting codes needed by MiniLEGO are currently only known achievable using codes with very large constants. For example, this means that the amortized communication complexity of a single AND gate in a circuit with around 10,000 gates has one to two orders of magnitude greater communication complexity than in most cut-and-choose based protocols. Based on this observation we adapt and optimize the basic commitment scheme of [CDD+15] to an extremely efficient XOR homomorphic commitment scheme. We then introduce several small optimizations to the MiniLEGO protocol and end with an instantiation that is several orders of magnitude more efficient than MiniLEGO (in terms of communication complexity). This outline is based on our results in the papers [FJNT15a] and [FJNT15b].

A Unified Approach to MPC

We previously mentioned that there are several other ways to realize secure computation besides the use of garbled circuits, such as based on MACs of secret shared values. For these types of protocols we try to unify and optimize the extensive and expensive preprocessing phase. We base our approach on an arbitrary OT extension. However, for particular OT extensions we manage to optimize even further. Specifically we manage to base much of our preprocessing on correlated and semi-honestly secure OTs. Here correlated means that there is a linear relation between the two messages in a given OT which is the same for each of the OTs in an execution of an extension. This fits well with most of the efficient OT extensions we currently know. The work includes pointing out (and fixing) a problem with the multi-party version of TinyOT [LOS14]

[6] In particular we do not rely on number theoretic assumptions or public-key cryptographic primitives dependent on the amount of commitments to construct.

and the first concrete preprocessing phase for MiniMAC. Furthermore, based on estimates, we manage to optimize the preprocessing phase of SPDZ around two orders of magnitude. This outline is based on our results in the paper [FKOS15].

1.2 Outline

The rest of this thesis is organized as follows: In the next chapter we introduce the formal notations we will use. We also introduce some specific constructions and paradigms we need in order to formalize our results. This includes a short description of the models we will use, that is the Random Oracle Model (ROM) and the Universal Composition (UC) proof paradigm. It also includes a formalization, and concrete instantiation of commitments, a garbling scheme, and a formal description of the OT functionality.

In Chapter 3 we formalize our notion of privacy-free garbling, using the formal frameworks introduced in Chapter 2. We describe concrete and efficient instantiations, and prove their security in the ROM. Furthermore, we introduce an efficient generalization to arbitrary fan-in gates.

We continue in Chapter 4 with a description of two SIMD parallelizable protocols for two-party SFE based cut-and-choose of garbled circuits. We prove the most efficient of these UC-secure and discuss how to efficiently instantiate it in the ROM. We then give an introduction to GPU programming based on CUDA and describe how we made a highly efficient proof of concept implementation of these two protocols in CUDA. We end the chapter with some benchmarks of our implementations, comparing these with the most efficient alternatives.

Chapter 2

Preliminaries

2.1 Notation

Let $\mathbb{N} = \{1, 2, \dots\}$ be the natural numbers, excluding 0. We write $[x; y]$ (with $x < y \in \mathbb{N}$) for $\{x, x+1, \dots, y\}$ and $[x]$ for $[1; x]$. If x is a vector, we will sometimes use subscript to index an element of it. That is, x_i returns the i'th element of x. Other times we will use brackets to return an element. That is, if x is a vector then $x[i]$ returns the i't element. We use the same paradigm for matrices. Thus if M is a matrix, then $M[i, j]$ returns the j'th element of the i'th row. We use $|\cdot|$ as a shorthand for the cardinality of a set, the length of a list, or the amount of bits in a string. If S is a set or a randomized algorithm we use $x \in_R S$ or $x \leftarrow^* S$ ($S \rightarrow^* x$) to denote that x is a uniformly random sampled element from S. We will use $\leftarrow$ ($\rightarrow$) to denote deterministically parsing, or computing, an element on the right-hand (left-hand) side into an element, a list, or set of elements on the left-hand (right-hand) side. We let $\text{poly}(\cdot)$ denote any polynomial of the argument. We will use $x := x \otimes y$, where $\otimes$ is an operator and y a variable, to denote an update to the variable x. We use $\perp$ to denote the empty set. We let $\text{lsb}(x)$ denote the LSB of a bit string, or bit vector $x \in \{0,1\}^*$. To be specific we view bitstrings as Most Significant Bit (MSB) first. Thus if $x = 01$ then $\text{lsb}(x) = 1$. Logarithms will always be base 2 unless otherwise stated. We will sometimes use $(x_i)_{i \in [y]}$ as a shorthand for the list $(x_1, x_2, \dots, x_y)$. We let $\|$ denote concatenation. Sometimes we abuse notation and let $1^x = 1\|1\cdots\|1$, that is the concatenation of 1, x times. However, we only do this when the base is 1 or 0.

Regarding variable names we let $s \in \mathbb{N}$ denote the statistical security parameter and $\kappa \in \mathbb{N}$ be the computational security parameter. We will assume that $s \leq \kappa$. We call a function $\text{negl} : \mathbb{N} \rightarrow \mathbb{R}^+$ negligible if, for a big enough κ, it holds that $\text{negl}(\kappa) < 1/\text{poly}(\kappa)$. In general we use $\text{negl}(\cdot)$ to denote any negligible function. When we just say that a function f is negligible, we mean in the security parameter κ.

We let $L \subset \{0,1\}^*$ be an arbitrary language in NP and M_L be the language verification function, i.e., for all $y \in L$ there exists a string $x \in \{0,1\}^{\text{poly}(|y|)}$ s.t. $M_L(x, y) = \texttt{accept}$ and for all $y \notin L$ and $x \in \{0,1\}^*$ we have $M_L(x, y) = \texttt{reject}$.

We will sometimes abuse notation and write $\Pr[\mathrm{A}]$ when we really mean $\Pr[\mathrm{A} = \top]$ for a possibly randomized algorithm A.

2.2 Universal Composability

When applicable, we will prove our protocols secure in the real/ideal setting using the UC framework [Can00, Can01]. We assume the reader is familiar with this framework, but briefly outline its overall structure.

Security of cryptographic protocols is proven by arguing indistinguishability between two systems of interactive Turing machines. One of these systems is called the *real world* and the other the *ideal world*. The real world is reflecting the true protocol, whereas the ideal world is reflecting a simulation of what we ideally wish of the true protocol. To argue indistinguishability we use a probabilistic polytime interactive Turing machine called the *environment*, which is denoted by Z. The environment gets to act with either the real world or the ideal world and outputs a single bit indicating which of the systems it guesses it was acting with. The idea is now that if no Z can distinguish between the real and ideal world with more than a negligible advantage in the security parameter κ, then the protocol of the real world securely realizes the functionality described by the ideal world.

In a bit more detail, we let the protocol in question be denoted Π and assume that it involves n parties. The real world then consists of n interactive Turing machines, $P_1, P_2, \ldots, P_n$ executing the protocol for each of the parties. We furthermore assume that there is another interactive Turing machine, $\mathcal{A}$, representing an adversary attacking the protocol. The execution is done by sequential activations of the machines. It can be seen as a token getting passed between the machines, where a machine can only run if it is in possession of the token. The machine that will get activated next is the machine which has its input tape written to. The adversary controls the communication between the parties and can read everything they output. This means that the adversary can always make the protocol abort by simply closing down the communication between parties or replacing real messages with garbage. The adversary may choose to corrupt some parties, when this happens it basically takes over the execution of the Turing machine of that party. It thus also gets to see everything written on the tapes of these machines.

The ideal world involves an ideal functionality $\mathcal{F}$, modeled as an interactive Turing machine, capturing the desired functionality of the protocol Π. It also involves n *dummy parties* $\tilde{P}_1, \tilde{P}_2, \ldots, \tilde{P}_n$ and a *simulator* S. The dummy parties simply pass on messages written on their input tapes to the ideal functionality $\mathcal{F}$. Whatever they get back they pass on to the environment Z. The simulator S will try to simulate $\mathcal{A}$ by using $\mathcal{F}$.

The environment gets to choose the input of the parties and gets to see their output. In the real world it also gets to see everything that is sent between the parties. Furthermore, it gets to fully control the adversary, and thus decide on its behavior. Thus, in the ideal world S must create a transcript that “looks like” the transcript in the real world. It must do so “online”, meaning that it must do so for each step during the protocol, and not just wait until the end of the execution and generate it in one shot. This is captured by the way we start activation; the environment is activated first and hands on the token by writing on the input tape of one of the interactive Turing machines.

To argue about indistinguishability we define the random variable $\mathsf{REAL}_{\Pi,\mathcal{A},\mathsf{Z}}(\kappa, z)$ representing the output of the environment Z when interacting with the adversary $\mathcal{A}$ and the parties running the protocol Π with security parameter $\kappa \in \mathbb{N}$, when Z is given $z \in \{0,1\}^*$, over all the randomness used by the interactive Turing machine. We then let $\mathsf{REAL}_{\Pi,\mathcal{A},\mathsf{Z}}$ denote the ensemble $\{\mathsf{REAL}_{\Pi,\mathcal{A},\mathsf{Z}}(\kappa, z)\}_{k\in\mathbb{N},z\in\{0,1\}^*}$. In a similar manner we let the random variable $\mathsf{IDEAL}_{\mathcal{F},\mathsf{S},\mathsf{Z}}(\kappa, z)$ represent the output of the environment Z when interacting with the simulator S and the ideal functionality $\mathcal{F}$ with security parameter $\kappa \in \mathbb{N}$, when Z is given $z \in \{0,1\}^*$, over all the randomness used by the interactive Turing machine. We then let $\mathsf{IDEAL}_{\mathcal{F},\mathsf{S},\mathsf{Z}}$ denote the ensemble

$\{\mathsf{IDEAL}_{\mathcal{F},\mathsf{S},\mathsf{Z}}(\kappa, z)\}_{k\in\mathbb{N}, z\in\{0,1\}^*}$.

We say that Π *securely realizes* the ideal functionality $\mathcal{F}$ if for any Probabilistic Polynomial Time (PPT) adversary $\mathcal{A}$ there exists a simulator S such that no PPT environment Z, on any input $\kappa \in \mathbb{N}$, $z \in \{0,1\}^*$ can tell with non-negligible advantage in κ if it is playing with $\mathcal{A}$, $P_1, P_2, \dots, P_n$ or $\mathsf{S}, \mathcal{F}, \tilde{P}_1, \tilde{P}_2, \dots, \tilde{P}_n$. More formally:

$$|\Pr[\mathsf{REAL}_{\Pi,\mathcal{A},\mathsf{Z}}(\kappa, z) = 1] - \Pr[\mathsf{IDEAL}_{\mathcal{F},\mathsf{S},\mathsf{Z}}(\kappa, z) = 1]| \leq \mathrm{negl}(\kappa) \ .$$

We denote this by $\mathsf{REAL}_{\Pi,\mathcal{A},\mathsf{Z}} \equiv \mathsf{IDEAL}_{\mathcal{F},\mathsf{S},\mathsf{Z}}$.

The crux of UC, as the name suggests, is composability, meaning that if we have proved a protocol to securely realize an ideal functionality $\mathcal{F}_1$ then we can use this functionality in another protocol, trying to realize some other functionality $\mathcal{F}_2$. We call this the *hybrid model*. For this example we can then prove a protocol for $\mathcal{F}_2$ secure in the $\mathcal{F}_1$-hybrid model. A bit more specifically the $\mathcal{F}_1$-hybrid model is similar to the real world, but with the addition of an unbounded number of copies of an interactive Turing machine $\mathcal{F}_1$. Each copy is identified with a *Session ID* (SID). Using this idea it is then possible to combine simple functionalities to construct advanced protocols. From this we have the UC composition theorem:

Theorem 1 (Universal Composition [Can01])**.** *Let* P *be an* n *party protocol securely realizing an ideal functionality* $\mathcal{F}$*. Next, let* Π *be an* n *party protocol in the* $\mathcal{F}$*-hybrid model realizing the ideal functionality* $\mathcal{G}$ *then for a real world adversary* $\mathcal{A}$ *there exists a hybrid model adversary* $\mathcal{H}$ *such that for any PPT environment* Z *we have* $\mathsf{REAL}_{\Pi^{\mathrm{P}},\mathcal{A},\mathsf{Z}} \equiv \mathsf{HYB}^{\mathcal{F}}_{\Pi,\mathcal{H},\mathsf{Z}}$ *where* Π^{P} *denotes the protocol where calls to* $\mathcal{F}$ *are replaced with called to the protocol* P *and* $\mathsf{HYB}^{\mathcal{F}}$ *denotes the hybrid world with functionality* $\mathcal{F}$*.*

The above theorem implies that if we have already proved that P securely realizes $\mathcal{F}$ then it is enough to prove that Π, using the hybrid functionality $\mathcal{F}$, securely realizes $\mathcal{G}$. This can of course be repeated, hence the composition.

In this thesis we always assume the parties have access to authenticated channels, meaning that the adversary cannot change the messages coming out of the honest parties. Furthermore, since we will only be considering maliciously secure two party functionalities we will interchangeably consider an explicit adversary and two parties, or just two parties where one of them is maliciously corrupted by the adversary, and thus under his complete control. This will yield less overhead in some of our presentations.

Adversarial Models

The power of the adversary attacking a cryptographic protocol can be specified using several different vectors of measurement. We briefly outline the major vectors which are relevant to the work in this thesis:

Semi-honest/Malicious In the *semi-honest* (also known as passive and honest-but-curious) setting the adversary may only try to break the security of the protocol by looking at the messages sent. That is, the corrupted parties *must* follow the protocol description, even when dishonest. In the *malicious* (also known as active) setting the adversary may have the corrupted parties act in any way he sees fit. In particular they might not follow the protocol description.

Static/Adaptive A static adversary decides which parties (if any) to corrupt before the start of the protocol, whereas an adaptive adversary can decide which parties to corrupt after part of the protocol has been executed (honestly).

Threshold If the adversary corrupts strictly less than half of the parties in the protocol we say that there is an *honest majority*, whereas if he corrupts half or more of the parties there is *dishonest majority*.

In this thesis we wish to construct protocols that are secure against a static and malicious adversary in a two-party protocol (meaning we have a dishonest majority).

Hash functions and Random Oracles

Most of the protocols we introduce in this thesis are proved secure in the ROM [BR93], where hash functions are replaced with random oracles. We define a random oracle as a function $\mathsf{R} : \{0,1\}^* \rightarrow \{0,1\}^c$ for some $c \in \mathbb{N}$. In particular we will have $c = \kappa$. The computation of R proceeds as follows: On a query $\mathsf{R}(x)$ with $x \in \{0,1\}^*$ if this is the first time R is queried on x then it picks a uniformly random value $y \in_R \{0,1\}^c$ which it returns. If it has been queried on x before then it returns the value y it returned on this same query previously. In the ROM we prove security assuming that a hash function is replaced by a random oracle. This is a rather liberal assumption, since proofs exist that a protocol might be secure in the ROM but completely insecure when replacing the oracle with a practical hash function [CGH04]. These are however very sought out, so in general we do believe the assumption is sensible in practical settings.

We note that the random oracle can be modeled in different ways. In particular we can talk about a *programmable* or *non-programmable* random oracle. We say that the oracle is programmable if the simulator can pick the reposes returned by the oracle, under the constraint that they are still sampled from a random distribution.

2.3 Cryptographic Building Blocks

We will use many different cryptographic primitives in this thesis and we assume the reader is familiar with common cryptographic primitives such as block ciphers, hash functions, pseudo-random generators and so on. However, for some primitives we need formal specifications of ideal functionalities, thus for completeness we define these in this section. Furthermore, since garbling is such a major part of this thesis we will also formalize and go into details of this primitive in the next section.

Commitments

A commitment scheme is the cryptographic equivalent to a locked box. The *committer commits* to a message by locking it in an opaque box, which he then hands over to the *receiver*. At a later time he can then *open* the commitment by handing the receiver the key used to lock the box. Such a scheme has two crucial features: before receiving the key, the receiver cannot learn anything about the message. We say the scheme is *hiding*. After sending the box to the receiver the committer cannot change its content. We say the scheme is *binding*.

We need a commitment scheme as defined in Fig. 2.1. The methods **Commit** and **Open** define a regular commitment scheme while the methods **Verifiable Commit**, **Checking**, and **Open** define a verifiable commitment scheme. By a verifiable commitment scheme, we are simply talking about a regular commitment scheme, but where the opening is exactly the message

committed to. This makes it possible for the receiver to test whether or not a commitment is of a given message. Thus the scheme is only computationally hiding.[1]

In the ROM commitments can simply be implemented using a hash function. For regular commitments, the committer sends a hash of the message to be committed to, concatenated with a random string. Verifiable commitments can be done similarly, but without concatenating the message with a random string. The following scheme (in Fig. 2.1) and proof are taken almost verbatim from [DKL+13], except that it is augmented to support verifiable commitments.

Functionality $\mathcal{F}_{\text{COM}}$

Commit

Upon receiving the command $(\texttt{commit}, \mathsf{name}, \mathsf{sid}, \mathsf{ssid}, m)$ from the party name, store $(\mathsf{name}, \mathsf{sid}, \mathsf{ssid}, m)$ and output $(\texttt{commit}, \mathsf{name}, \mathsf{sid}, \mathsf{ssid})$ to both parties. Ignore any future $\texttt{commit}$ or $\texttt{vc}$ messages with same ssid from name.

Verifiable Commit

Upon receiving the command $(\texttt{vc}, \mathsf{name}, \mathsf{sid}, \mathsf{ssid}, m)$ from party name store $(\mathsf{name}, \mathsf{sid}, \mathsf{ssid}, m)$ and output $(\texttt{vc}, \mathsf{name}, \mathsf{sid}, \mathsf{ssid})$ to both parties. Ignore any future $\texttt{commit}$ or $\texttt{vc}$ messages with same ssid from name.

Checking

Upon receiving the command $(\mathsf{check}, \mathsf{name}, \mathsf{sid}, \mathsf{ssid}, m')$ from any party, if the call $(\texttt{vc}, \mathsf{name}, \mathsf{sid}, \mathsf{ssid}, m)$ has ever been made where $m = m'$ then output $(\mathsf{check}, \mathsf{name}, \mathsf{sid}, \mathsf{ssid}, \top)$ to the party making the call. Otherwise output $(\mathsf{check}, \mathsf{name}, \mathsf{sid}, \mathsf{ssid}, \bot)$ to the party making the call.

Open

Upon receiving $(\mathsf{open}, \mathsf{name}, \mathsf{sid}, \mathsf{ssid})$ from party name if a commitment $(\mathsf{name}, \mathsf{sid}, \mathsf{ssid}, \cdot)$ is stored then the functionality outputs $(\mathsf{open}, \mathsf{name}, \mathsf{sid}, \mathsf{ssid}, m)$ to both parties. If name is corrupted and the adversary gives the command $(\mathsf{NoOpen}, \mathsf{name}, \mathsf{sid}, \mathsf{ssid})$ then output $(\texttt{reject}, \mathsf{name}, \mathsf{sid}, \mathsf{ssid})$ to both parties.

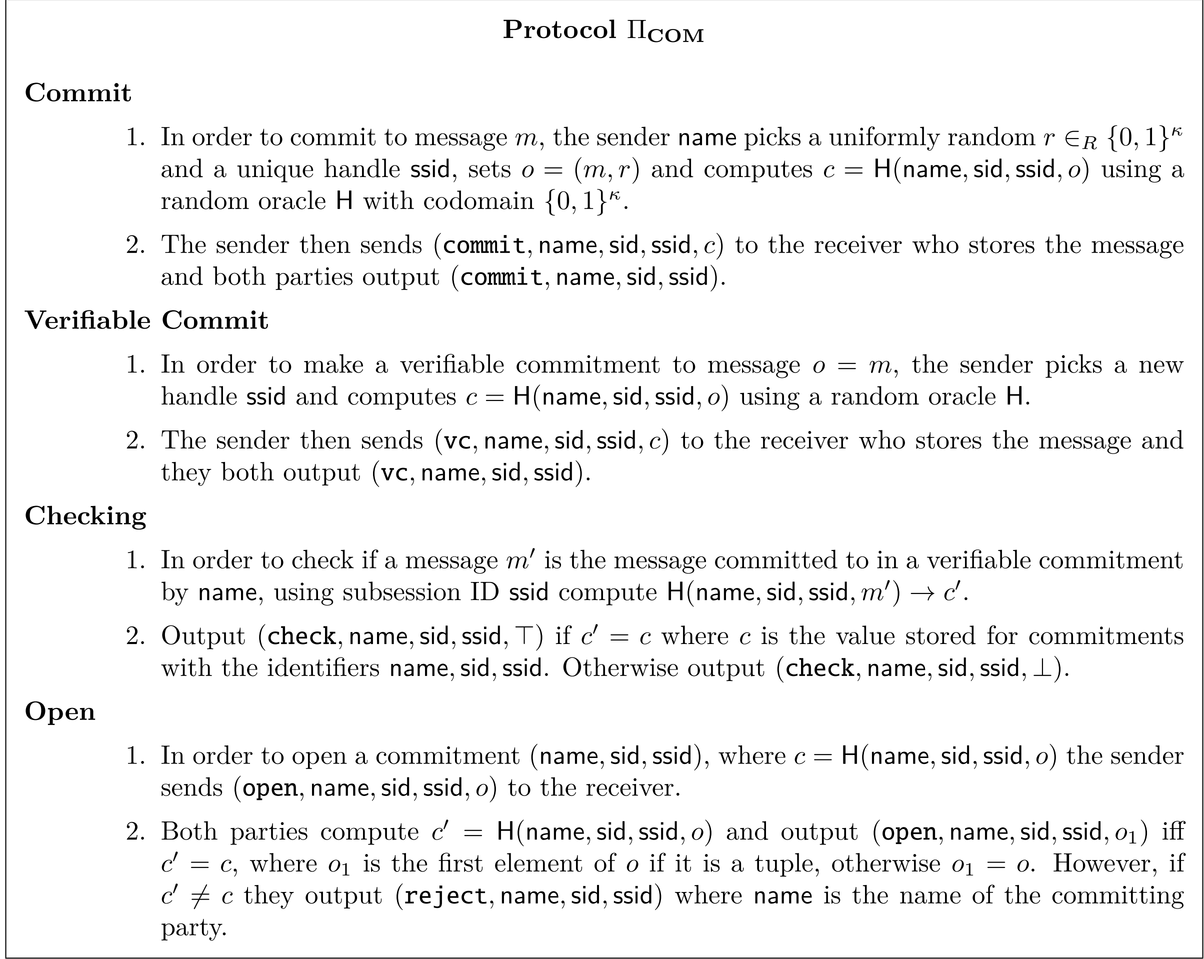
Figure 2.1: The ideal functionality $\mathcal{F}_{\text{COM}}$ for commitments and verifiable commitments.

Lemma 1. *In the programmable random oracle model the protocol Π_{COM} in Fig. 2.2 UC-securely realizes the ideal functionality $\mathcal{F}_{\text{COM}}$ from Fig. 2.1 against any PPT static and malicious adversary corrupting either A or B, assuming messages given as inputs to **Verifiable Commit** have min-entropy at least κ bits in the view of the receiving party.*

Proof. We start by sketching the simulator. We notice that it learns the input given to H when the adversary queries it as it simulates its behavior. Furthermore, since we only consider two parties we assume, wlog that A is the committing party and B is the receiving party.

First consider that the committing party, A, is honest. When she commits the simulator receives the value $(\texttt{commit}, \mathsf{A}, \mathsf{sid}, \mathsf{ssid})$ from $\mathcal{F}_{\text{COM}}$. Assuming the value ssid has not been used before with sid, the simulator then picks a uniformly random value c from the codomain of H and sends $(\texttt{commit}, \mathsf{A}, \mathsf{sid}, \mathsf{ssid}, c)$ to B. If A is corrupt, assuming the value ssid has not been used before with sid, the simulator will receive a value $(\texttt{commit}, \mathsf{A}, \mathsf{sid}, \mathsf{ssid}, c^*)$ from the adversary. If the adversary queried $\mathsf{H}(\mathsf{A}, \mathsf{sid}, \mathsf{ssid}, o)$ with output c^* then the simulator will set $o^* = o$, otherwise it will let o^* be a dummy input and set an internal flag $\mathsf{Abort}_{\mathsf{A},\mathsf{sid},\mathsf{ssid}}$ to true. It then sends $(\texttt{commit}, \mathsf{A}, \mathsf{sid}, \mathsf{ssid}, o^*)$ to $\mathcal{F}_{\text{COM}}$.

[1] In fact only so if the entropy of the message is $\Omega(\kappa)$.



Protocol Π_{COM}

Commit

1. In order to commit to message m, the sender name picks a uniformly random $r \in_R \{0,1\}^\kappa$ and a unique handle ssid, sets $o = (m, r)$ and computes $c = \mathsf{H}(\mathsf{name}, \mathsf{sid}, \mathsf{ssid}, o)$ using a random oracle H with codomain $\{0,1\}^\kappa$.
2. The sender then sends $(\mathtt{commit}, \mathsf{name}, \mathsf{sid}, \mathsf{ssid}, c)$ to the receiver who stores the message and both parties output $(\mathtt{commit}, \mathsf{name}, \mathsf{sid}, \mathsf{ssid})$.

Verifiable Commit

1. In order to make a verifiable commitment to message $o = m$, the sender picks a new handle ssid and computes $c = \mathsf{H}(\mathsf{name}, \mathsf{sid}, \mathsf{ssid}, o)$ using a random oracle H.
2. The sender then sends $(\mathsf{vc}, \mathsf{name}, \mathsf{sid}, \mathsf{ssid}, c)$ to the receiver who stores the message and they both output $(\mathsf{vc}, \mathsf{name}, \mathsf{sid}, \mathsf{ssid})$.

Checking

1. In order to check if a message m' is the message committed to in a verifiable commitment by name, using subsession ID ssid compute $\mathsf{H}(\mathsf{name}, \mathsf{sid}, \mathsf{ssid}, m') \to c'$.
2. Output $(\mathsf{check}, \mathsf{name}, \mathsf{sid}, \mathsf{ssid}, \top)$ if $c' = c$ where c is the value stored for commitments with the identifiers $\mathsf{name}, \mathsf{sid}, \mathsf{ssid}$. Otherwise output $(\mathsf{check}, \mathsf{name}, \mathsf{sid}, \mathsf{ssid}, \bot)$.

Open

1. In order to open a commitment $(\mathsf{name}, \mathsf{sid}, \mathsf{ssid})$, where $c = \mathsf{H}(\mathsf{name}, \mathsf{sid}, \mathsf{ssid}, o)$ the sender sends $(\mathsf{open}, \mathsf{name}, \mathsf{sid}, \mathsf{ssid}, o)$ to the receiver.
2. Both parties compute $c' = \mathsf{H}(\mathsf{name}, \mathsf{sid}, \mathsf{ssid}, o)$ and output $(\mathsf{open}, \mathsf{name}, \mathsf{sid}, \mathsf{ssid}, o_1)$ iff $c' = c$, where o_1 is the first element of o if it is a tuple, otherwise $o_1 = o$. However, if $c' \neq c$ they output $(\mathtt{reject}, \mathsf{name}, \mathsf{sid}, \mathsf{ssid})$ where name is the name of the committing party.

Figure 2.2: Protocol Π_{COM} realizing the $\mathcal{F}_{\text{COM}}$ functionality in the ROM model.

The case for the verifiable commitments is the same as above, except that the tag $\mathtt{commit}$ is exchanged for vc.

Consider check calls. If the party making the call is honest, then no information from $\mathcal{F}_{\text{COM}}$ is given to the simulator, so nothing needs to be simulated. However, the party making the call will still get the correct output, as all vc calls have been passed on to $\mathcal{F}_{\text{COM}}$. Even if the internal flag $\mathsf{Abort}_{\mathsf{A},\mathsf{sid},\mathsf{ssid}}$ is set to true, we notice that a dummy value will have been committed to in $\mathcal{F}_{\text{COM}}$ and thus it will always return $(\mathsf{check}, \mathsf{A}, \mathsf{sid}, \mathsf{ssid}, \bot)$. On the other hand, if the party making the call is corrupted, then the simulator learns the query $(\mathsf{A}, \mathsf{sid}, \mathsf{ssid}, m')$ from her call to H. If the simulator has previously returned $(\mathsf{vc}, \mathsf{A}, \mathsf{sid}, \mathsf{ssid}, c)$ to the party making the call, where c is in the codomain of H and the internal flag $\mathsf{Abort}_{\mathsf{A},\mathsf{sid},\mathsf{ssid}}$ is false, then call $(\mathsf{check}, \mathsf{A}, \mathsf{sid}, \mathsf{ssid}, m')$ on $\mathcal{F}_{\text{COM}}$. If it returns $(\mathsf{check}, \mathsf{A}, \mathsf{sid}, \mathsf{ssid}, \top)$ then return c to the party making the call. Otherwise pick a new uniformly random value c' from the codomain of H and return this. Furthermore, if the same query is made again then return the same c'.

We now consider open calls. First consider the case where the committer, A, is honest. In this case the simulator receives $(\mathsf{open}, \mathsf{A}, \mathsf{sid}, \mathsf{ssid}, m)$ from $\mathcal{F}_{\text{COM}}$ when A inputs $(\mathsf{open}, \mathsf{A}, \mathsf{sid}, \mathsf{ssid})$. If the simulator previously got the response $(\mathtt{commit}, \mathsf{A}, \mathsf{sid}, \mathsf{ssid})$ from $\mathcal{F}_{\text{COM}}$ it samples a uniformly random $r \in_R \{0,1\}^\kappa$ and sends $(\mathsf{A}, \mathsf{sid}, \mathsf{ssid}, (m, r))$ to B. If B ever queries H on $(\mathsf{A}, \mathsf{sid}, \mathsf{ssid}, (m, r))$

then it sends c as response. If the simulator instead previously got the response $(\mathsf{vc}, \mathsf{A}, \mathsf{sid}, \mathsf{ssid})$ from $\mathcal{F}_{\mathrm{COM}}$, then if B ever queries H on $(\mathsf{A}, \mathsf{sid}, \mathsf{ssid}, m)$ then it sends c as response. Finally it returns $(\mathsf{open}, \mathsf{A}, \mathsf{sid}, \mathsf{ssid}, m)$ to B.

If the committer is corrupt, then the simulator will get $(\mathsf{open}, \mathsf{A}, \mathsf{sid}, \mathsf{ssid}, o^*)$ from the adversary. If the flag $\mathsf{Abort}_{\mathsf{A},\mathsf{sid},\mathsf{ssid}}$ is set to true then it sends $(\mathsf{NoOpen}, \mathsf{A}, \mathsf{sid}, \mathsf{ssid})$ to $\mathcal{F}_{\mathrm{COM}}$, otherwise it sends $(\mathsf{open}, \mathsf{A}, \mathsf{sid}, \mathsf{ssid})$ to $\mathcal{F}_{\mathrm{COM}}$.

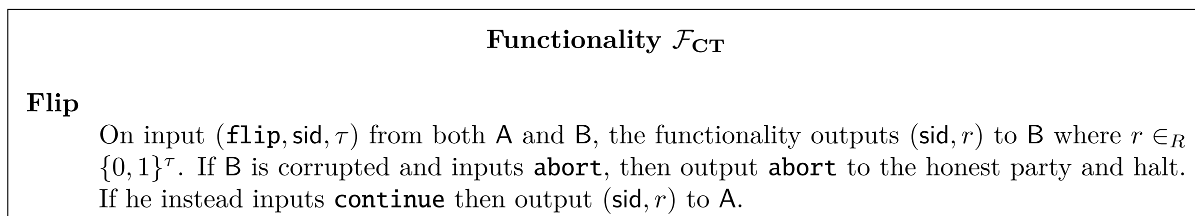
The adversary will only notice that queries to H are simulated if it queries $c' = \mathsf{H}(\mathsf{A}, \mathsf{sid}, \mathsf{ssid}, o^*)$ for which it has previously received $(\mathtt{commit}, \mathsf{A}, \mathsf{sid}, \mathsf{ssid}, c)$ or $(\mathsf{vc}, \mathsf{A}, \mathsf{sid}, \mathsf{ssid}, c)$ where $c \neq c'$. Consider the first case, i.e. that c results from a call to `commit`. In this case we notice that the suffix of o^* will be a κ bit uniformly random string and the output of H is also κ bits. Since the adversary is polynomially time bounded it can only make polynomially many of these queries and thus not guess the κ-bit suffix with non-negligible probability. In the second case we have assumed that the messages for which the verifiable commitments are constructed have min-entropy at least κ in the view of the receiving party so the result follows by the same argument as above.

Finally notice that an adversary that does not query H on a input with the correct handles when committing will result in `reject` being outputted when she tried to open the commitment, except if the adversary can guess the output of H, however since this is κ bits, she can only do so with negligible probability.

□

Coin-tossing

In order to prove security by simulation in some of our protocols, we need to have some of the random choices made to be done using a coin-tossing functionality. We define an ideal functionality for coin-tossing in Fig. 2.3 and a protocol for it in Fig. 2.4, which is more or less a multi-bit UC version of Blum's result [Blu81].



Functionality $\mathcal{F}_{\mathbf{CT}}$

Flip

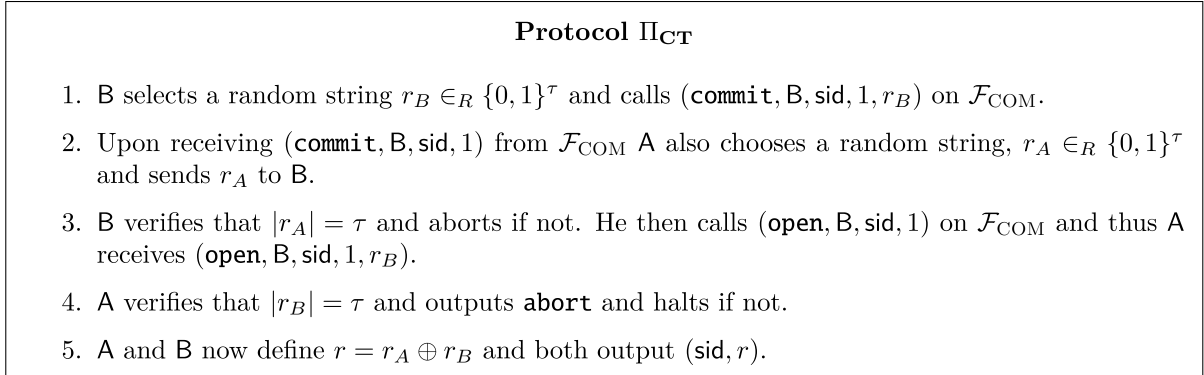
On input $(\mathtt{flip}, \mathsf{sid}, \tau)$ from both A and B, the functionality outputs (sid, r) to B where $r \in_R \{0,1\}^\tau$. If B is corrupted and inputs `abort`, then output `abort` to the honest party and halt. If he instead inputs `continue` then output (sid, r) to A.

Figure 2.3: The ideal functionality $\mathcal{F}_{\mathrm{CT}}$ for maliciously secure coin tossing of τ bits.

Theorem 2. *The protocol Π_{CT} in Fig. 2.4 UC-securely realizes the functionality $\mathcal{F}_{\mathrm{CT}}$ described in Fig. 2.3 in the $\mathcal{F}_{\mathrm{COM}}$-hybrid model against any PPT static and malicious adversary corrupting either A or B.*

Proof. We start by considering that A is corrupted and thus under the control of the adversary, which we denote by $\mathcal{A}$. We now describe a simulator S_A simulating the interaction between $\mathcal{A}$ and an honest B, using the functionality $\mathcal{F}_{\mathrm{CT}}$:

1. S_A simulates a commitment on behalf of B by sending $(\mathtt{commit}, \mathsf{B}, \mathsf{sid}, 1)$ to $\mathcal{A}$. It then receives back a string r_A from $\mathcal{A}$. S_A verifies that r_A is of length τ and if not, it aborts and halts.



Protocol $\Pi_{\mathbf{CT}}$

1. B selects a random string $r_B \in_R \{0,1\}^\tau$ and calls $(\mathtt{commit}, \mathsf{B}, \mathsf{sid}, 1, r_B)$ on $\mathcal{F}_{\mathrm{COM}}$.
2. Upon receiving $(\mathtt{commit}, \mathsf{B}, \mathsf{sid}, 1)$ from $\mathcal{F}_{\mathrm{COM}}$ A also chooses a random string, $r_A \in_R \{0,1\}^\tau$ and sends r_A to B.
3. B verifies that $|r_A| = \tau$ and aborts if not. He then calls $(\mathsf{open}, \mathsf{B}, \mathsf{sid}, 1)$ on $\mathcal{F}_{\mathrm{COM}}$ and thus A receives $(\mathsf{open}, \mathsf{B}, \mathsf{sid}, 1, r_B)$.
4. A verifies that $|r_B| = \tau$ and outputs $\mathtt{abort}$ and halts if not.
5. A and B now define $r = r_A \oplus r_B$ and both output (sid, r).

Figure 2.4: The protocol Π_{CT} realizing $\mathcal{F}_{\mathrm{CT}}$ in the $\mathcal{F}_{\mathrm{COM}}$-hybrid model.

2. Then S_A queries $\mathcal{F}_{\mathrm{CT}}$ with input $(\mathtt{flip}, \mathsf{sid}, \tau)$ and gets back a random string $r \in \{0,1\}^\tau$.
3. S_A then simulates the opening to the commitment by sending $(\mathsf{open}, \mathsf{B}, \mathsf{sid}, 1, r \oplus r_A)$ to $\mathcal{A}$.

It is easy to see that the real and ideal executions are indistinguishable. First notice that in both the real (hybrid) and ideal (simulated) world both $\mathcal{A}$ and B outputs the same τ bit random string, since $r_B = r \oplus r_A$ and so $r = r_A \oplus r_B$. We also notice that an honest B can abort if $\mathcal{A}$ sends a bitstring of the wrong length (Step 3 in the protocol) before any party learns the output. However, since S_A passes this string along directly from $\mathcal{A}$ this will happen similarly in the real and ideal world.

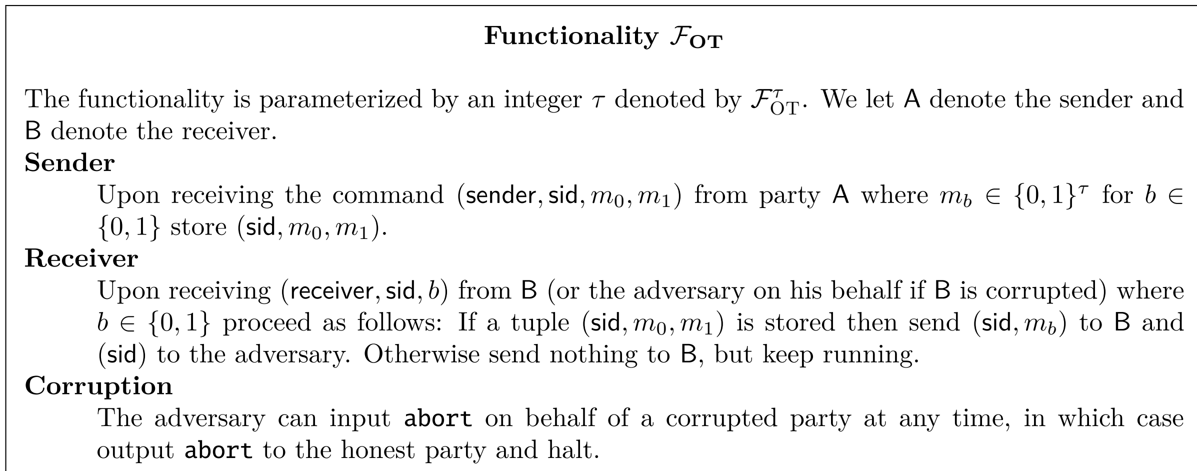
Now consider a corrupt B controlled by the adversary, denoted by $\mathcal{B}$. We now describe a simulator S_B for the execution:

1. When receiving $(\mathtt{commit}, \mathsf{B}, \mathsf{sid}, 1, r_B)$ from $\mathcal{B}$ through his call to to $\mathcal{F}_{\mathrm{COM}}$ then S_B queries $\mathcal{F}_{\mathrm{CT}}$ with input $(\mathtt{flip}, \mathsf{sid}, \tau = |r_B|)$ and gets back a random string $r \in \{0,1\}^\tau$.
2. S_B defines $r_A = r \oplus r_B$. It sends r_A to $\mathcal{B}$.
3. If $\mathcal{B}$ returns $(\mathsf{NoOpen}, \mathsf{B}, \mathsf{sid}, 1)$ then S_B inputs $\mathtt{abort}$ to $\mathcal{F}_{\mathrm{CT}}$. If $\mathcal{B}$ instead returns $(\mathsf{open}, \mathsf{B}, \mathsf{sid}, 1)$ then S_B inputs $\mathtt{continue}$ to $\mathcal{F}_{\mathrm{CT}}$.

It is easy to see that the real and ideal executions are indistinguishable by the same arguments as the case of a corrupted A. In particular $\mathcal{B}$ learns the output first and can then decide if A should also learn it. □

Oblivious Transfer

We describe the ideal function for a simple OT functionality in Fig. 2.5. This box is taken almost verbatim from [CLOS02] and expresses the one-out-of-two OT where the sender inputs two messages of equal length and the receiver inputs a single bit. If the bit is 0 he learns the first message, if it is 1 then he learns the second message, and nothing more. Furthermore, the sender learns nothing about the bit given by the receiver. This type of OT was introduced in [EGL85]. We note that several other types of OT exists, in particular the concept was introduced by Rabin [Rab81] where the sender only gives a single message as input and the receiver does not get to choose a bit, but instead learns the single message with probability one half. The sender does not learn whether or not the receiver learned the message.



Functionality $\mathcal{F}_{\mathbf{OT}}$

The functionality is parameterized by an integer τ denoted by $\mathcal{F}_{\mathrm{OT}}^{\tau}$. We let A denote the sender and B denote the receiver.

Sender

Upon receiving the command $(\mathsf{sender}, \mathsf{sid}, m_0, m_1)$ from party A where $m_b \in \{0,1\}^{\tau}$ for $b \in \{0,1\}$ store (sid, m_0, m_1).

Receiver

Upon receiving $(\mathsf{receiver}, \mathsf{sid}, b)$ from B (or the adversary on his behalf if B is corrupted) where $b \in \{0,1\}$ proceed as follows: If a tuple (sid, m_0, m_1) is stored then send (sid, m_b) to B and (sid) to the adversary. Otherwise send nothing to B, but keep running.

Corruption

The adversary can input `abort` on behalf of a corrupted party at any time, in which case output `abort` to the honest party and halt.

Figure 2.5: The ideal functionality $\mathcal{F}_{\mathrm{OT}}$.

2.4 Garbling

The idea of garbled circuits is due to Yao. The inception of this is often accredited to [Yao82] or [Yao86], the first of which introduces the notion of secure multi-party computation and the second of which expands on the security notions by introducing the concept of *fairness*. However, neither of these papers contain a description of garbled circuits as we know them today (and as described in the introduction in Chapter 1). Still, according to Goldreich [Gol03] the notion of garbled circuits, and their usage in semi-honestly secure 2PC was given by Yao during a presentation of [Yao86]. The first written description (according to Bellare *et al.* [BHR12b]) of such a protocol was given in 1987 in [GMW87] and the name, garbled circuit, was not used until 1990 in the paper [BMR90]. A formal specification of the protocol, along with an explicit proof in the real/ideal paradigm was not given until 2004 [LP09].

A Formal Framework for Garbling Schemes

Everything we have seen so far gives a nice idea of what a garbled circuit is and how to construct one. However, this has all been given in a very informal and imprecise notation. In the rest of this section we try to describe the formal framework of *garbling schemes* as presented by Bellare *et al.* in their seminal three-part work [BHR12b, BHR12a, BHKR13]. However, we augment their definitions slightly as their abstract definitions are not sufficient for our settings.

Roughly we can define a garbling scheme to be a tuple of 5 PPT algorithms:

$$\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev}) \ .$$

Letting f be a function mapping n bits to m bits then we have Gb being a randomized polytime algorithm, taking f as input along with a computational security parameter κ and outputting a triple $(F, e, d) \leftarrow^* \mathsf{Gb}(1^{\kappa}, f)$. Sometimes we will make the randomness used in the algorithm explicit and thus define the deterministic function $\mathsf{Gb}(1^{\kappa}, f; r) \rightarrow (F, e, d)$. We call F the *garbled function* (or *garbled circuit*), e the *encoding information*, and d the *decoding information*. The encoding information turns the *plain input* $x \in \{0,1\}^n$ into a *garbled input* $X = e(x)$. The garbled function can then be evaluated using the garbled input to learn the *garbled output* $Y = F(X)$. Afterwards one can use the decoding information on the garbled output to learn

the *plain output* $y \in \{0,1\}^m$, where $y = d(Y)$. The plain output should then be equal to the plain output resulting of evaluating the plain circuit on the plain input, that is $y = f(x)$.

Sometimes we will add another algorithm, Ve, not included in Bellare's *et al.* framework. This algorithm is defined from a garbling scheme itself, using it in a black box manner. It is used to verify that a tuple (F, e, d) is correctly constructed in accordance with f using some randomness r. If this algorithm is added to the scheme then we talk about a *verifiable* garbling scheme.

In the above we abused notation quite a bit (which we will continue doing for the rest of this thesis) since the values e, d and F are not in and of themselves functions, but rather bitstrings. This is where Ve and the 4 other parts of $\mathcal{G}$ come into play:

En This is a deterministic function taking as input e and x, applying the semantic meaning of e on x to return $X \leftarrow \mathsf{En}(e, x)$.

Ev This is a deterministic function taking as input F and X, "evaluating" F on X to return $y \leftarrow \mathsf{Ev}(F, X)$.

De This is a deterministic function taking as input Y and d, applying the semantic meaning of d on Y to return $y \leftarrow \mathsf{De}(d, Y)$.

ev This is a deterministic function taking as input f and x, applying the semantic meaning of f on x to return $y \leftarrow \mathsf{ev}(f, x)$.

Ve This is a deterministic function taking as input F, f, e, d and r, executing the deterministic function $\mathsf{Gb}(1^\kappa, f; r) \to (F', e', d')$. Then it evaluates equality of $(F, e, d) \stackrel{?}{=} (F', e', d')$. If this is the case 1 is returned, if not 0 is returned. That is, $\mathsf{Ve}(F, f, e, d, r) \to b$.

We illustrate the relationship between these algorithms and their input/output in Fig. 2.6.

Let us elaborate on what exactly the input and output of these algorithms consists of in order to have a well defined garbling scheme. To start of, consider how exactly we are going to define a plain function f. To do so we must first agree on when a plain circuit is legal. In Chapter 1 we already mentioned that we require it consists of fan-in 2 gates with unlimited fan-out and that the circuit can be described as a DAG. However, a few undefined details remain to be solved, such as whether a circuit input wire can be a circuit output wire. Besides this, a formal description of the circuit is also needed.

We define the plain function (or *plain circuit*) f as a 6-tuple; $f = (n, m, q, L, R, G)$ where $n, m, q \in \mathbb{N}$ and $|x| = n \geq 2$ represents the amount of circuit input wires, $|y| = m \geq 1$ represents the amount of circuit output wires and $q \geq 1$ is the number of *gates* in the circuit. We let $w = n + q$ represent the amount of wires in the circuit. We then define the sets $\mathsf{Inputs} = [n]$, $\mathsf{Wires} = [w]$, $\mathsf{Outputs} = [w - m + 1; w]$ and $\mathsf{Gates} = [n + 1; w]$. Next we let L, respectively R be functions mapping a gate (by its index), say g, to the index of the gate whose output wire is the left, respectively right input wire of gate g. That is $L : \mathsf{Gates} \to \mathsf{Wires} \backslash \mathsf{Outputs}$, respectively $R : \mathsf{Gates} \to \mathsf{Wires} \backslash \mathsf{Outputs}$. Finally, G is a Boolean function mapping a given gate index along with two bits (to represent its left and right input bit respectively) to a bit. Thus for each gate in the circuit this will be used to describe which binary gate it represents, that is $G : \mathsf{Gates} \times \{0,1\}^2 \to \{0,1\}$. Thus we might say that $G(g, \cdot, \cdot) \equiv \text{AND}$ for $g \in \mathsf{Gates}$ if we wish to have gate g compute an AND gate, similarly $G(g, \cdot, \cdot) \equiv \text{XOR}$ if we wish to have gate g compute a XOR gate or $G(g, \cdot, \cdot) \equiv \text{NAND}$ if we wish to have gate g compute a NAND gate. Furthermore, we have the constraint that $L(g) < R(g) < g$ for all $g \in \mathsf{Gates}$.

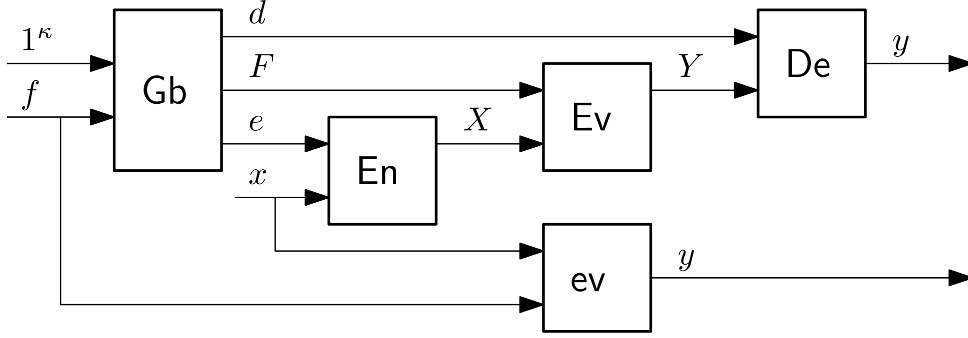


Figure 2.6: Illustration of the algorithms in a garbling scheme, $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$ in accordance with Bellare's *et al.* definitions. The randomized algorithm Gb takes as input an unary representation of the security parameter κ and a description of the plain function f, and returns the garbled circuit, encoding and decoding information (F, e and d). The encoding information e is used with the plain input x to construct the garbled input X, using the deterministic algorithm En. The garbled input X is used with the garbled circuit F to find the garbled output Y using the deterministic algorithm Ev. The garbled output Y is used along with the decoding information d to find the plain output y using the deterministic algorithm De. The plain output y, however, can also be found using the plain function f and plain input x with the deterministic algorithm ev. This figure is heavily inspired by Fig. 1 in [BHR12b].

The formalization above captures that we separate gates and wires, in that we view the circuit input wires as singular elements, not associated with a gate. Yet we let each gate be identified by an index which matches the index we associate with its output wire. This indexing, along with the constraint on the numbering of gates, makes it possible to have a topological ordering of all the gates in the circuit. This again makes it trivial to have a deterministic order in which to construct, evaluate, store, and send gates. Furthermore, it is captured by the formalization that we do not allow circuit output wires to also be input wires of other gates (because circuit output wires are the last wires) or to be used twice. Finally, we also see that the topological ordering ensures that the circuit is a DAG and that a gate cannot take the same wire as both its inputs. We will quite often abuse notation by not distinguishing a plain function from its semantic meaning and its 6-tuple, i.e. by saying $f(x) = y$ instead of $\mathsf{ev}(f, x) = y$.

Regarding the representation of the 6-tuple f, we require that $n = f.n$ and $m = f.m$ depend on f and are efficiently computable from this. In particular we require that they can be extracted in linear time from f.

For the other values used in the garbling scheme we require that $|F|$, $|e|$, and $|d|$ only depend on κ, n, m, and $|f|$. Furthermore, we demand a *length* condition; if $n = n'$, $m = m'$, and $|f| = |f'|$ with $(F, e, d) \leftarrow^* \mathsf{Gb}(1^\kappa, f)$ and $(F', e', d') \leftarrow^* \mathsf{Gb}(1^\kappa, f')$ then it must hold that $|F| = |F'|$, $|e| = |e'|$, and $|d| = |d'|$. We also demand a *non-degeneracy* condition; e and d may only depend on κ, n, m, $|f|$ and the random coins, r, of Gb. Formally, if $n = n'$, $m = m'$ and $|f| = |f'|$ with $(F, e, d) \leftarrow \mathsf{Gb}(1^\kappa, f; r)$ and $(F', e', d') \leftarrow \mathsf{Gb}(1^\kappa, f'; r)$ then it must hold that $e = e'$ and $d = d'$. We also have the *correctness* requirement stating that if $f \in \{0,1\}^*$, $\kappa \in \mathbb{N}$, $x \in \{0,1\}^n$ and $(F, e, d) \leftarrow^* \mathsf{Gb}(1^\kappa, f)$ then it must hold that $\mathsf{De}(d, \mathsf{Ev}(F, \mathsf{En}(e, x))) = \mathsf{ev}(f, x)$ except with negligible probability in κ. Finally, we also require that $n = \mathrm{poly}(\kappa)$ and $|f| = \mathrm{poly}(\kappa)$. When elements of a garbling scheme meet the requirements above we say that they are *well-defined*.

Furthermore, we say that $\mathcal{G}$ is a *circuit garbling* scheme (we will just say "garbling scheme" for short) if ev interprets f as a circuit (in accordance with the constraints of a circuit given in

the beginning of this section).

Finally, we define a specific form of garbling schemes called *projective*, which will be the only type of garbling schemes considered in this thesis. Basically a garbling scheme is projective if e can be viewed as a set of $2 \cdot n$ tokens; two for each of the n circuit input wires, where one represents the 0-key and the other the 1-key of the wire. Specifically, for $x \in \{0,1\}^n$ we have $e = (X_1^0, X_1^1, X_2^0, X_2^1, \ldots, X_n^0, X_n^1)$ and $\mathsf{En}(e, x) = (X_1^{x_1}, X_2^{x_2}, \ldots, X_n^{x_n})$. More formally we demand that for all well-defined f with $x, x' \in \{0,1\}^n$, $\kappa \in \mathbb{N}$ and $i \in [m]$ where $(F, e, d) \leftarrow^* \mathsf{Gb}(1^\kappa, f)$, $X \leftarrow \mathsf{En}(e, x)$ and $X' \leftarrow \mathsf{En}(e, x')$ then $X = (X_1, X_2, \ldots, X_n)$ and $X' = (X'_1, X'_2, \ldots, X'_n)$ are n element vectors, $|X_i| = |X'_i|$ and $X_i = X'_i$ if $x[i] = x'[i]$.

Similarly we say that a garbling scheme is *output projective* if there are exactly two possible tokens associated with each of the m circuit output wires, where one represents the 0-key and the other the 1-key of the wire. Specifically, for $y \in \{0,1\}^m$ we have a unique set $(Y_1^0, Y_1^1, Y_2^0, Y_2^1, \ldots, Y_m^0, Y_m^1)$ and $\mathsf{Ev}(F, X) \to \left(Y_1^{y[1]}, Y_2^{y[2]}, \ldots, Y_m^{y[m]}\right)$. More formally we demand that for all well-defined f with $x, x' \in \{0,1\}^n$, $\kappa \in \mathbb{N}$ and $i \in [m]$ where $y \leftarrow \mathsf{ev}(f, x)$, $y' \leftarrow \mathsf{ev}(f, x')$, $(F, e, d) \leftarrow^* \mathsf{Gb}(1^\kappa, f)$, $X \leftarrow \mathsf{En}(e, x)$ and $X' \leftarrow \mathsf{En}(e, x')$ then for $\mathsf{Ev}(F, X) \to Y$ and $\mathsf{Ev}(F, X') \to Y'$ we have that $Y = (Y_1, Y_2, \ldots, Y_m)$ and $Y' = (Y'_1, Y'_2, \ldots, Y'_m)$ are m element vectors, $|Y_i| = |Y'_i|$ and $Y_i = Y'_i$ if $y[i] = y'[i]$.

We note that the notion of output projective is not defined by Bellare *et al.* [BHR12b, BHR12a, BHKR13]. However, it seems a natural extension of the (input) projective notion and is strictly needed in the protocol we present in Chapter 4.

Having these formal, yet abstract, notions in place we still need to establish the notions of *security* we demand of our schemes and, as a minimum, a concrete instantiation.

Let us first consider security, to which end we have two distinct classes of demands:

Secrecy: We can define secrecy based on either the notion of *privacy* or *obliviousness*. For privacy we demand that a party learning (F, X, d) does not learn anything besides some allowed *leakage* and whatever information is achieved by getting to know Y (for example by computing $\mathsf{De}(d, \mathsf{Ev}(F, X))$). The interpretation of this notion is that the semantic values on the internal wires remain private towards the party who has garbled the circuit. The allowed leakage is captured through a *side-information function* $\mathbf{\Phi}$, which is queried on the plain function f and returns the allowed leakage of this function. In the case of obliviousness we assume the evaluating party does not know the decoding information d, and thus is only in possession of (F, X). We then wish that he does not learn anything about f, x or y, and thus only learns what is permitted by $\mathbf{\Phi}$. The interpretation of this notion is that everything about the function and inputs remain secret to the evaluator; he is oblivious of what he does.

Authenticity: Like obliviousness we define the notion of *authenticity* towards a party which is only given F and X. We wish that he is not able to construct a garbled output $Y^* \neq \mathsf{Ev}(F, X)$ such that $\mathsf{De}(d, Y^*) \neq \perp$. The interpretation of this notion is that one cannot construct permissible garbled output different from what is dictated by X and F.

What is leaked by $\mathbf{\Phi}$ varies from the context of the usage of the garbling scheme. For example in the setting of 2PC the leakage is often the plain function itself or its topology.

When we wish to use garbled circuits in the setting of malicious adversaries we often also need the notion of *verifiability*. This basically means that a garbled circuit can be completely "opened" to verify that it does in fact compute the function it is supposed to.

Verifiability: We define the notion of *verifiability* to make it possible for a party who did not construct F to verify that on all possible legal inputs X, F computes f. Thus we want $\mathsf{Ve}(F, f, e, d, r) \rightarrow 1$ iff F, e, and d has been honestly constructed through the call $\mathsf{Gb}(1^\kappa, f; r)$.

As previously noted Ve is defined from black box usage of the algorithms in a garbling scheme. Specifically Ve is defined as follows:

$\mathsf{Ve}(F, f, e, d, r) \rightarrow b$**:**

1. Run $\mathsf{Gb}(1^\kappa, f; r) \rightarrow (F', e', d')$.
2. If $(F, e, d) \neq (F', e', d')$ then output 0, otherwise output 1.

To formalize these notions we use *game-based* definitions. Of these, when considering the secrecy demand, there are two possible types; *simulation* based or *indistinguishability* based. We define these security games in Fig. 2.7. However, we will only consider privacy and not obliviousness, but include it in the definitions for completeness. For more information on obliviousness of garbling schemes we direct the reader to [BHR12b].

We notice that the game for verifiability (the ver property) requires the existence of a new algorithm Ext. This is an extractor which needs to run in expected polynomial time. The job of this extractor is to find a value which corresponds to a specific output value. This is needed to ensure that maliciously generated circuits still compute the correct function and does not leak anything about the input.

We let the advantage of a PPT adversary $\mathcal{A}$ playing game "game" using the verifiable garbling scheme $\mathcal{G}$ with security parameter κ and potentially an auxiliary function tuple δ, be denoted by:

$$\mathrm{Adv}_{\mathcal{G}}^{\mathrm{game},\delta}(\mathcal{A}, \kappa) \ .$$

For the games in Fig. 2.7 these are defined as follows for an arbitrary PPT adversary $\mathcal{A}$:

$$\begin{aligned}
\mathrm{Adv}_{\mathcal{G}}^{\mathrm{prv.ind},\mathbf{\Phi}}(\mathcal{A}, \kappa) &= 2\Pr\left[\mathrm{PrvInd}_{\mathcal{G},\mathbf{\Phi}}^{\mathcal{A}}\left(1^\kappa\right) = \top\right] - 1, \\
\mathrm{Adv}_{\mathcal{G}}^{\mathrm{prv.sim},\mathbf{\Phi},\mathsf{S}}(\mathcal{A}, \kappa) &= 2\Pr\left[\mathrm{PrvSim}_{\mathcal{G},\mathbf{\Phi},\mathsf{S}}^{\mathcal{A}}\left(1^\kappa\right) = \top\right] - 1, \\
\mathrm{Adv}_{\mathcal{G}}^{\mathrm{obl.ind},\mathbf{\Phi}}(\mathcal{A}, \kappa) &= 2\Pr\left[\mathrm{OblInd}_{\mathcal{G},\mathbf{\Phi}}^{\mathcal{A}}\left(1^\kappa\right) = \top\right] - 1, \\
\mathrm{Adv}_{\mathcal{G}}^{\mathrm{obl.sim},\mathbf{\Phi},\mathsf{S}}(\mathcal{A}, \kappa) &= 2\Pr\left[\mathrm{OblSim}_{\mathcal{G},\mathbf{\Phi},\mathsf{S}}^{\mathcal{A}}\left(1^\kappa\right) = \top\right] - 1, \\
\mathrm{Adv}_{\mathcal{G}}^{\mathrm{aut}}(\mathcal{A}, \kappa) &= \Pr\left[\mathrm{Aut}_{\mathcal{G}}^{\mathcal{A}}\left(1^\kappa\right) = \top\right] \\
\mathrm{Adv}_{\mathcal{G}}^{\mathrm{ver}}(\mathcal{A}, \kappa) &= \Pr\left[\mathrm{Ver}_{\mathcal{G}}^{\mathcal{A}}\left(1^\kappa\right) = \top\right] \ .
\end{aligned}$$

An Efficient Garbling Scheme

Based on the optimizations previously discussed, along with the formal specification of a garbling scheme, we define a concrete and efficient garbling scheme, which we call *GaXR*. This scheme is taken verbatim from [BHKR13]. The scheme uses the point-and-permute, free-XOR, and garbled row reduction optimizations surveyed earlier, it is specified in Fig. 2.8.

Game $\text{PrvInd}^{\mathcal{A}}_{\mathcal{G},\mathbf{\Phi}}(1^\kappa)$. **Property** prv.ind.

1. Run $\mathcal{A}(1^\kappa)$ to produce (f_0, f_1, x_0, x_1).
2. If $x_0, x_1 \notin \{0,1\}^{f_0.n}$, $\mathbf{\Phi}(f_0) \neq \mathbf{\Phi}(f_1)$ or $\mathsf{ev}(f_0, x_0) \neq \mathsf{ev}(f_1, x_1)$ then output $\bot$.
3. Sample a uniformly random bit $b \leftarrow^* \{0,1\}$.
4. Run $(F, e, d) \leftarrow^* \mathsf{Gb}(1^\kappa, f_b)$.
5. Compute $X \leftarrow \mathsf{En}(e, x_b)$.
6. Let $b' \leftarrow^* \mathcal{A}(F, X, d)$.
7. If $b' = b$ then output $\top$, otherwise output $\bot$.

Game $\text{PrvSim}^{\mathcal{A}}_{\mathcal{G},\mathbf{\Phi},\mathsf{S}}(1^\kappa)$. **Property** prv.sim.

1. Run $\mathcal{A}(1^\kappa)$ to produce (f, x).
2. Sample a uniformly random bit $b \leftarrow^* \{0,1\}$.
3. If $x \notin \{0,1\}^{f.n}$ then output $\bot$.
4. If $b = 1$ then compute $(F, e, d) \leftarrow^* \mathsf{Gb}(1^\kappa, f)$ and $X \leftarrow \mathsf{En}(e, x)$. If instead $b = 0$ let $y \leftarrow \mathsf{ev}(f, x)$ and $(F, X) \leftarrow^* \mathsf{S}(1^\kappa, y, \mathbf{\Phi}(f))$.
5. Let $b' \leftarrow \mathcal{A}(F, X, d)$.
6. If $b' = b$ then output $\top$, otherwise output $\bot$.

Game $\text{OblInd}^{\mathcal{A}}_{\mathcal{G},\mathbf{\Phi}}(1^\kappa)$. **Property** obl.ind.

1. Run $\mathcal{A}(1^\kappa)$ to produce (f_0, f_1, x_0, x_1).
2. If $x_0, x_1 \notin \{0,1\}^{f_0.n}$ or $\mathbf{\Phi}(f_0) \neq \mathbf{\Phi}(f_1)$ then output $\bot$.
3. Sample a uniformly random bit $b \leftarrow^* \{0,1\}$.
4. Run $(F, e, d) \leftarrow^* \mathsf{Gb}(1^\kappa, f_b)$.
5. Compute $X \leftarrow \mathsf{En}(e, x_b)$.
6. Let $b' \leftarrow^* \mathcal{A}(F, X)$.
7. If $b' = b$ then output $\top$, otherwise output $\bot$.

Game $\text{OblSim}^{\mathcal{A}}_{\mathcal{G},\mathbf{\Phi},\mathsf{S}}(1^\kappa)$. **Property** obl.sim.

1. Run $\mathcal{A}(1^\kappa)$ to produce (f, x).
2. Sample a uniformly random bit $b \leftarrow^* \{0,1\}$.
3. If $x \notin \{0,1\}^{f.n}$ then output $\bot$.
4. If $b = 1$ then compute $(F, e, d) \leftarrow^* \mathsf{Gb}(1^\kappa, f)$ and $X \leftarrow \mathsf{En}(e, x)$. If instead $b = 0$ let $(F, X) \leftarrow^* \mathsf{S}(1^\kappa, \mathbf{\Phi}(f))$.
5. Let $b' \leftarrow \mathcal{A}(F, X)$.
6. If $b' = b$ then output $\top$, otherwise output $\bot$.

Game $\text{Aut}^{\mathcal{A}}_{\mathcal{G}}(1^\kappa)$. **Property** aut.

1. Run $\mathcal{A}(1^\kappa)$ to produce (f, x).
2. If $x \notin \{0,1\}^{f.n}$ then output $\bot$.
3. Run $(F, e, d) \leftarrow^* \mathsf{Gb}(1^\kappa, f)$.
4. Compute $X \leftarrow \mathsf{En}(e, x)$.
5. Let $Y \leftarrow \mathcal{A}(F, X)$.
6. If $\mathsf{De}(d, Y) \neq \bot$ *and* $Y \neq \mathsf{Ev}(F, X)$ then output $\top$, otherwise output $\bot$.

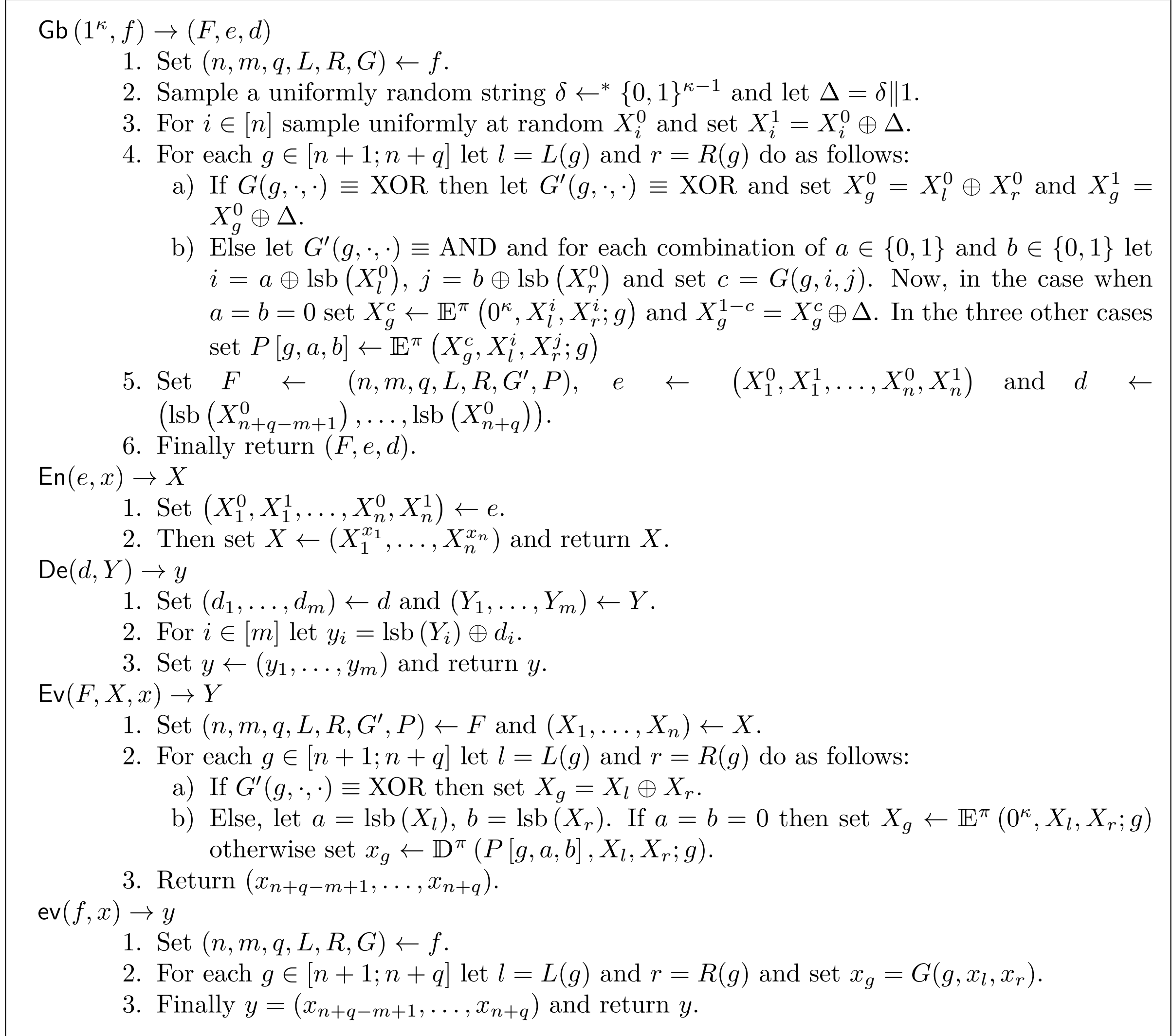
Game $\text{Ver}^{\mathcal{A}}_{\mathcal{G}}(1^\kappa)$. **Property** ver.

1. Run $\mathcal{A}(1^\kappa)$ to produce (F, f, e, d, r).
2. If $\mathsf{Ve}(F, f, e, d, r) = 0$ output $\bot$.
3. If there exists $x \in \{0,1\}^{f.n}$ s.t. $\text{Ext}(r, f, \mathsf{ev}(f, x)) \neq \mathsf{Ev}(F, \mathsf{En}(e, x))$ then output $\top$, otherwise output $\bot$.

Figure 2.7: Security games for garbling schemes

Dual-key cipher

An important aspect we have not discussed yet is the encryption scheme used to construct each of the entries in the garbled computation table. For this we require the usage of a *Dual-Key Cipher*. This is an encryption scheme that encrypts a message under two keys and some salt. Formally we say that a Dual-Key Cipher (DKC) is a function $\mathbb{E} : \Omega \times \{0,1\}^\kappa \times \{0,1\}^\kappa \times \{0,1\}^\kappa \times \{0,1\}^\tau \rightarrow \{0,1\}^\kappa$ where $\tau \in \mathbb{N}$ and Ω is the set of functions from $\{0,1\}^*$ to $\{0,1\}^\kappa$. Having $\pi \in \Omega$, $K_l, K_r \in \{0,1\}^\kappa$ and $T \in \{0,1\}^\tau$ we have $\mathbb{E}^\pi(K_o, K_l, K_r; T) : \{0,1\}^\kappa \rightarrow \{0,1\}^\kappa$ denote the function that encrypts some key $K_o \in \{0,1\}^\kappa$ under the keys K_l and K_r using the salt T to a ciphertext C. In a similar manner we define an inverse function of $\mathbb{E}$, which we denote $\mathbb{D}$ that decrypts a ciphertext constructed by $\mathbb{E}$. Formally we have $\mathbb{D} : \Omega \times \{0,1\}^\kappa \times \{0,1\}^\kappa \times \{0,1\}^\kappa \times \{0,1\}^\tau \rightarrow \{0,1\}^\kappa$



$\mathsf{Gb}(1^\kappa, f) \to (F, e, d)$

1. Set $(n, m, q, L, R, G) \leftarrow f$.
2. Sample a uniformly random string $\delta \leftarrow^* \{0,1\}^{\kappa-1}$ and let $\Delta = \delta \| 1$.
3. For $i \in [n]$ sample uniformly at random X_i^0 and set $X_i^1 = X_i^0 \oplus \Delta$.
4. For each $g \in [n+1; n+q]$ let $l = L(g)$ and $r = R(g)$ do as follows:
   a) If $G(g, \cdot, \cdot) \equiv \text{XOR}$ then let $G'(g, \cdot, \cdot) \equiv \text{XOR}$ and set $X_g^0 = X_l^0 \oplus X_r^0$ and $X_g^1 = X_g^0 \oplus \Delta$.
   b) Else let $G'(g, \cdot, \cdot) \equiv \text{AND}$ and for each combination of $a \in \{0,1\}$ and $b \in \{0,1\}$ let $i = a \oplus \text{lsb}\left(X_l^0\right)$, $j = b \oplus \text{lsb}\left(X_r^0\right)$ and set $c = G(g, i, j)$. Now, in the case when $a = b = 0$ set $X_g^c \leftarrow \mathbb{E}^\pi\left(0^\kappa, X_l^i, X_r^i; g\right)$ and $X_g^{1-c} = X_g^c \oplus \Delta$. In the three other cases set $P[g, a, b] \leftarrow \mathbb{E}^\pi\left(X_g^c, X_l^i, X_r^j; g\right)$
5. Set $F \leftarrow (n, m, q, L, R, G', P)$, $e \leftarrow \left(X_1^0, X_1^1, \ldots, X_n^0, X_n^1\right)$ and $d \leftarrow \left(\text{lsb}\left(X_{n+q-m+1}^0\right), \ldots, \text{lsb}\left(X_{n+q}^0\right)\right)$.
6. Finally return (F, e, d).

$\mathsf{En}(e, x) \to X$

1. Set $\left(X_1^0, X_1^1, \ldots, X_n^0, X_n^1\right) \leftarrow e$.
2. Then set $X \leftarrow (X_1^{x_1}, \ldots, X_n^{x_n})$ and return X.

$\mathsf{De}(d, Y) \to y$

1. Set $(d_1, \ldots, d_m) \leftarrow d$ and $(Y_1, \ldots, Y_m) \leftarrow Y$.
2. For $i \in [m]$ let $y_i = \text{lsb}(Y_i) \oplus d_i$.
3. Set $y \leftarrow (y_1, \ldots, y_m)$ and return y.

$\mathsf{Ev}(F, X, x) \to Y$

1. Set $(n, m, q, L, R, G', P) \leftarrow F$ and $(X_1, \ldots, X_n) \leftarrow X$.
2. For each $g \in [n+1; n+q]$ let $l = L(g)$ and $r = R(g)$ do as follows:
   a) If $G'(g, \cdot, \cdot) \equiv \text{XOR}$ then set $X_g = X_l \oplus X_r$.
   b) Else, let $a = \text{lsb}(X_l)$, $b = \text{lsb}(X_r)$. If $a = b = 0$ then set $X_g \leftarrow \mathbb{E}^\pi(0^\kappa, X_l, X_r; g)$ otherwise set $x_g \leftarrow \mathbb{D}^\pi(P[g, a, b], X_l, X_r; g)$.
3. Return $(x_{n+q-m+1}, \ldots, x_{n+q})$.

$\mathsf{ev}(f, x) \to y$

1. Set $(n, m, q, L, R, G) \leftarrow f$.
2. For each $g \in [n+1; n+q]$ let $l = L(g)$ and $r = R(g)$ and set $x_g = G(g, x_l, x_r)$.
3. Finally $y = (x_{n+q-m+1}, \ldots, x_{n+q})$ and return y.

Figure 2.8: GaXR

and $\mathbb{D}^\pi(C, K_l, K_r; T) : \{0,1\}^\kappa \to \{0,1\}^\kappa$ denote the function that decrypts some ciphertext $C \in \{0,1\}^\kappa$ using the keys K_l and K_r and the salt T to return K_o.

There are several different ways we can realize the DKC. In one particular method we let π be a hash function (H) (modeled as a random oracle), and then define the encryption function to be $\mathbb{E}^{\mathsf{H}}(K_o, K_l, K_r; T) = \mathsf{H}(K_l \| K_r \| T) \oplus K_o = C$ and decryption function $\mathbb{D}^{\mathsf{H}}(C, K_l, K_r; T) = \mathsf{H}(K_l \| K_r \| T) \oplus C$.

Another way is based on random permutations. In this case we talk about a particular type of DKC called σ-derived [BHKR13]:

Definition 1 (σ-derived DKC)**.** Let $\sigma : \{0,1\}^\kappa \times \{0,1\}^\kappa \times \{0,1\}^\tau \to \{0,1\}^\kappa$ be a function. We say that $\mathbb{E}$ is a σ-derived DKC if $\mathbb{E}^\pi(K_o, K_l, K_r; T) = \pi(K) \oplus K \oplus K_o$ where $K = \sigma(K_l, K_r, T)$ and the function σ satisfies the following two conditions:

1. $\sigma(K_l \oplus K_l^*, K_r \oplus K_r^*, T \oplus T^*) = \sigma(K_l, K_r, T) \oplus \sigma(K_l^*, K_r^*, T^*)$ for every $K_l, K_l^*, K_r, K_r^* \in \{0,1\}^\kappa$ and $T, T^* \in \{0,1\}^\tau$.

2. $\sigma(0^\kappa, 0^\kappa, T) \neq 0^\kappa$ unless $T = 0^\tau$.

We formally define a secure DKC by the DKC game in Fig. 2.9.

Game $\mathrm{DKC}_\pi^{\mathcal{A}}(1^\kappa)$. **Property** DKC.

1. Sample uniformly random bit $b \leftarrow^* \{0,1\}$ and a uniformly random key $K \leftarrow^* \{0,1\}^\kappa$.
2. Sample an infinite series of uniformly random strings $R_1, R_2, \ldots \leftarrow^* \{0,1\}^\kappa$.
3. For each $i \in \mathbb{N}$ let $K_{2i-1} \leftarrow^* \{0,1\}^{\kappa-1}\|0$ and $K_{2i} \leftarrow^* \{0,1\}^{\kappa-1}\|1$.
4. Give $\mathrm{lsb}(K)$ to $\mathcal{A}$ and wait for him to return a set of queries: $\{(i,j,p,T)\}$. Let $S \leftarrow \emptyset$ be an initially empty set and for each query, do as follows:
   a) If $T \in S$ or $i \geq j$ output $\perp$. Otherwise let $S = S \cup \{T\}$.
   b) If $p = 1$ then let $(K_l, K_r) \leftarrow (K, K_i)$, otherwise let $(K_l, K_r) \leftarrow (K_i, K)$.
   c) If $b = 1$ then set $X = K_j$, otherwise set $X = R_j$.
   d) Finally return $\left(K_i, K_j, \mathbb{E}^\pi(X, K_l, K_r; T)\right)$ to $\mathcal{A}$.
5. $b' \leftarrow^* \mathcal{A}$.
6. If $b' = b$ then output $\top$, otherwise output $\perp$.

Figure 2.9: Security for a DKC

Intuitively what we get from this game is that an adversary cannot tell the difference of whether a known key or a uniformly random string has been encrypted under a pair of uniformly random keys, when at most one of these is known to the adversary. The game furthermore says that this should remain true even when the keys are reused in a manner that they would be in a garbled circuit. More specifically the game first picks a bit b. If this bit is 1 it will only encrypt real keys later on, otherwise it will only encrypt random strings. It then picks the key K which will remain secret and be used as the "unknown" key for each of the adversary's queries. However, the adversary is given the least significant bit of this key. Then the random strings to be used if $b = 0$ are picked, along with pairs of random keys, where the keys with even index have LSB equal to 0 and the keys with odd index have LSB equal to 1. The adversary then gets to pick queries of ways the unknown key is used in conjunction with the other keys and, if $b = 0$ the random strings. He is allowed to reuse keys and pick his own salts (assuming he never uses the same salt twice). At each query he is given one of the encryption keys, along with the key that was encrypted (if $b = 1$) and the ciphertext. For the game we have that the advantage of the adversary is $\mathrm{Adv}_\pi^{\mathrm{DKC}}(\mathcal{A}, \kappa) = 2\Pr\left[\mathrm{DKC}_\pi^{\mathcal{A}}(1^\kappa)\right] - 1$ and we say that a DKC is *secure* if $\mathrm{Adv}_\pi^{\mathrm{DKC}}(\mathcal{A}, \kappa)$ is negligible in κ for every PPT adversary $\mathcal{A}$.

Defining Our Scheme

We will use the scheme GaXR from [BHKR13], however, our dual-key cipher will be the one used in several previous papers, such as [PSSW09, KSS12].

The garbling function Gb works by first selecting a global difference Δ uniformly at random, under the constraint that its LSB is 1. Then uniformly random 0-keys are chosen for the circuit input wires and the 1-keys are computed by XOR'ing Δ onto each of these. Afterwards each of the gates and the wire keys on their output wires are constructed in an iterative manner; if a given gate computes XOR, then no garbled computation table is stored and the gate output 0-key is defined to be XOR of the 0-keys of its left and right input wires, and again the 1-key is defined to be the 0-key XOR Δ. For all other gates, let c be the output value of the given

gate when given as input the permutation bit of its left and right input wire. Then the output c-key is defined to be the encryption of the 0-string under the left and right input keys whose external values are both 0. The output $c-1$-key is defined to be the output c-key XOR Δ as usual. The three entries in the garbled computation table are defined to be the encryptions of the output keys in correspondence with the external values of the input keys. The garbled circuit, F, is then defined to be the garbled computation tables along with the same tuples as used to describe the plain function, but where the gate descriptions of non-XOR gates are all, arbitrarily, set to be AND gates. The encoding information, e, is set to be the pair of 0- and 1-keys for each of the circuit input wires. Finally, the decoding information, d, is set to be the pair of 0- and 1-keys for each of the circuit output wires.

The evaluation function Ev works by first extracting the keys of the input wires from X, then each of the gates are evaluated in an iterative manner; if a given gate computes XOR then no garbled computation table is stored and the output wire key of this gate is simply set to be the XOR of the input wire keys of that gate. For all other gates there are two cases; if the LSB of both the left input key and the right input key is 0 then the output wire key is defined to be the encryption of the 0-string under the input keys. Otherwise, the output wire key is defined to be the decryption of the entry in the garbled computation table described by the external values. In the end the output wire keys of the circuit are returned.

The encoding function En works by first extracting the key pairs of the input wires from e and then for each bit of input, taking the 0-key, if the input bit was 0, and the 1-key if the input bit was 1. The decoding function works by first extracting the key pairs of the decoding information d, and the output wire keys from Y. For each key pair it compares to the output key and lets the plain output be 0 if the output key is equal to the 0-key in the pair and 1 if the output key is equal to the 1-key in the pair.

The plain evaluation function, ev, simply extracts each bit of the input in x and evaluates each gate one at a time, based on the gate descriptions. Finally, it returns the bits of the last m wires in the circuit.

Since the GaXR scheme is verifiable and we also define the extractor Ext needed for the $\mathrm{Ver}_{\mathcal{G}}^{\mathcal{A}}(1^{\kappa})$ game as follows, where $y = \mathsf{ev}(f, x)$:

$\mathrm{Ext}(r, f, y) \to Y'$**:**

1. Compute $(F, e, d) \leftarrow \mathsf{Gb}(1^{\kappa}, f; r)$.
2. Parse $(n, m, q, L, R, G) \leftarrow f$, $(n, m, q, L, R, G', P) \leftarrow F$ and $\left(X_1^0, X_1^1, \ldots, X_n^0, X_n^1\right) \leftarrow e$.
3. Set $\Delta = X_1^0 \oplus X_1^1$.
4. For each $g \in [n+1; n+q]$ let $l = L(g)$ and $r = R(g)$ do as follows:
   a) If $G(g, \cdot, \cdot) \equiv \mathrm{XOR}$ then set $X_g^0 = X_l^0 \oplus X_r^0$.
   b) Else let $a = \mathrm{lsb}\left(X_l^0\right)$, $b = \mathrm{lsb}\left(X_r^0\right)$, set $c = G(g, a, b)$ and $X_g^c \leftarrow \mathbb{E}^{\pi}(0^{\kappa}, X_l^0 \oplus (a \cdot \Delta), X_r^0 \oplus (b \cdot \Delta); g)$. If $c = 1$ compute $X_g^0 = X_g^1 \oplus \Delta$.
5. Return $\left(X_{w-m+1}^{y[1]}, \ldots, X_w^{y[m]}\right)$.

In relation to this garbling scheme (and many others) we can define several features.

Definition 2 (Gate Garbling Scheme)**.** We say that a *projective* and *output projective* garbling scheme $\mathcal{G}$ is a *Gate Garbling Scheme* if for all well-defined $f \to (n, m, q, L, R, G)$ with $\kappa \in \mathbb{N}$ where $(F, e, d) \leftarrow^{*} \mathsf{Gb}(1^{\kappa}, f)$ it is possible to efficiently and uniquely parse $F \to (\gamma, \delta_{n+1}, \delta_{n+2}, \ldots, \delta_w)$ such that there exists an efficient and well-defined procedure $\mathsf{GEv}\Big(\delta_g, X_l^a,$

$X_r^b, \gamma_g\Big) \rightarrow X_g^{G(g,a,b)}$ for all $a, b \in \{0,1\}$, $l = L(g)$, $r = R(g)$ for $g \in [n+1; w]$. Furthermore, it must hold that for $j \in [m]$ and $b \in \{0,1\}$ that $Y_j^b = X_{w-m+j}^b$.

We note that the above definition only makes sense if the leakage function as a "minimum" leaks the topology of the circuit. For convenience we define $\mathbf{\Phi}_{\mathsf{topo}}$ to be the leakage function leaking the topology of the circuit. That is, letting $f \rightarrow (n, m, q, L, R, G)$ we have $\mathbf{\Phi}_{\mathsf{topo}}(f) = (n, m, q, L, R)$.

Definition 3 (Key Size Preserving)**.** We say that a *projective* and *output projective* garbling scheme $\mathcal{G}$ is *key size preserving* if for all well-defined f, $x \in \{0,1\}^n$, $\kappa \in \mathbb{N}$ and $i \in [n]$ where $(F, e, d) \leftarrow^* \mathsf{Gb}(1^\kappa, f)$, $(X_1, X_2, \ldots, X_n) \leftarrow \mathsf{En}(e, x)$ then for $\mathsf{Ev}(F, X) \rightarrow (Y_1, Y_2, \ldots, Y_m)$ it holds that $|X_i| = |Y_j|$ for $j \in [m]$.

Furthermore, if $\mathcal{G}$ is also a *gate garbling scheme* it must also hold that $|X_i| = |X_g^b|$ for $b \in \{0,1\}$ and $g \in [n+1; w]$.

Finally if it also holds that $X_i^b, X_g^b, Y_j \in \{0,1\}^*$ then we say the scheme is a *binary key size preserving* gate garbling scheme.

Definition 4 (Free-XOR Gate Garbling Scheme)**.** We say that a gate garbling scheme $\mathcal{G}$ is a *Free-XOR Gate Garbling Scheme* if it is a gate garbling scheme and for all well-defined f, $\kappa \in \mathbb{N}$ where $(F, e, d) \leftarrow^* \mathsf{Gb}(1^\kappa, f)$ the following holds:

1. There is a single value Δ and a public, efficiently computable binary and symmetric function L, such that for all $i \in [w]$ it holds that $\mathsf{L}(X_i^0, X_i^1) = \Delta$.

2. For each gate $g \in [n+1; w]$ where $G(g, \cdot, \cdot) \equiv \text{XOR}$ we demand that for $l \in L(g)$ and $r \in R(g)$ there exist a public, efficiently computable binary function E for which it holds that $\mathsf{E}\left(X_l^a, X_r^b\right) \rightarrow X_g^{a \oplus b}$ for $a, b \in \{0,1\}$.

We note that the above definition only makes sense if both the topology is leaked (as in any gate garbling scheme) and which gates compute XOR. For convenience we define $\mathbf{\Phi}_{\mathsf{xor}}$ to be the leakage function leaking the topology and positions of XOR gates of the circuit. That is, letting $f \rightarrow (n, m, q, L, R, G)$ we have $\mathbf{\Phi}_{\mathsf{xor}}(f) = (n, m, q, L, R, G')$ where for each gate $g \in [n+1; w]$ where $G'(g, \cdot, \cdot) \equiv \text{XOR}$ and, arbitrarily $G'(g, \cdot, \cdot) \equiv \text{AND}$ otherwise.

Definition 5 (Structure Free Gate Garbling)**.** We say that a (free-XOR) gate garbling scheme $\mathcal{G}$ is *structure free* if for $a, b \in \{0,1\}$, $l = L(g)$, $r = R(g)$ and $g \in [n+1; w]$ (where $G(g, \cdot, \cdot) \not\equiv \text{XOR}$) it holds that for any two pairs $\left(\bar{X}_l^0, \bar{X}_l^1\right)$ and $\left(\bar{X}_r^0, \bar{X}_r^1\right)$ where $\bar{X}_p^0 \neq X_p^0$ or $\bar{X}_p^1 \neq X_p^1$ for $p = l$ or $p = r$ then it holds that the set $\left\{\mathsf{GEv}\left(\delta_g, \bar{X}_l^a, \bar{X}_r^b, \gamma_g\right)\right\}_{a,b \in \{0,1\}}$ contains at least two elements except with negligible probability in κ for all honestly generated garbled circuits.

To get a better understanding of structure freeness consider the following. Assume we are given two pairs of keys, one pair of keys for the left input wire of a gate, and one pair for the right input wire of a gate. Furthermore, assume that at most one key in each of these pairs is equal to a "correctly constructed" key in correspondence with the keys defined from the honest garbling. Structure freeness then states that for all four possible pairings of a key from the left input wire pair and a key from the right input wire pair, evaluating a correctly constructed garbled gate will yield at least two possible output keys. This means that if we try to evaluate a correctly constructed gate on a pair of gate input keys where at most one is "correct" then

the output key will be incorrect, and thus distinct from one of the correct output key as defined by the garbling.[2]

Features of Our Scheme

Privacy. The GaXR scheme presented in Fig. 2.8 is proved to have prv.ind in [BHKR13] when the leakage function is $\mathbf{\Phi}_{\mathsf{xor}}$ and the DKC is σ-derived, in the random permutation model.[3] However, it is clear to see that the scheme will still have privacy in the ROM, when $\pi(K) \oplus K$ is replaced with $\mathsf{H}(K)$, as the output of H will be uniformly random for a specific input and the adversary has no option of inverting the function. Thus, we intuitively restrict the adversary's power by moving from the RPM to the ROM.

Verifiability. We argue that GaXR has property ver. To win the verifiability game the adversary must construct F, f, e, d and randomness r such that $\mathsf{Gb}\left(1^\kappa, f; r\right) \to (F', e', d') = (F, e, d)$ but where there exists a $x \in \{0,1\}^{f.n}$ such that $\mathsf{Ev}(F, \mathsf{En}(e, x))$ is different from the output keys corresponding to the keys from the garbling of f using r when the output is $\mathsf{ev}(f, x)$. However, he cannot succeed in this since there is only one possible output key for each bit on each wire and during an honest evaluation one of these keys (in correspondence with f) will be restored with probability 1. In particular the two possible keys of a given wire will always be distinct because of the permutation bits.

Free-XOR gate garbling. We see that it is clear that the GaXR scheme is both projective and output projective since we have $e \to \left(X_1^0, X_1^1, \ldots, X_n^0, X_n^1\right)$ and that each wire in the circuit (in particular the output wires) can take one of two possible values of equal size. Next, see that it is in fact also a gate garbling scheme since F is of the form $(\gamma, \delta_{n+1}, \ldots, \delta_w)$ where $\gamma = (n, m, q, L, R, G')$ and δ_{n+g} is the g'th entry in the table P. Using this we can define the gate evaluation method as follows:

$\mathsf{GEv}\left(P[g,a,b], X_l^a, X_r^b, G'(g,\cdot,\cdot)\right) \to X_g^{G(g,a,b)}$:

1. If $G'(g,\cdot,\cdot) \equiv \text{XOR}$ then set $X_g = X_l \oplus X_r$.
2. Else, let $a = \text{lsb}\left(X_l\right)$, $b = \text{lsb}\left(X_r\right)$. If $a = b = 0$ then set $X_g \leftarrow \mathbb{E}^\pi\left(0^\kappa, X_l, X_r; g\right)$ otherwise set $X_g \leftarrow \mathbb{D}^\pi\left(P\left[g,a,b\right], X_l, X_r; g\right)$.

To continue, notice that the scheme is also (binary) key size preserving as all possible wire keys have the same length. Furthermore, see that it is also a free-XOR scheme when $\mathsf{L} = \mathsf{E} = \oplus$.

From the above discussion we can thus state the following proposition:

Proposition 1. *The garbling scheme GaXR in Fig. 2.8 is a binary key size preserving, free-XOR gate garbing scheme with the properties* prv.ind. *and* ver. *and has the leakage function* $\mathbf{\Phi}_{\mathsf{xor}}$.

Finally, we see that the scheme is also structure free when we implement the DKC using a hash function in the ROM:

[2]One might extend this notion to also require that at most one of the outputs is equal to a "correct" output key.

[3]Since $\mathbf{\Phi}_{\mathsf{xor}}$ is efficiently invertible [BHKR13] it means that the scheme GaXR also has the property prv.sim based on the relation between indistinguishably and simulation as described in [BHR12b].

Proposition 2. *The garbling scheme GaXR in Fig. 2.8 is structure free when the encryption function is* $\mathbb{E}^{\mathsf{H}}(K_o, K_l, K_r; T) = \mathsf{H}(K_l \| K_r \| T) \oplus K_o = C$ *and decryption function* $\mathbb{D}^{\mathsf{H}}(C, K_l, K_r; T) = \mathsf{H}(K_l \| K_r \| T) \oplus C$ *where* H *is a non-programmable random oracle.*

Proof. The statement follows directly from the permutation bits: Using the notation of Definition 5, if $\left\{\bar{X}_l^a, \bar{X}_l^{1-a}, \bar{X}_r^b, \bar{X}_r^{1-b}\right\}_{a,b\in\{0,1\}} = \left\{X_l^a, X_l^{1-a}, X_r^b, X_r^{1-b}\right\}_{a,b\in\{0,1\}}$ then $\left|\{\mathsf{GEv}(P[g,a,b], \bar{X}_l^a, \bar{X}_r^b, G'(g,\cdot,\cdot))\}_{a,b\in\{0,1\}}\right| = 2$. If this is not the case, then we notice that if $\mathrm{lsb}(\bar{X}_l^a) = \mathrm{lsb}(\bar{X}_r^b) = 0$ we have that $\mathsf{GEv}\left(P[g,a,b], \bar{X}_l^a, \bar{X}_r^b, G'(g,\cdot,\cdot)\right)$ will be the digest of a random oracle query, on these inputs. Now, if $\bar{X}_l^a \neq X_l^0$ or $\bar{X}_l^a \neq X_l^1$ for any $a \in \{0,1\}$ or $\bar{X}_r^b \neq X_r^0$ or $\bar{X}_r^b \neq X_r^1$ for any $b \in \{0,1\}$ then the digest returned by the oracle will be different from both the real output keys (X_g^0, X_g^1), except with negligible probability. Now see that any other combination of two input keys, $\bar{X}_l$, $\bar{X}_r$ will decrypt to either one of the real output keys, X_g (if $\bar{X}_l = X_l$ and $\bar{X}_r = X_r$) or a uniformly random value (otherwise). In either case we will have a set of output descriptions of size at least 2. However, if $\mathrm{lsb}(\bar{X}_l^a) = \mathrm{lsb}(\bar{X}_r^b) \neq 0$ for any pair $a, b \in \{0,1\}$ then the gate evaluation procedure will try to decrypt an entry of the garbled table on at least two distinct combinations of input keys, e.g. the pair $\left(X_l^a, X_r^b\right)$ and $\left(X_l^{1-a}, X_r^b\right)$. However, this will be done by XOR'ing the output of the random oracle on two different inputs with the same value (the entry of the table), which will result in two distinct values except with negligible probability. Again, we see that we will have a set of output descriptions of size at least 2. □

Other Garbling Schemes

Besides statically secure garbled circuits working on a Boolean circuit of fan-in 2 gates, it is possible to construct garbled circuits in other settings. We here outline a few of these other types of garbling.

Adaptive Garbling

It is possible to extend the definitions above to yield security against an adaptive adversary [BHR12a]. In particular meaning that the corrupted party's input may depend on information already released by the honest parties. For the setting of garbled circuits this means that (part of) the input x may depend on F. This is needed in applications such as one-time programs and secure outsourcing. Thus a garbled circuit should be simulatable before the input is given. For privacy for example, this means that the simulator will not have access to the circuit output when it must construct the garbled circuit. It only becomes aware of the plain circuit output when it must construct the garbled input.

Bellare *et al.* [BHR12a] present generic solutions for transforming statically secure garbling schemes into adaptively secure garbling schemes. The general idea is simply to one-time pad the garbled circuit and decoding information using uniform randomness during garbling, and letting these pads be part of the encoding information. When it becomes time to learn the garbled input, then this is done as in the statically secure garbling scheme, but the pads for the garbled circuit and decoding information are also given. Now, during evaluation and decoding it is a simple matter of using a XOR operation to remove the padding. This approach works since the one-time pad will ensure that the transformed garbled circuit and decoding information is uniformly random and thus easy to simulate. If we are in the ROM then the padding can be

constructed from a seed with size of the security parameter, and thus save significantly in the space used to represent the transformed garbled circuit.

The above solution only works for privacy, if we wish obliviousness and/or authenticity it also becomes necessary to use a Pseudorandom Function (PRF) (or a random oracle in the ROM) to construct a tag on the one-time pad for the decoding information. This is needed as the adversary could forge a fake pad and in some schemes thus change the output of the decoding function.

Information Theoretic Garbled Circuits

Kolesnikov [Kol05] considered an approach to information theoretic garbled circuits, which he called *Gate Evaluation Secret Sharing* (GESS). The overall idea is to use secret sharing instead of a symmetric encryption scheme or hash function. Thus the output of a given gate is seen as the secret of a secret sharing scheme, where the two possible values on each of the two input wires to a given gate are seen as the shares which are used to reconstruct two possible secrets on the output wire. The scheme is constructed "top down", from the output wires down to the input wires. Each input wire key to a gate will contain shares that can be combined with a given gate's other input wire key to construct the output wire key through a simple XOR operation. This means that we in fact do not have actual garbled gates (as separate unit of data) but instead long input wire keys that can be repeatedly used to construct shorter output wire keys. Because of the increase in size of keys on each wire in an information theoretically garbled circuit, it is only possible to efficiently construct low depth garbled circuits using this approach.

Arithmetic Garbled Circuits

Applebaum *et al.* introduced in [AIK14] the notion of arithmetic garbled circuits. These garbled circuits work over an arbitrary (bounded) ring, and so the gates in such a circuit do either multiplication, addition or subtraction of two ring elements. How to lift a Boolean garbling scheme to the arithmetic setting is not immediately apparent. One approach might be the same as in Fig. 2.8 (and many other garbling schemes), which is to encrypt each entry of the truth table for each gate individually.[4] If the cardinality of the ring is big then this will lead to an undesirable blowup in size of the garbled circuit, and will be infeasible for superpolynomially sized rings. Alternatively, we could "emulate" computation in a ring by implementing its operations using Boolean gates. Even though this is asymptotically reasonable, it is still an undesirable approach since the concrete efficiency will be low and there might be situations where it is not possible to access the particular bits of a ring element.

Instead Applebaum *et al.* suggest to use affine functions to define circuit input keys (which are vectors in the ring of length polynomial in the security parameter). The garbled circuit is then constructed layer-wise by all gates in a given layer at a time. A layer is garbled by making sure the output is affine, which can be achieved using a key shrinking gadget. This gadget basically uses a special type of functional encryption. The authors present two approaches; one based on the Learning With Errors (LWE) problem and another only assuming the existence of one-way functions. The latter being less efficient than the first.

[4] The scheme in Fig. 2.8 is a bit better than this, since it allows to remove a row from the truth table.

Reusable Garbled Circuits

All previously discussed garbling schemes are *one-time*, meaning that no security is guaranteed against an adversary that receives the garbling of two different inputs for the same garbled circuit. A recent line of work considers *reusable garbled circuits* [GKP+13] and their (asymptotic) overhead [GGH+13]. Starting with [GKP+13] the authors manage to construct a reusable garbled circuit with privacy. That is, a garbled circuit with a notion of privacy, which can be evaluated on polynomially many inputs without compromising its security. They construct these using a functional encryption scheme for general functions, whose ciphertext only grows with the depth of the function to be evaluated. They give a reduction from any attribute based encryption scheme and fully homomorphic encryption scheme to such a functional encryption scheme.

The work on reusable garbled circuits is continued in [GGH+13] where the authors manage to construct reusable garbled circuits with size only bounded by the size of the plain circuit and an additive overhead which is polynomial in the security parameter and depth of the circuit. They achieve this using a new construction of attribute based encryption.

While the concept of reusable garbled circuits has numerous applications in establishing important theoretical feasibility results, their use of computationally heavy crypto machinery makes them (still) far from being practical.

Other Work on Garbling Schemes

As previously mentioned the observation that it is possible to construct XOR gates for free in some garbling schemes was introduced by Kolesnikov and Schneider in [KS08]. They prove their construction secure in the non-programmable random oracle model, but conjecture that some sort of correlation-robust hash function will suffice to prove security. This statement is investigated by Choi *et al.* [CKKZ12] where they reduce the security of their construction to a circular notion of 2-correlation robustness of the hash function used for encryption. Later Applebaum [App13] investigated whether or not it is possible to construct a free-XOR garbling scheme secure in the standard model. His point of departure is a new definition of a combination of *Related Key* and *Key Dependent Message* attack security (RK-KDM) for symmetric encryptions schemes. He proves that the free-XOR construction is secure using a symmetric encryption scheme which is RK-KDM. He then shows how to realize such a scheme based on the Learning Parity with Noise (LPN) problem.

Part II

Privacy-Free Garbling

Chapter 3

Privacy-Free Garbling

In this chapter we present the work of [FNO15]. In particular, we consider garbling schemes which offer authenticity but not privacy or obliviousness. We show how to construct such garbling schemes more efficiently[1] than garbling schemes offering the "full security" of privacy (or obliviousness) and authenticity, thus showing a *natural* separation between the notions of secrecy and authenticity of garbling schemes. Furthermore, we show that these "privacy-free" garbling schemes can be used in some real settings, such as in the protocol of Jawurek *et al.* [JKO13] for doing Zero Knowledge Argument of Knowledge (ZKAoK) of non-algebraic statements.

We present the following concrete garbling schemes, each working with their specific, previously known optimization:[2]

Privacy-Free GRR1, cheap XOR: In this garbling scheme we only send one ciphertext for each garbled gate (both XOR and non-XOR gates). The circuit evaluator uses 3 calls to a KDF for each non-XOR gate, and none for each XOR gate (so from a computational point of view, XOR gates are free). The scheme combines the row reduction technique with non-oblivious gate evaluation.

Privacy-Free GRR2, free-XOR: In this garbling scheme we send two ciphertexts for each encrypted non-XOR gate, and XOR gates are "for free". The circuit evaluator uses 3 calls to a KDF for each non-XOR gate (and none for XOR gates). The scheme is similar to GRR1, but using the free-XOR technique reduces the degrees of freedom we have in choosing the output keys and therefore requires higher communication complexity for non-XOR gates.

Privacy-Free fleXOR: In this garbling scheme we combine either our GRR1 or GRR2 scheme with the fleXOR technique of [KMR14]. The cost of non-XOR gates is unchanged from the underlaying scheme, i.e. 1 or 2 ciphertexts per gate, but now the cost of a XOR gate depends on the structure of the circuit: XOR gates require no cryptographic operations, while for communication, depending on the circuit structure, XOR gates require communication of $2, 1$ or 0 ciphertexts. Also note that our fleXOR variant, being tailored for privacy-free garbled circuits, performs better than the original.

[1] At the time publication of the preprint version of [FNO15]. It has since been outdone in efficiency by [ZRE15].

[2] The naming convention here follows [PSSW09], where GRR stands for *garbled row reduction.* We use the number following GRR to indicate the amount of ciphertexts needed in a garbled gate of fan-in 2. Thus GRR1 has one ciphertext in each garbled table and GRR2 has two ciphertexts in each garbled table.

Furthermore, we present a formal generalization of garbling schemes for gates with arbitrary fan-in and show how to construct each of our privacy-free schemes in such a setting. It turns our that all types of our privacy-free garbled gates yield even more significant improvements in computation (and in some settings also communication) over general garbled garbles when fan-in is larger than two.

Concurrent and follow-up work. It should be noted that independently from [FNO15] Ishai and Wee [IW14] defined the notion of *partial garbling*: like us, they noticed that in some applications related to secure computation, one party controls all the inputs and therefore it is possible to construct garbling schemes which are more efficient than traditional ones. However they develop this observation in a very different direction compared to us: the two works use different abstraction models (*garbling schemes* vs. *randomized encodings*), are useful for different tasks, and use completely different techniques.

Furthermore, recently Zahur *et al.* [ZRE15] constructed a gate garbling scheme, demonstrating that it is possible to combine (in a very clever way) two privacy-free garbling schemes – where each party knows all of the inputs for one of the two garbled circuits – into a garbling scheme which guarantees privacy and is more efficient than existing ones, in terms of both computational and communication complexity. Their results yield more efficient garbling schemes, both in the general setting and the privacy-free setting. Interestingly though, their privacy-free garbling scheme is more efficient than their most efficient general garbling schemes, giving more credence to the hypothesis that privacy-free garbling is strictly more efficient than general garbling.

Outline

We start by giving a formal definition of a privacy-free garbling scheme in Section 3.1, using the notation of [BHR12b] introduced in Chapter 2. We continue by giving an intuitive description of our privacy-free garbling schemes in Section 3.2 and Section 3.3. In Section 3.4 we consider how much more efficiently we can garble certain circuits using our privacy-free schemes. Finally, we sketch the setting of ZKAoK of non-algebraic statements, where privacy-free garbling is sufficient, in Section 3.5.

3.1 Defining Our Garbling Scheme

We start by considering a plain description of a Boolean circuit with a single output bit, consisting of Boolean gates having arbitrary fan-in. We formalize this using the same notation and paradigm as we gave in Chapter 2, but generalized to support gates with arbitrary fan-in along with non-oblivious gate evaluation. Furthermore, for simplicity we do wlog restrict the setting to only consider circuits with a single output, that is $m = 1$ and $\mathsf{Outputs} = \{n + q\}$.

To support arbitrary fan-in we add a function I, mapping each element of Gates to an integer describing the fan-in of a gate, i.e., $I : \mathsf{Gates} \to \mathbb{N}$. We also remove the functions L and R, and replace them with the function W. This function maps an element of Gates, along with an integer i (representing a gate's i'th input wire) to an element in Wires. When calling W on some $g \in \mathsf{Gates}$ we require that the i'th input wire is in $[I(g)]$, otherwise we return $\perp$. Thus, the signature for the method is $W : \mathsf{Gates} \times \mathbb{N} \to \{\mathsf{Wires}\backslash\mathsf{Outputs}\}^* \cup \{\perp\}$. We further require that $W(g, i) < W(g, i + 1) < g$ for all $g \in \mathsf{Gates}$ and $i \in [I(g) - 1]$ in order to avoid circularities in the circuit description.

Furthermore, we modify the function G such that it now takes as input an element of Gates along with an array of bits and returns a single bit or $\perp$. That is, $G : \mathsf{Gates} \times \{0,1\}^* \to \{0,1\} \cup \{\perp\}$. Specifically G is a description of the functionality of each gate in the circuit along with a short-circuit features such that $\perp$ is returned if the amount of elements in the binary input array is not equal to the integer returned by I when queried on the same gate index. More formally $G\left(g, (b_i)_{i\in[I(g)]}\right) \in \{0,1\}$ for all $g \in \mathsf{Gates}$, $b_i \in \{0,1\}$ and $\perp$ otherwise. Sometimes we abuse notation and simply write $G(g,b)$ if $g \in \mathsf{Gates}$ and $b \in \{0,1\}^t$ when $I(g) = t$. We also say $G(g,\cdot) \equiv \text{NAND}$ or $G(g,\cdot) \equiv \text{XOR}$ if the truth table constructed from G is the truth table of a NAND or XOR gate respectively. Thus, we now have that $f = (n, q, I, W, G)$.

Regarding the algorithms of the scheme, we notice that it now makes sense to give the plain input, x to the evaluation functionality. Furthermore, we will again also assume the existence of a verifiability function, Ve used to verify the construction.

Thus we define a *verifiable* projective garbling scheme by a tuple

$$\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev}, \mathsf{Ve}) \ ,$$

such that:

$\mathsf{Gb}\left(1^\kappa, f\right) \to^* (F, e, d)$**:** The *garbling function.* A randomized algorithm that takes as input a security parameter 1^κ and a description of a Boolean function $(n, q, I, W, G) \leftarrow f$. The function outputs a triple (F, e, d) representing a garbled circuit, F, input encoding information, e, and output decoding information d. Sometimes we make the randomness explicit and thus define a similar *deterministic* function $\mathsf{Gb}\left(1^\kappa, f; r\right) \to (F, e, d)$, where r is some explicit randomness.

$\mathsf{En}(e, x) \to X$**:** The *encoding function.* A deterministic function that uses the input encoding information e to map an input x to a *garbled input* X. In this chapter we are only interested in projective schemes and therefore we do not use the En function explicitly, but instead let $e = (X_1^0, X_1^1, \ldots, X_n^0, X_n^1)$.

$\mathsf{De}(d, Y) \to y$**:** The *decoding function.* A deterministic functionality that, using the string d, decodes the encoded output Y into a plaintext bit, y. We are only interested in whether $y = 1$ (e.g., the NP relation accepts in the ZK setting), therefore we let $d = Y^1$ and $\mathsf{De}(d, Y)$ outputs $y = 1$ if $Y \stackrel{?}{=} Y^1$ and $y = 0$ otherwise.

$\mathsf{Ev}(F, X, x) \to Y$**:** The *evaluation function.* A deterministic functionality that produces an encoded output Y by evaluating a garbled circuit F on an encoded input X. We assume that for fixed F, the evaluation can output at most two values Y^0 and Y^1.

$\mathsf{ev}(f, x) \to y$**:** The *plaintext evaluation function.* A deterministic functionality that evaluates the plain function described by f on some input x, i.e., $\mathsf{ev}(f, x) = f(x)$. It returns a bit $y \in \{0,1\}$ representing the value of the single output wire of f.

$\mathsf{Ve}(F, f, e, d, r) \to b$**:** The *verification function.* A deterministic functionality that on input a garbled circuit F, a plain function, f, encoding, and decoding information, e, d along with the randomness r used in the garbling, returns a bit b. The bit $b = 1$ if the garbled circuit F computes the functionality of f and e, d is the encoding and decoding information resulting from a garbling of f using r as the randomness. Otherwise the functionality outputs $b = 0$.

Like for the general garbling schemes, we require *correctness*, which is almost the same as for the general garbling schemes:

Definition 6 (Correctness)**.** Let $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev}, \mathsf{Ve})$ be a verifiable garbling scheme. We say that $\mathcal{G}$ enjoys *correctness* if for all $n = \text{poly}(\kappa)$, well-defined $f : \{0,1\}^n \to \{0,1\}$ and all $x \in \{0,1\}^n$ s.t. $\mathsf{Ev}(f, x) = 1$ and $(F, e, d) \leftarrow^* \mathsf{Gb}(1^\kappa, f)$ then it holds that $\mathsf{De}(d, \mathsf{Ev}(F, \mathsf{En}(e, x))) = 1 = \mathsf{Ev}(f, x)$ except with negligible probability in κ.

The *authenticity* and *verifiability* requirements are defined like the games in Fig. 2.7, except that we add the following constraint:[3]

- If $m \neq 1$ or $\mathsf{ev}(f, x) = 0$ then output $\perp$.

The algorithm Ve is defined in a black box manner from the other algorithms in the garbling scheme in the same manner as in Chapter 2.

With these games in hand we define a privacy-free garbling scheme:

Definition 7 (Privacy-Free Garbling Scheme)**.** Let $\mathcal{G}$ be a verifiable garbling scheme described as above. If this scheme enjoys *correctness* in accordance with Definition 6 and $\text{Adv}_{\mathcal{G}}^{\text{aut}}(\mathcal{A}, \kappa)$ and $\text{Adv}_{\mathcal{G}}^{\text{aut}}(\mathcal{A}, \kappa)$ are negligible in κ then $\mathcal{G}$ is a secure privacy-free garbling scheme.

Key Derivation Function

We are going to use a "compressing" key derivation function $\mathsf{KDF} : \{0,1\}^* \to \{0,1\}^\kappa$ mapping an arbitrary binary string to a pseudorandom string of κ bits. The applications of the function will be of the form $P = \mathsf{KDF}(K_1, \ldots, K_t; T)$ for some $t \in \mathbb{N}$, where $K_i \in \{0,1\}^\kappa$ is a wire key and $T \in \{0,1\}^*$ is a unique label or salt.

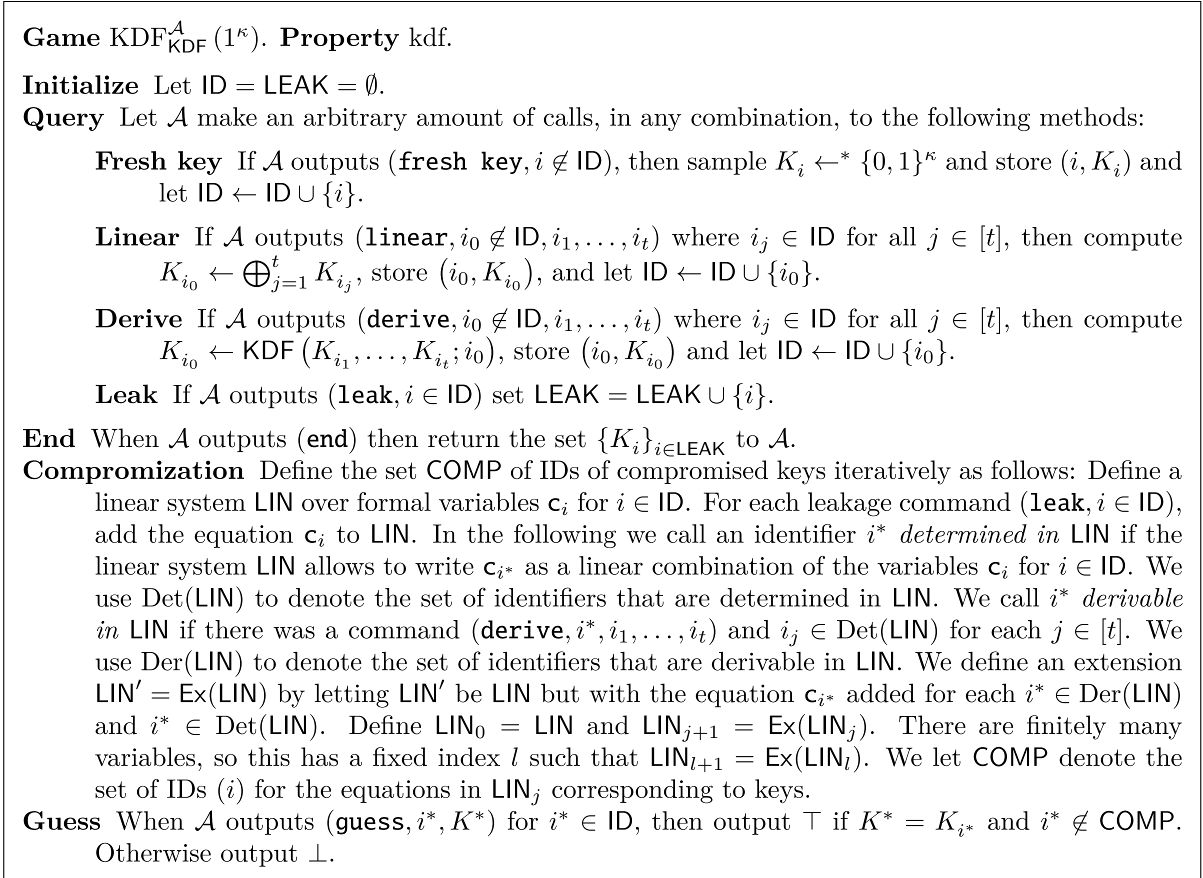
We need a notion of security where the adversary cannot compute the output of the key derivation function, except if he can do so trivially because he knows the entire input. Specifically we let keys be fresh uniformly random values, derived, or linear combinations of other keys, and T be publicly known. We require that the adversary cannot guess a key derived from at least one uniformly random key, "uncompromised" derived key, or linear combination of keys where at least one is "uncompromised". An uncompromised derived key is one that was derived from at least one uniformly random key, uncompromised derived key or linear combination where at least one key in the combination was uncompromised. We allow the adversary to compromise keys by leaking them and construct new keys through linear combinations or key derivations. Furthermore, we call a (potential) key compromised if the leaked keys allow to determine the key, in which case the adversary can trivially compute it. More precisely we define this in the game KDF in Fig. 3.1.

We let $\text{Adv}_{\mathsf{KDF}}^{\text{kdf}}(\mathcal{A}, \kappa) = \Pr\left[\text{KDF}_{\mathsf{KDF}}^{\mathcal{A}}(1^\kappa) = \top\right]$ denote the advantage of the adversary in winning the game. Using this we define the notion of a *garbling secure KDF*.

Definition 8 (Garbling Secure KDF)**.** We say that a $\mathsf{KDF}(\cdot)$ is (garbling) secure if $\text{Adv}_{\mathsf{KDF}}^{\text{kdf}}(\mathcal{A}, \kappa)$ is negligible in κ for any PPT adversary.

As we shall see the game captures the ways keys and entries of a garbled gate can be constructed: either as new, random strings (Fresh key), as the XOR combination of already existing elements (Linear), and finally using the KDF on already existing elements. We allow the adversary to ask for construction of elements using these methods, each with a unique ID. We furthermore allow him to ask to learn some of these elements (through the LEAK set).

[3] This constraint is not actually needed, but only considering a single bit output makes the overall presentation of our schemes simpler.



Game $\mathrm{KDF}^{\mathcal{A}}_{\mathsf{KDF}}(1^\kappa)$. **Property** kdf.

Initialize Let $\mathsf{ID} = \mathsf{LEAK} = \emptyset$.

Query Let $\mathcal{A}$ make an arbitrary amount of calls, in any combination, to the following methods:

- **Fresh key** If $\mathcal{A}$ outputs ($\texttt{fresh key}, i \notin \mathsf{ID}$), then sample $K_i \leftarrow^* \{0,1\}^\kappa$ and store (i, K_i) and let $\mathsf{ID} \leftarrow \mathsf{ID} \cup \{i\}$.
- **Linear** If $\mathcal{A}$ outputs $(\texttt{linear}, i_0 \notin \mathsf{ID}, i_1, \ldots, i_t)$ where $i_j \in \mathsf{ID}$ for all $j \in [t]$, then compute $K_{i_0} \leftarrow \bigoplus_{j=1}^{t} K_{i_j}$, store (i_0, K_{i_0}), and let $\mathsf{ID} \leftarrow \mathsf{ID} \cup \{i_0\}$.
- **Derive** If $\mathcal{A}$ outputs $(\texttt{derive}, i_0 \notin \mathsf{ID}, i_1, \ldots, i_t)$ where $i_j \in \mathsf{ID}$ for all $j \in [t]$, then compute $K_{i_0} \leftarrow \mathsf{KDF}(K_{i_1}, \ldots, K_{i_t}; i_0)$, store (i_0, K_{i_0}) and let $\mathsf{ID} \leftarrow \mathsf{ID} \cup \{i_0\}$.
- **Leak** If $\mathcal{A}$ outputs $(\texttt{leak}, i \in \mathsf{ID})$ set $\mathsf{LEAK} = \mathsf{LEAK} \cup \{i\}$.

End When $\mathcal{A}$ outputs ($\texttt{end}$) then return the set $\{K_i\}_{i \in \mathsf{LEAK}}$ to $\mathcal{A}$.

Compromization Define the set COMP of IDs of compromised keys iteratively as follows: Define a linear system LIN over formal variables c_i for $i \in \mathsf{ID}$. For each leakage command $(\texttt{leak}, i \in \mathsf{ID})$, add the equation c_i to LIN. In the following we call an identifier i^* *determined in* LIN if the linear system LIN allows to write c_{i^*} as a linear combination of the variables c_i for $i \in \mathsf{ID}$. We use $\mathrm{Det}(\mathsf{LIN})$ to denote the set of identifiers that are determined in LIN. We call i^* *derivable in* LIN if there was a command $(\texttt{derive}, i^*, i_1, \ldots, i_t)$ and $i_j \in \mathrm{Det}(\mathsf{LIN})$ for each $j \in [t]$. We use $\mathrm{Der}(\mathsf{LIN})$ to denote the set of identifiers that are derivable in LIN. We define an extension $\mathsf{LIN}' = \mathsf{Ex}(\mathsf{LIN})$ by letting LIN' be LIN but with the equation c_{i^*} added for each $i^* \in \mathrm{Der}(\mathsf{LIN})$ and $i^* \in \mathrm{Det}(\mathsf{LIN})$. Define $\mathsf{LIN}_0 = \mathsf{LIN}$ and $\mathsf{LIN}_{j+1} = \mathsf{Ex}(\mathsf{LIN}_j)$. There are finitely many variables, so this has a fixed index l such that $\mathsf{LIN}_{l+1} = \mathsf{Ex}(\mathsf{LIN}_l)$. We let COMP denote the set of IDs (i) for the equations in LIN_j corresponding to keys.

Guess When $\mathcal{A}$ outputs $(\texttt{guess}, i^*, K^*)$ for $i^* \in \mathsf{ID}$, then output $\top$ if $K^* = K_{i^*}$ and $i^* \notin \mathsf{COMP}$. Otherwise output $\perp$.

Figure 3.1: Security game for a KDF used in garbling

However, we demand that elements are leaked only once, when the adversary is done with all his construction queries. The game ends with the adversary choosing an ID of an "uncompromised" element and inputting his guess of what value its key is. We notice that the definition of uncompromised elements is a bit cumbersome. First of, it includes all elements leaked to the adversary; the elements with IDs in LEAK. Furthermore, it includes all elements that the adversary would be able to efficiently compute by a combination of calls to the KDF and XOR operations. A bit more formally this is captured by the variables denoted by c_i. These are iteratively defined in two steps: for the set of elements currently compromised we add the IDs of all elements which can be constructed by linear combinations of the currently compromised elements. We then add to the set, the elements which IDs can be computed using the KDF of the already compromised elements.

It can be proven using standard techniques that a (non-programmable) random oracle is a garbling secure KDF in the above sense. More precisely:

Theorem 3. *If* $\mathsf{KDF}(\cdot)$ *is modeled by a non-programmable random oracle with* κ *bits output then for any PPT* $\mathcal{A}$ *it holds that* $\mathrm{Adv}^{\mathrm{kdf}}_{\mathsf{KDF}}(\mathcal{A}, \kappa)$ *is negligible in* κ.

Proof. We do the proof by constructing several hybrid games and then show a polytime reduction between them. We then show that any PPT adversary can only win the last hybrid game with negligible probability in κ and thus conclude that this must also be the case for the real game. First consider the following hybrids:

Definition 9 (Hybrid 1)**.** Let Hybrid 1, denoted by $\mathrm{H}^{\mathcal{A},1}_{\mathsf{KDF}}(1^\kappa)$, be the game defined in the same way as game $\mathrm{KDF}^{\mathcal{A}}_{\mathsf{KDF}}(1^\kappa)$ from Fig. 3.1, except that the command **Guess** is defined as follows:

Guess

1. Sample a uniformly random bit $b \leftarrow^* \{0,1\}$.
2. If $b = 0$ let $K' \leftarrow^* \{0,1\}^\kappa$, otherwise let $K' = K_{i^*}$
3. When $\mathcal{A}$ outputs $(\texttt{guess}, i^* \in \mathsf{ID} \setminus \mathsf{COMP})$ return K' to $\mathcal{A}$.
4. When $\mathcal{A}$ outputs a bit $c \in \{0,1\}$ return $\top$ if $c = b$, otherwise return $\bot$.

Definition 10 (Hybrid 2)**.** Let Hybrid 2, denoted by $\mathrm{H}^{\mathcal{A},2}_{\mathsf{KDF}}(1^\kappa)$, be the game defined in the same way as $\mathrm{H}^{\mathcal{A},1}_{\mathsf{KDF}}(1^\kappa)$, except that the command **Derive** does not exist.

In general we let $\mathrm{H}^{\mathcal{A},i}_{\mathsf{KDF}}(1^\kappa)$ denote Hybrid i. Furthermore, we let $\mathrm{Adv}^{\mathrm{H},i}_{\mathsf{KDF}}(\mathcal{A},\kappa) = 2 \cdot \Pr\left[\mathrm{H}^{\mathcal{A},i}_{\mathsf{KDF}}(1^\kappa) = \top\right] - 1$ denote the advantage of any PPT adversary $\mathcal{A}$ playing with hybrid i.

In the rest of the proof we say a key K_i is "fresh" if it has been constructed by the call $(\texttt{fresh key}, i)$. Similarly we say that a key K_{i_0} is "derived", respectively "linear" if it has been constructed by the call $(\texttt{derive}, i_0, i_1, \ldots, i_t)$, respectively $(\texttt{linear}, i_0, i_1, \ldots, i_t)$.

Before we start on the hybrid reductions consider the following lemmas:

Lemma 2. *Let $\mathcal{A}^2$ be any PPT adversary attacking Hybrid 2. Then for any $i \in \mathsf{ID} \setminus \mathsf{COMP}$, where K_i is a fresh key, K_i is indistinguishable from an uniformly random κ bit string in the view of $\mathcal{A}^2$.*

Proof. First notice that for $i \in \mathsf{ID}$ but $i \notin \mathsf{COMP}$ where K_i has been constructed by the call $(\texttt{fresh key}, i)$, then the value K_i will be a uniformly random sampled element which has not been given to $\mathcal{A}^2$, since $\mathsf{LEAK} \subseteq \mathsf{COMP}$. Thus anything that $\mathcal{A}^2$ can learn about K_i must necessarily be based on leaked information on keys constructed by calls to **Linear** which involves K_i. This follows since any other information the adversary can learn from the game will be independent of K_i. Next notice that any leaked linearly constructed key, say K', depending on K_i is constructed as a linear combination of some keys and K_i. Then see that each of these keys are either fresh or linear keys. If they are linear keys we can simply substitute their linear expressions. This can be done recursively. Thus we get that $K' = K_i \oplus \left(\bigoplus_{j\in[t]} K_j\right)$ for some integer $t \geq 1$. Next see that it must be the case that at least one $j \notin \mathsf{COMP}$, otherwise it would be the case that $i \in \mathsf{COMP}$. For each $j \notin \mathsf{COMP}$ we have that K_j is uniformly random in the view of $\mathcal{A}^2$. Furthermore, since any value XOR'ed with a uniformly random value is uniformly random, we have that K' will be a one-time pad encryption of K_i under the key K_j (potentially XOR'ed with other compromised or uncompromised keys). This will obviously be the case for all leaked linear constructed keys depending on K_i. However, if the leaked linear keys are different from the key equivalent to the one-time encryption key will also be different, otherwise, following a similar argument as above, K_i would be compromised.

Finally, since one-time pads are perfectly secure (leak no information) when the key is unused and uniformly random, like above, then it will be impossible to use these to gain an advantage in distinguishing between K_i and another uniformly random string. □

Lemma 3. *Let $\mathcal{A}^2$ be any PPT adversary attacking Hybrid 2, then for any $i_0 \in \mathsf{ID} \setminus \mathsf{COMP}$ where K_{i_0} is constructed by the call $(\texttt{linear}, i_0, i_1, \ldots, i_t)$ and $i_1, \ldots, i_t \in \mathsf{ID}$ then K_{i_0} is indistinguishable from an uniformly random κ bit string in the view of $\mathcal{A}^2$.*

Proof. First notice that since $i_0 \notin \mathsf{COMP}$ then there must be at least one $j \in [t]$ for which $i_j \notin \mathsf{COMP}$. Next notice that we can express the key K_{i_0} as a linear combination of keys constructed by calls to **Fresh key** (by recursively expanding any key K_l constructed by a call to **Linear**). If K_l has been constructed by the call $(\texttt{fresh key}, l)$ then we have by Lemma 2 that K_l is indistinguishable from a uniformly random κ bit string for $\mathcal{A}^2$. Thus $K_{i_0} = K^{'} \oplus K_l$ where K_l is indistinguishable from a uniformly random string and $K^{'}$ is some, perhaps known, string. From this we get that K_{i_0} is indistinguishable from a uniformly random string. However, we must still show K_{i_0} remains indistinguishable from a uniformly random string no matter what following queries the adversary makes. Like in the proof of Lemma 2, the only way the adversary can gain information on K_{i_0} is through compromised linear queries depending on either K_{i_0} or the keys used in the linear construction of K_{i_0}. First consider the case of linear keys depending directly on K_{i_0}. Notice that any leaked linearly constructed key, say $K^{'}$ depending on K_{i_0} can be expressed as a linear combination of some keys and K_{i_0}. Then see that each of these keys is either a fresh key or linear key. So for any linear keys we can simply substitute their linear expressions. Thus we get that $K^{'} = K_{i_0} \oplus \left(\bigoplus_{j\in[t]} K_{i_j}\right)$ for some integer $t \geq 1$. Next see that it must be the case that at least one $j \in [t]$ such that $i_j \notin \mathsf{COMP}$, otherwise it would be the case that $i_0 \in \mathsf{COMP}$. For each $j \in [t]$ where $i_j \notin \mathsf{COMP}$ we have that K_{i_j} is uniformly random sampled. Furthermore, since any value XOR'ed with a uniformly random value is uniformly random, we have that $K^{'}$ will be a one-time pad encryption of K_{i_0} under the key K_{i_j} for some $j \in [t]$ where $i_j \notin \mathsf{COMP}$ (potentially XOR'ed with other compromised or uncompromised keys). This will obviously be the case for all leaked linearly constructed keys depending on K_{i_0}. However, if they are different from the key equivalent to the one-time pad encryption key they will also be different. If not, following a similar argument as above, K_{i_0} would be compromised. Since one-time encryptions are perfectly hiding when the key is unused and uniformly random, like above, then it will be impossible to use these to gain an advantage in distinguishing between K_{i_0} and another uniformly random string.

Now consider the case of keys used in the linear combination of K_{i_0}. Notice that if there is only one uncompromised key in the linear combination defining K_{i_0} (say K_l) then any indistinguishability advantage in such a key can be directly used in distinguishing K_{i_0} from a random string. This is so since we could write $K_{i_0} = K^{'} \oplus K_l$ for a known (or polytime computable string $K^{'}$) and thus efficiently compute the "adjusted" knowledge of K_{i_0} based on K_l. ☐

Lemma 4. *For any adversary $\mathcal{A}^2$ it holds that $\mathrm{Adv}^{\mathrm{H},2}_{\mathsf{KDF}}(\mathcal{A}, \kappa)$ is negligible in κ.*

Proof. First notice that all keys in this game have been constructed by calls to either **Fresh key** or **Linear**. So when $\mathcal{A}^2$ calls $(\texttt{guess}, i^*)$ then it must necessarily be the case that K_{i^*} is a fresh or linear key. Now since we are in Hybrid 2 we see that $\mathcal{A}^2$ cannot win the game with non-negligible probability in κ following Lemma 2 and Lemma 3, since no matter what key he tries to guess it will be indistinguishable from random in his view. ☐

Next we show a reduction between Hybrid 1 and Hybrid 2. First we notice that the only difference between Hybrid 1 and 2 is the possibility of the adversary to construct new keys with the method **Derive**.

Lemma 5. *For any PPT adversary $\mathcal{A}^1$ there exists a PPT adversary $\mathcal{A}^2$ such that*

$$\mathrm{Adv}^{\mathrm{H},1}_{\mathsf{KDF}}\left(\mathcal{A}^1, \kappa\right) \leq \mathrm{Adv}^{\mathrm{H},2}_{\mathsf{KDF}}\left(\mathcal{A}^2, \kappa\right) - \mathrm{negl}(\kappa) \ ,$$

when $\mathsf{KDF}(\cdot)$ is modeled as a non-programmable random oracle.

Proof. We fix any adversary $\mathcal{A}^1$ playing with Hybrid 1. We then construct a polytime adversary $\mathcal{A}^2$ playing with Hybrid 2, which runs $\mathcal{A}^1$ internally. We then argue that the advantage of $\mathcal{A}^2$ is at least the same as of $\mathcal{A}^1$, except with negligible difference. This means that if $\mathcal{A}^1$ can attack Hybrid 1 with non-negligible probability in κ then there exists an adversary $\mathcal{A}^2$ that can attack Hybrid 2 with non-negligible probability in κ.

We let $\mathcal{A}^2$ play the role of the challenger in Hybrid 1 against $\mathcal{A}^1$. $\mathcal{A}^2$ starts by initializing an empty map $\mathbb{D}$, such that queries of undefined elements result in $\perp$. $\mathcal{A}^2$ also initializes an empty list $\mathbb{D}'$. It then stores internally each query it gets from $\mathcal{A}^1$, and passes them on to the Hybrid 2 game. Whatever it gets back from Hybrid 2 it stores internally, and passes back to $\mathcal{A}^1$, except for the following cases:

Linear If $\mathcal{A}^1$ outputs $(\texttt{linear}, i_0 \notin \mathsf{ID}, i_1, \ldots, i_t)$ for $i_j \in \mathsf{ID}$, then update $\mathbb{D}$ such that $(i_1, \ldots, i_t) = \mathbb{D}(i_0)$, append i_0 to the end of $\mathbb{D}'$ and pass the call to Hybrid 2.

Derive If $\mathcal{A}^1$ outputs $(\texttt{derive}, i_0 \notin \mathsf{ID}, i_1, \ldots, i_t)$ for $i_j \in \mathsf{ID}$, then update $\mathbb{D}$ such that $(i_1, \ldots, i_t) = \mathbb{D}(i_0)$, append i_0 to the end of $\mathbb{D}'$ and call $(\texttt{fresh key}, i_0)$.

End When $\mathcal{A}^1$ outputs $(\texttt{end})$ initialize an empty set $\mathsf{K}' = \emptyset$. Now for each $i \in \mathsf{COMP}$ where $i \notin \mathsf{LEAK}$ we have $\mathcal{A}^2$ call $(\texttt{leak}, i)$ and internally store the response from Hybrid 2. Then $\mathcal{A}^2$ calls $(\texttt{end})$. Now let K be the set of keys returned by Hybrid 2. Next, iterate through the list $\mathbb{D}'$, starting from the beginning, let i_0 be the current entry and set $(i_1, \ldots, i_t) = \mathbb{D}(i_0)$. Now in each iteration if it holds for all $j \in [t]$, that $i_j \in \mathsf{COMP}$ and K_{i_0} is a derived key, then, using the keys in the set K, define $K'_{i_0} = \mathsf{KDF}\left(K_{i_1}, \ldots, K_{i_t}; i_0\right)$ and set $\mathsf{K}' = \mathsf{K}' \cup \{K'_{i_0}\}$. If instead K_{i_0} is a linear key, then, again using the keys in the set K, define $K'_{i_0} = \bigoplus_{j \in [t]} K_{i_j}$ and set $\mathsf{K}' = \mathsf{K}' \cup \{K'_{i_0}\}$. Furthermore, if $K_{i_0} \in \mathsf{K}$ then replace K_{i_0} with K'_{i_0} in the set K. Finally return the set K.

Guess When $\mathcal{A}^1$ outputs $(\texttt{guess}, i^*)$ pass the call on to Hybrid 2 and let K' be the output of Hybrid 2, then proceed as follows:

- If K_{i^*} is a fresh or derived key then return K'.
- If $\mathcal{A}^1$ has previously made the call $(\texttt{linear}, i^*, i_1, \ldots, i_t)$ then for each $j \in [t]$ where $i_j \in \mathsf{COMP}$ and $K'_{i_j} \in \mathsf{K}'$ set $K' := K' \oplus K_{i_j} \oplus K'_{i_j}$.[4]

Finally, when receiving a bit c from $\mathcal{A}^1$ pass on the same bit to Hybrid 2.

Some notes are due regarding the simulation above. We use $\mathbb{D}'$ as a list of IDs, in chronological order, whose associated keys are either linear or derived and thus might need to be simulated by $\mathcal{A}^2$ (as they can be based on compromised derived keys). We then let $\mathcal{A}^2$ pass on the calls to **Linear** and **Derive** to Hybrid 2, except that it adds the IDs from such calls to the list $\mathbb{D}'$. For **Derive** it furthermore simulates these as **Fresh key** calls. When **End** is finally executed we let $\mathcal{A}^2$ leak all compromised keys from Hybrid 2, as these might be needed for "adjustments". In particular we let $\mathcal{A}^2$ potentially adjust (using XOR operations or replacement) each leaked key it gets back from Hybrid 2 such that it matches what $\mathcal{A}^1$ would expect if playing with Hybrid 1. More specifically, if all keys used to construct a derived key have been compromised then we construct the derived key returned to $\mathcal{A}^1$ using the KDF, even though the key has been constructed using a call to **Fresh key** in Hybrid 2. The same goes for linearly constructed

[4]The key K_{i_j} will always be known by $\mathcal{A}^2$ after **End** as it asks Hybrid 2 to leak it.

keys using a compromised derived key in its construction: Specifically by XOR'ing out the fresh key constructed by Hybrid 2 and then XOR'ing in the derived keys, constructed using KDF by $\mathcal{A}^2$. Regarding the guess phase we let $\mathcal{A}^2$ make sure that the guess given by $\mathcal{A}^1$ is adjusted in the same manner to reflect the keys in Hybrid 2. This is in particular needed as derived keys are simulated using calls to **Fresh key** and the guess of $\mathcal{A}^1$ might be a linear key which is constructed from compromised derived keys.

We now proceed with the proof that the view of $\mathcal{A}^1$ playing with $\mathcal{A}^2$ and Hybrid 2 is computationally indistinguishable from the view of $\mathcal{A}^1$ playing with Hybrid 1.

First see that until the call to **End** the view of $\mathcal{A}^1$ is perfectly indistinguishable whether it is playing with $\mathcal{A}^2$ or Hybrid 1 since nothing is returned and the same calls are permitted. Furthermore, if $\mathcal{A}^1$ does not call **Derive** then the games will be perfectly indistinguishable, thus in the following we only consider games where $\mathcal{A}^1$ calls **Derive**. Now consider indistinguishability of the output $\mathcal{A}^1$ gets after calling **End**.

Start by noticing that for each $i \in \mathsf{LEAK}$ where $\mathcal{A}^1$ made the call $(\texttt{fresh key}, i)$ the key K_i will be perfectly indistinguishable whether $\mathcal{A}^1$ plays with $\mathcal{A}^2$ or Hybrid 1 as it will in both cases be uniformly random sampled. The same goes for each K_{i_0} with $i_0 \in \mathsf{LEAK}$ where $\mathcal{A}^1$ made the call $(\texttt{linear}, i_0, i_1, \ldots, i_t)$ and for all $j \in [t]$ it was the case that i_j was constructed by a call to $(\texttt{fresh key}, i_j)$. This is again the case since $\mathcal{A}^2$ does exactly the same as Hybrid 1. Furthermore, we can extend the case for $i_0 \in \mathsf{LEAK}$ where $\mathcal{A}^1$ made the call $(\texttt{linear}, i_0, i_1, \ldots, i_t)$ and for all $j \in [t]$ the key K_{i_j} is either a fresh key, or a linear key. This applies recursively.

Next consider indistinguishability of K_{i_0} of the first $i_0 \in \mathbb{D}'$ where $\mathcal{A}^1$ called $(\texttt{derive}, i_0, i_1, \ldots, i_t)$:

- If $i_1, \ldots, i_t \in \mathsf{COMP}$ remember that we construct $K_{i_0} \leftarrow \mathsf{KDF}\left(K_{i_1}, \ldots, K_{i_t}; i_0\right)$ and thus K_{i_0} is constructed exactly the same as in Hybrid 1. This construction is possible since $\mathcal{A}^1$ will know the values $K_{i_1}, \ldots, K_{i_t}$ and thus can query the oracle himself.
- If $\exists j \in [t]$ s.t. $i_j \notin \mathsf{COMP}$ we simulate key K_{i_0} by asking Hybrid 2 to construct and leak a fresh key, i.e. a uniformly random key. Thus we must argue that a key sampled uniformly random is indistinguishable from a derived key when at least one of the keys used to construct the derived key is not compromised. To see this consider the following cases for an uncompromised key K_{i_j}:

  1. K_{i_j} was constructed by a call to **Fresh key**.
  2. K_{i_j} was constructed by a call to **Linear**. Assume w.l.o.g. that all keys used in the linear combination were constructed either by a call to **Fresh key** (if not we can repeatedly expand the keys in the linear combination defining K_{i_j} to get a single linear combination of keys made with **Fresh key**).

  Now in the first case, K_{i_j} is indistinguishable from a uniformly random element following the proof of Lemma 2 and the observation that $\mathcal{A}^1$ at this point has not learned anything based on the calls to **Derive** where K_{i_j} has been used. The second case follows from the proof of Lemma 3 and again the observation that $\mathcal{A}^1$ at this point has not learned anything based on the calls to **Derive** where K_{i_j} has been used. Thus K_{i_j} is uniformly random in the view of $\mathcal{A}^1$. This means that for $\mathcal{A}^1$ to distinguish between a uniformly random sampled key and $\mathsf{KDF}\left(K_{i_1}, \ldots, K_{i_t}; i_0\right)$ he must query exactly $\mathsf{KDF}\left(K_{i_1}, \ldots, K_{i_t}; i_0\right)$ to get an advantage, since $\mathsf{KDF}(\cdot)$ is a random oracle. However, K_{i_j} is κ uniformly random bits in his view and he is bounded by polynomial time in κ, thus his advantage can at

most be $\text{poly}(\kappa)/2^\kappa$, which is negligible in κ. Thus the view induced on $\mathcal{A}^1$ by $\mathcal{A}^2$ is computationally indistinguishable from the view of $\mathcal{A}^1$ when playing with Hybrid 1.

Next we must argue that the view remains indistinguishable for the rest of the derived keys. Following the argument above, this remains true for each uncompromised derived key when the keys used in the derivation have, perhaps recursively, been constructed by calls to **Fresh key**. Thus the remaining case we must argue is when at least one uncompromised derived key has been used in the construction, either directly or as part of a linear combination: First notice that we just showed that the first uncompromised derived key is computationally indistinguishable from a uniformly random sampled key in the view of $\mathcal{A}^1$, thus the argument above goes through if such a key is used instead of a fresh key. The same remains true for linearly constructed keys consisting of at least one uncompromised derived key: Since the uncompromised derived key is computationally indistinguishable from a random key, a linear key where it is replaced with a fresh key will remain computationally indistinguishable from a uniformly random key.

Now, see that the set of the compromised keys returned to $\mathcal{A}^1$ will be indistinguishable in both the game played with $\mathcal{A}^2$ and Hybrid 1 as they will be sampled in exactly the same way (by the fact that $\mathcal{A}^2$ does appropriate adjustments). Then see that since $\mathcal{A}^1$ is bounded by $\text{poly}(\kappa)$ we can at most have $\text{poly}(\kappa)$ keys, each computationally indistinguishable from a uniformly random element, and thus the distinguishability advantage of the total view will be bounded by $\text{poly}(\kappa) \cdot \text{negl}(\kappa) = \text{negl}(\kappa)$.

Finally consider the view in regards to the guess by $\mathcal{A}^1$. By the previous arguments if $\mathcal{A}^1$ outputs i^* where K_{i^*} is a fresh or derived key then it will be computationally indistinguishable from a uniformly random string and thus the advantage of $\mathcal{A}^1$ will be the same as the advantage of $\mathcal{A}^2$. If instead K_{i^*} was constructed by the call $(\texttt{linear}, i^*, i_1, \ldots, i_t)$ then before $\mathcal{A}^2$ passes the key it gets from Hybrid 2 back to $\mathcal{A}^1$ it adjusts it according to the compromised derived keys it might depend on. Thus the key returned to $\mathcal{A}^1$ will be computationally indistinguishable whether it comes from $\mathcal{A}^2$ or Hybrid 1, assuming each uncompromised derived key is indistinguishable from a random string, which is exactly the case (as we showed previously). This in turn implies the advantage of $\mathcal{A}^2$ is the same as $\mathcal{A}^1$, except with negligible difference, as $\mathcal{A}^2$ inputs the same bit to the game as it received by $\mathcal{A}^1$.

Since the views are at most negligibly distinguishable the advantage of $\mathcal{A}^2$ must be the same as $\mathcal{A}^1$ with at most negligible difference. □

Lemma 6. *For any PPT adversary $\mathcal{A}$ that wins the game $\text{KDF}^{\mathcal{A}}_{\mathsf{KDF}}(1^\kappa)$ with non-negligible advantage there exists a PPT adversary $\mathcal{A}^1$ that has non-negligible advantage in winning the $\text{H}^{\mathcal{A},1}_{\mathsf{KDF}}(1^\kappa)$ game when $\mathsf{KDF}(\cdot)$ is modeled as a non-programmable random oracle.*

Proof. We fix any adversary $\mathcal{A}$ attacking the KDF game. We then construct a polytime adversary $\mathcal{A}^1$ attacking Hybrid 1, which runs $\mathcal{A}$ internally and argue that the advantage of $\mathcal{A}^1$ is at least the same as that of $\mathcal{A}$, except with negligible difference in κ. We let $\mathcal{A}^1$ play the role of the challenger in the KDF game against $\mathcal{A}$. $\mathcal{A}^1$ passes on each query it gets from $\mathcal{A}$ to the Hybrid 1 game, stores it internally, and passes back to $\mathcal{A}$, the result it gets from Hybrid 1, except for the case of **End**: When $\mathcal{A}$ outputs (end, K^*) call (end) on Hybrid 1. Let K' be the key returned from Hybrid 1. If $K^* = K'$ then input $c = 1$ to Hybrid 1, otherwise input $c = 0$.

First notice that the view of $\mathcal{A}$ will be perfectly indistinguishable whether it is playing with $\mathcal{A}^1$ or the KDF game since the calls and values returned to $\mathcal{A}$ in both cases are constructed similarly. Thus we only need to argue that the advantage of $\mathcal{A}^1$ is the same as $\mathcal{A}$ playing with the KDF game except with negligible difference in κ.

See that if the bit b chosen by Hybrid 1 is 1 the winning probability of $\mathcal{A}^1$ and $\mathcal{A}$ is the same. To see this notice that if $K^* = K^{'}$ then both $\mathcal{A}$ and $\mathcal{A}^1$ will win and if $K^* \neq K^{'}$ then both $\mathcal{A}$ and $\mathcal{A}^1$ will lose. If instead $b = 0$ and $K^* = K^{'}$ then $\mathcal{A}$ will win and $\mathcal{A}^1$ will loose. However, since $K^{'}$ in this case is uniformly random sampled $K^* = K^{'}$ will only occur with probability $2^{-\kappa}$. If instead $K^* \neq K^{'}$ (happening with probability $1 - 2^{-\kappa}$) then $\mathcal{A}$ will lose and $\mathcal{A}^1$ will win. This will make $\mathcal{A}^1$ win with probability $1 - 2^{-\kappa}$ if $b = 0$ and probability $\text{Adv}^{\text{kdf}}_{\mathsf{KDF}}(\mathcal{A}, \kappa)$ if $b = 1$. Since b is uniformly random sampled we get that $\mathcal{A}^1$ wins the game with probability $\frac{1}{2} \cdot \left(1 - 2^{-\kappa} + \text{Adv}^{\text{kdf}}_{\mathsf{KDF}}(\mathcal{A}, \kappa)\right)$. Thus giving $\mathcal{A}^1$ an advantage $\frac{1}{2}\text{Adv}^{\text{kdf}}_{\mathsf{KDF}}(\mathcal{A}, \kappa) - 2^{-\kappa-1}$. We see that if $\text{Adv}^{\text{kdf}}_{\mathsf{KDF}}(\mathcal{A}, \kappa)$ is non-negligible so is $\frac{1}{2}\text{Adv}^{\text{kdf}}_{\mathsf{KDF}}(\mathcal{A}, \kappa)$. Now, since $2^{-\kappa-1}$ is negligible in κ we have that $\text{Adv}^{\text{kdf}}_{\mathsf{KDF}}(\mathcal{A}, \kappa)$ is non-negligible so is $\frac{1}{2}\text{Adv}^{\text{kdf}}_{\mathsf{KDF}}(\mathcal{A}, \kappa) - 2^{-\kappa-1}$. $\square$

Now we have from Lemma 4 that $\text{Adv}^{\text{H},2}_{\mathsf{KDF}}(\mathcal{A}^2, \kappa) \leq \text{negl}(\kappa)$ for any PPT adversary $\mathcal{A}^2$. Thus, using Lemma 5, we see that there cannot exist a PPT adversary $\mathcal{A}^1$ winning the game $\text{H}^{\mathcal{A}^1,1}_{\mathsf{KDF}}(1^\kappa)$ with non-negligible advantage, as this would imply the existence of an adversary $\mathcal{A}^2$ winning $\text{H}^{\mathcal{A}^2,2}_{\mathsf{KDF}}(1^\kappa)$ with non-negligible advantage, contradicting Lemma 4. Now, using Lemma 6 we see that there cannot exist a PPT adversary $\mathcal{A}$ winning the game $\text{KDF}^{\mathcal{A}}_{\mathsf{KDF}}(1^\kappa)$ with non-negligible advantage, as this would imply the existence of an adversary $\mathcal{A}^1$ winning $\text{H}^{\mathcal{A}^1,1}_{\mathsf{KDF}}(1^\kappa)$ with non-negligible advantage, contradicting our discussion above.

Combining the steps above we see that there cannot exist a PPT adversary $\mathcal{A}$ winning $\text{KDF}^{\mathcal{A}}_{\mathsf{KDF}}(1^\kappa)$ with non-negligible advantage. This concludes the proof. $\square$

We leave as future work the investigation of which exact computational assumptions are required for implementing our different garbling schemes: while it is clear that the free-XOR and fleXOR variants require a strong notion of security (security under related-key attack and a flavor of circular security), it seems that the GRR1 variant could be instantiated using standard security notions.

3.2 Our Privacy-Free Garbling Schemes

In this section we present our novel garbling schemes. Our schemes support gates with arbitrary fan-in, but as a warm-up we first present the garbling schemes for gates with fan-in 2 using GRR1 or GRR2 with free-XOR. Both allow to garble every Boolean gate with fan-in 2 using only 3 calls to the KDF for non-XOR gates and require no calls to the KDF for XOR gates.

Our first scheme has communication complexity of κ bits per gate while our second garbling scheme is compatible with "free-XOR", but requires communication complexity of 2κ bits for non-XOR gates. Afterwards we present our two schemes for gates with arbitrary fan-in.

Warm-up

To simplify notation and give the intuition of our scheme we here only describe how to garble/evaluate a single NAND or XOR gate while abstracting away the salt used by the KDF. We call the input keys to the left wire of a gate L^0, L^1, the input keys to the right wire R^0, R^1, and the output keys O^0, O^1. All these values are elements of $\{0,1\}^\kappa$.

Again we point out that in contrast with general garbled circuits, in our case if the circuit evaluator has two keys L^a, R^b, he knows the corresponding bits a and b.

First consider a NAND gate with GRR1:

Garbling a GRR1 NAND Gate: Let $O^0 = \mathsf{KDF}(L^1, R^1)$ and $O^1 = \mathsf{KDF}(L^0)$. Compute $P = \mathsf{KDF}(R^0) \oplus O^1$ and output P.

Evaluating a GRR1 NAND Gate: To evaluate on input L^a, R^b, if $a = b = 1$ then output $O^0 = \mathsf{KDF}(L^1, R^1)$ otherwise, if $a = 0$ compute $O^1 = \mathsf{KDF}(L^0)$. Otherwise, if $b = 0$ compute $O^1 = P \oplus \mathsf{KDF}(R^0)$.

It should be clear that the scheme is correct. The intuition of authenticity is that if the evaluator only knows one input key for each wire, he can only learn one output key unless he can guess the output of KDF on an input he does not know. Next consider a XOR gate:

Garbling a GRR1 XOR Gate: Let $O^0 = L^0 \oplus R^0$ along with $O^1 = L^0 \oplus R^1$. Output $P = L^0 \oplus L^1 \oplus R^0 \oplus R^1$.

Evaluating a GRR1 XOR Gate: On input L^a, R^b if $a = 0$ then output $O^{a \oplus b} = L^a \oplus R^b$. Otherwise compute and return $O^{a \oplus b} = P \oplus L^a \oplus R^b$.

Again, it should be clear that the scheme is correct. The authenticity intuitively follows from the fact that the evaluator can only learn the XOR of two unknown keys which will not help him decrypting the next gate.

Now consider how to achieve the same, while allowing support for free-XOR gates (and in turn GRR2). In this scheme there is a global difference Δ s.t., for any wire i in a garbled circuit, the key pair X_i^0, X_i^1 satisfies $X_i^0 \oplus X_i^1 = \Delta$.

Garbling a GRR2 NAND Gate: Let $O^0 = \mathsf{KDF}(L^1, R^1)$. This defines $O^1 = O^0 \oplus \Delta$ as well. Let $P[0] = \mathsf{KDF}(L^0) \oplus O^1$ and $P[1] = \mathsf{KDF}(R^0) \oplus O^1$. Finally output P.

Evaluating a GRR2 NAND Gate: To evaluate on input L^a, R^b, if $a = b = 1$ then output $O^0 = \mathsf{KDF}(L^1, R^1)$ otherwise, if $a = 0$ output $O^1 = \mathsf{KDF}(L^0) \oplus P[0]$ otherwise output $O^1 = \mathsf{KDF}(R^0) \oplus P[1]$.

Next consider a XOR gate:

Garbling a free-XOR Gate: Let $O^0 = L^0 \oplus R^0$. This defines $O^1 = O^0 \oplus \Delta$ as well. Output nothing.

Evaluating a free-XOR Gate: On input L^a, R^b, output $O^{a \oplus b} = L^a \oplus R^b$.

Again correctness should be clear and authenticity follows from the security of free-XOR, i.e. that it is hard to learn Δ, unless one is given both keys on some wire.

Generalization Intuition

We now consider how our approaches generalizes to gates with arbitrary fan-in.

NAND gates. Consider a NAND gate with fan-in t, call this gate g. Recall that for this gate the output bit $b_g = 0$ should occur exactly if all the input bits are equal to 1, $b_1 = b_2 = \ldots = b_t = 1$. This means that we can define the output key representing bit 0 directly from these: If we denote the key on input wire i by $K_i^{b_i}$, then the output 0-key is computed as

$$K_g^0 = \mathsf{KDF}\left(K_1^1, K_2^1, \ldots, K_t^1\right) .$$

		Garb.	Eval.	Size
GRR1	NAND	$t+1$	1	$\kappa(t-1)$
	XOR	0	0	$\kappa(t-1)$
Free-XOR	NAND	$t+1$	1	κt
	XOR	0	0	0
FleXOR	NAND	$t+1$	1	$\kappa(t-1)$
	d-XOR	0	0	κd

Table 3.1: Exact performances of our privacy-free garbling scheme. The “Garb.” and “Eval.” columns state the number of calls to a KDF required for garbling and evaluation respectively, as a function of the gate fan-in t. The letter d denotes how many ciphertexts are needed for a given fleXOR gate. The column “Size” states the number of bits added to the garbled circuit for each gate. We only report the fleXOR variant based on “Safe” wire ordering, see [KMR14] for details.

Now, if we are not using a free-XOR scheme we define the output 1-key to be $K_g^1 = \mathsf{KDF}\left(K_1^0\right)$. Then the garbled computation table is defined as follows:

$$P_g = \left\{P_g[i] = K_g^1 \oplus \mathsf{KDF}\left(K_i^0\right)\right\}_{i=2}^{t} .$$

When we are using a free-XOR scheme we have another entry in the garbled computation table since the output key K_g^1 needs to meet the constraint $K_g^1 = K_g^0 \oplus \Delta$ and thus we cannot simply define it to be $\mathsf{KDF}\left(K_1^0\right)$. However, similarly to the scheme above that does not use free-XOR we use the KDF applied to the first input key (which we have not used to hide anything in the scheme above) to hide K_g^1. We let the rest of the table remain as before and thus the whole garbled computation table is computed as follows:

$$P_g = \left\{P_g[i] = K_g^1 \oplus \mathsf{KDF}\left(K_i^0\right)\right\}_{i=1}^{t} .$$

Now consider the evaluation: Call the input keys $K_1^{b_1}, K_2^{b_2}, \ldots, K_t^{b_t}$. If $b_i = 1$ for all $i \in [t]$ then the output is $K_g^0 = \mathsf{KDF}\left(K_1^{b_1}, K_2^{b_2}, \ldots, K_t^{b_t}\right)$. Otherwise find the first value of i for which $b_i \neq 1$ and output $K_g^1 = P_g[i] \oplus \mathsf{KDF}\left(K_i^{b_i}\right)$, except if $i = 1$ and we do not use a free-XOR garbling scheme, in which case the output is $K_g^1 = \mathsf{KDF}\left(K_1^{b_1}\right)$.

XOR gates. To garble XOR gates (when we are not using the free-XOR method), we define the output 0-key from information based on all the input 0-keys. Specifically as

$$K_g^0 = K_1^0 \oplus K_2^0 \oplus \cdots \oplus K_t^0 = \bigoplus_{i=1}^{t} K_i^0 .$$

In a similar manner we define the output 1-key from information based on the first input 1-key and all the other input 0-keys, that is

$$K_g^1 = K_1^1 \oplus K_2^0 \oplus \cdots \oplus K_t^0 = K_1^1 \oplus \left(\bigoplus_{i=2}^{t} K_i^0\right) .$$

Let b_i for $i \in [t]$ be the input bits at evaluation time and $b_1 \oplus \ldots \oplus b_t$ be the output of that gate. It might be the case that $b_1 \neq 1$ or that there is another $j \in [t]$ s.t., $b_j = 1$. So we let the garbled computation table consist of information which makes it possible for the evaluator to compute the right output key in any such situation. Specifically we define the table as the following set:

$$P_g = \left\{ P_g[i] = K_i^0 \oplus K_i^1 \oplus K_1^0 \oplus K_1^1 \right\}_{i=2}^{t} .$$

It is clear that, for any $j \in [2;t]$

$$\left(\bigoplus_{i \in [t]} K_i^{b_i} \right) \oplus P_g[j] = K_j^{b_j \oplus 1} \oplus K_1^{b_1 \oplus 1} \oplus \bigoplus_{i \in [2;t]: i \neq j} K_i^{b_i} .$$

Thus by XOR'ing all the $P_g[i]$'s for which $b_i = 1$ with the XOR of all input keys we can learn $K_g^{b_g}$. A bit more formally:

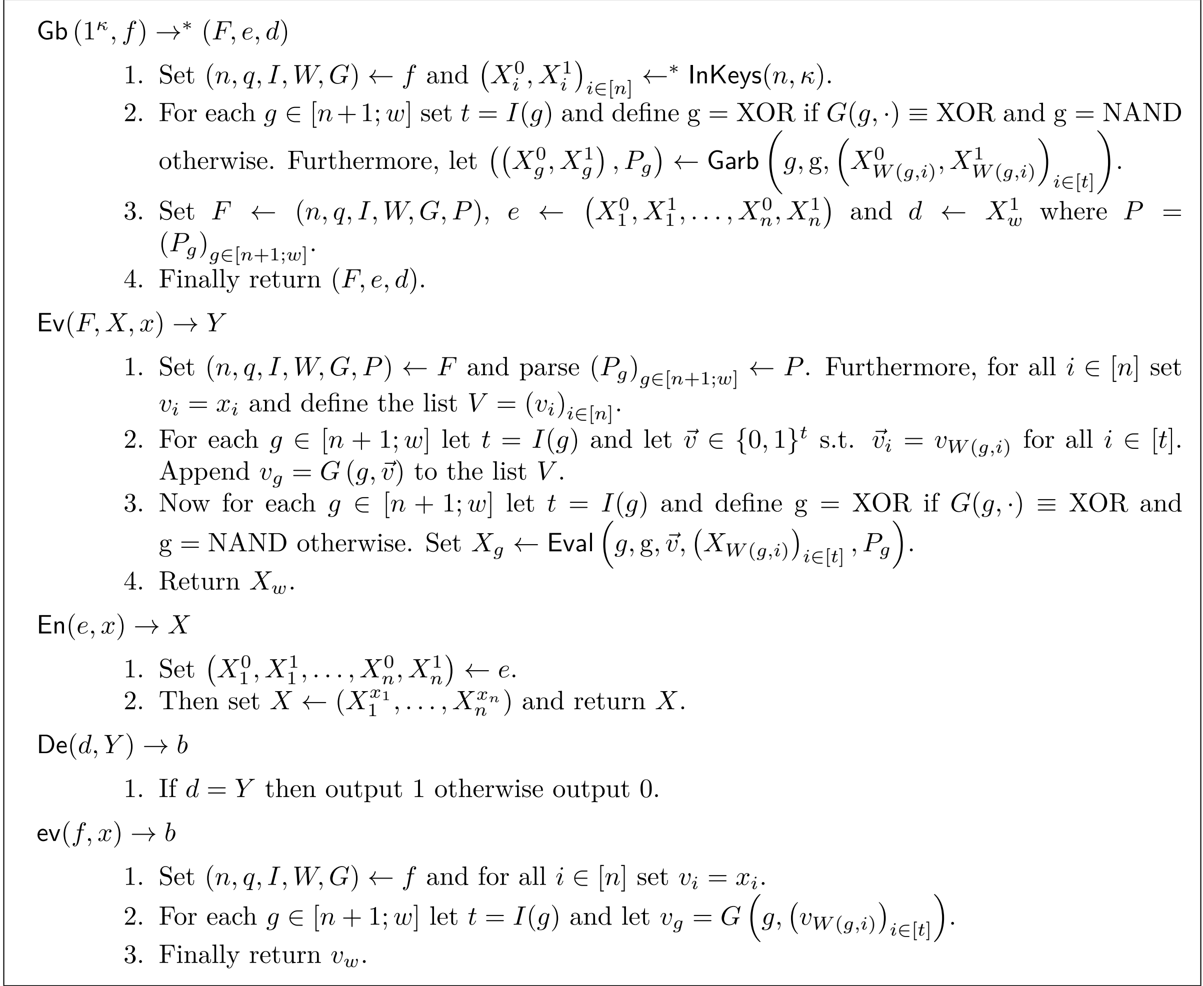
$$\left(\bigoplus_{i \in [t]} K_i^{b_i} \right) \oplus \left(\bigoplus_{i \in [2;t]: b_i = 1} P_g[i] \right) = K_1^{b_1 \oplus \ldots \oplus b_t} \oplus \left(\bigoplus_{i \in [2;t]: b_i = 0} K_i^0 \right) \oplus \left(\bigoplus_{i \in [2;t]: b_i = 1} K_i^{1 \oplus 1} \right) = K_g^{b_g} .$$

Other gates. It is easy to see that our garbling schemes can also be applied to other kind of gates such as AND, (N)OR, NXOR etc. This is true both for fan-in 2 gates, but also in the case of high fan-in (by using a different partitioning of the inputs and relabeling the outputs). However, our garbling schemes cannot be used for generic "unstructured" gates of high fan-in.

Using high fan-in gates. Note that our garbling scheme is favorable for gates with high fan-in since the complexity, shown in Table 3.1 (both in terms of communication and computational complexity), only grows linearly with the gate fan-in, while a straightforward use of general garbled circuit leads to an exponential blow-up in the gate fan-in. Even when comparing the garbling of a gate with fan-in t to a circuit implementing the same functionality (e.g., a tree of fan-in 2 AND gates with a NAND gate at the output layer to implement a NAND gate with fan-in t) our scheme is still favorable. Depending on the garbling scheme we can save a factor 2-3 in terms of computation for the garbler and also save in communication. In addition, the evaluator has an overhead of $\log(t)$ when evaluating such a sub-circuit (versus a single call to the KDF in our case).

Formal Specification

We describe our gate garbling schemes in the same notation as [BHR12b], but with some changes in order to reflect that we only require privacy, only assume one bit output, and that we support gates of arbitrary fan-in. The specification of the garbling scheme is given in Fig. 3.2. The realizations for individual gate garbling is given in Fig. 3.3 and Fig. 3.4, depending on whether or not one uses free-XOR or GRR1. We note that all of these are very similar to the methods for the scheme GaXR in Fig. 2.8, but supporting arbitrary fan-in, authenticity, only one bit output and where the gate garbling, gate evaluation, and input key sampling instructions are factored out into separate methods for convenience of presentation.



$\mathsf{Gb}(1^\kappa, f) \to^* (F, e, d)$

1. Set $(n, q, I, W, G) \leftarrow f$ and $\left(X_i^0, X_i^1\right)_{i\in[n]} \leftarrow^* \mathsf{InKeys}(n, \kappa)$.
2. For each $g \in [n+1; w]$ set $t = I(g)$ and define $\mathrm{g} = \mathrm{XOR}$ if $G(g, \cdot) \equiv \mathrm{XOR}$ and $\mathrm{g} = \mathrm{NAND}$ otherwise. Furthermore, let $\left(\left(X_g^0, X_g^1\right), P_g\right) \leftarrow \mathsf{Garb}\left(g, \mathrm{g}, \left(X^0_{W(g,i)}, X^1_{W(g,i)}\right)_{i\in[t]}\right)$.
3. Set $F \leftarrow (n, q, I, W, G, P)$, $e \leftarrow \left(X_1^0, X_1^1, \ldots, X_n^0, X_n^1\right)$ and $d \leftarrow X_w^1$ where $P = (P_g)_{g\in[n+1;w]}$.
4. Finally return (F, e, d).

$\mathsf{Ev}(F, X, x) \to Y$

1. Set $(n, q, I, W, G, P) \leftarrow F$ and parse $(P_g)_{g\in[n+1;w]} \leftarrow P$. Furthermore, for all $i \in [n]$ set $v_i = x_i$ and define the list $V = (v_i)_{i\in[n]}$.
2. For each $g \in [n+1; w]$ let $t = I(g)$ and let $\vec{v} \in \{0,1\}^t$ s.t. $\vec{v}_i = v_{W(g,i)}$ for all $i \in [t]$. Append $v_g = G(g, \vec{v})$ to the list V.
3. Now for each $g \in [n+1; w]$ let $t = I(g)$ and define $\mathrm{g} = \mathrm{XOR}$ if $G(g, \cdot) \equiv \mathrm{XOR}$ and $\mathrm{g} = \mathrm{NAND}$ otherwise. Set $X_g \leftarrow \mathsf{Eval}\left(g, \mathrm{g}, \vec{v}, \left(X_{W(g,i)}\right)_{i\in[t]}, P_g\right)$.
4. Return X_w.

$\mathsf{En}(e, x) \to X$

1. Set $\left(X_1^0, X_1^1, \ldots, X_n^0, X_n^1\right) \leftarrow e$.
2. Then set $X \leftarrow (X_1^{x_1}, \ldots, X_n^{x_n})$ and return X.

$\mathsf{De}(d, Y) \to b$

1. If $d = Y$ then output 1 otherwise output 0.

$\mathsf{ev}(f, x) \to b$

1. Set $(n, q, I, W, G) \leftarrow f$ and for all $i \in [n]$ set $v_i = x_i$.
2. For each $g \in [n+1; w]$ let $t = I(g)$ and let $v_g = G\left(g, \left(v_{W(g,i)}\right)_{i\in[t]}\right)$.
3. Finally return v_w.

Figure 3.2: Privacy-Free Garbling

The garbling scheme. The first method in Fig. 3.2, Gb, constructs a garbled circuit, F, along with encoding information, e, to encode a binary string as garbled input to this garbled circuit, and decoding information, d, to check if the output of an evaluation of the garbled circuit has the semantic value 1. In step 1 the algorithm chooses two keys for each of the n input bits to f in accordance with the specific type of garbling scheme used. These are the 0-, respectively, 1-input keys. Step 2 involves iteratively constructing each of the q garbled gates of the circuit, along with the two output keys needed for each of these gates. It is done by first using I to decide the fan-in of a given gate, then using G to find the specific functionality of the given gate. Finally the input keys for that gate (which have already been constructed) are loaded using W and all the information is passed to the gate garbling method Garb. In step 3 the garbled circuit, F, is set to include all the information of f along with the garbled computation table returned by Garb in the previous step for all the gates in the circuit. These tables are called P. Furthermore, the encoding information e is set to be the two keys for each input wire and the decoding information d is set to be the output 1-key of the final gate in the circuit. In the last step the garbled circuit F, the input encoding information, e, and decoding information, d, is returned.

The second method in Fig. 3.2, Ev, evaluates a garbled circuit, F, and returns the output key

of the final gate as a result of this evaluation, Y. In step 1 the method parses the information stored in the garbled circuit F and defines a list of bits, V, which represents the bits on each wire in the garbled circuit. Initially this list only includes the bits of the input wires. Step 2 iteratively evaluates the garbled circuit one gate at a time. It first finds the fan-in of a given gate using I and then evaluates the gate in plain using the list V along with the gate description G. After evaluating the gate in plain it updates V to contain the output bit of the given gate. Thus at the end V contains the expected bit on each wire given the garbled circuit F and the binary input x. In step 3 the method proceeds to evaluate each garbled gate iteratively. Again it uses I to learn the fan-in for a given gate, it uses G to decode the specific functionality of the gate and the elements of V to find the semantic meaning of the keys supposed to be input to the garbled gate. Using this information, along with the garbled computation table of the gate, P, it calls Eval to evaluate the garbled gate and stores the output key which the method returns. Finally in step 4 it returns the output key of the final gate in the garbled circuit.

The third method in Fig. 3.2, En, constructs a list of input keys to a garbled circuit, X. In the first step the method parses e as n ordered pairs of keys. In step 2 the functionality returns an ordered subset of the keys. In particular if the i'th bit of x is 0 then the i'th element in the list is the i'th 0-key, otherwise it is the i'th 1-key.

The fourth method in Fig. 3.2, De, evaluates whether some value, Y, is equal to the output 1-key of a garbled circuit, d. The method only has one step which checks if $d = Y$ and returns 1 if that is true, otherwise it returns 0.

The last method in Fig. 3.2, ev, evaluates the Boolean functionality f in plain using a binary input vector x. In Step 1 it parses the functionality f and finds the bits supposed to be on each wire in the circuit when evaluated on input x. It does so iteratively in step 2, starting with the bits on the circuit input wires, x. It proceeds by first learning the fan-in of the given gate using I and then using G with the given gate index and bits already found. It updates the list V with the result. Finally it returns the result of evaluating the final gate in the circuit.

Gate garbling. All of our garbling schemes have two methods: Garb and Eval. The first constructs a garbled gate, P_g, and two keys, $\left(X_g^0, X_g^1\right)$. It takes as input a gate ID, g, a function mapping a binary vector to a bit, g, along with a pair of input keys for each input wire to the gate. The second method reconstructs a single output key. It takes as input a gate ID, g, a function mapping a binary vector to a bit, g, a binary vector describing the bits on the input wires to the gate, $\vec{v}$, a list of input keys $(X_i)_{i\in[t]}$ along with a list which is the garbled computation table P_g.[5] Two concrete schemes are shown in Fig. 3.3 and Fig. 3.4.

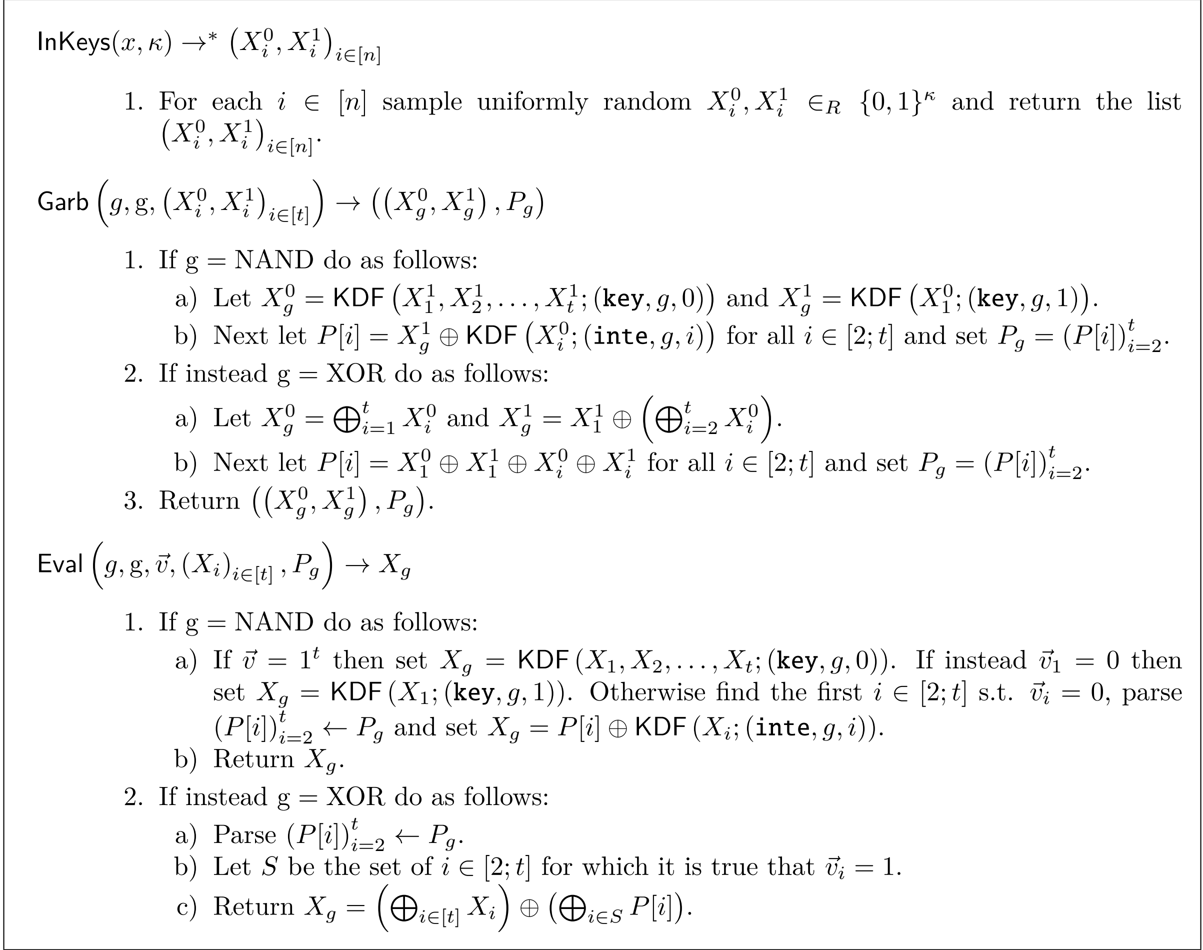
Security

The scheme presented in Fig. 3.2 composed with Fig. 3.3 and Fig. 3.4 respectively are clearly correct. In fact, any correctly garbled circuit evaluates to the correct output key with probability 1.

In regards to verifiability we define the extractor needed for the verifiability demand in the case of the GRR1 scheme (the scheme in Fig. 3.2 composed with Fig. 3.3) as follows:

$\mathrm{Ext}(r, f, y = 1) \to Y'$**:**

[5] Note that, as it is described, the running time of Eval depends on the particular input used. To prevent leakage of the input based on timing attacks, any implementation of Eval would need to take appropriate countermeasures, and ensure that the running time does not depend on the input used.



$\mathsf{InKeys}(x, \kappa) \rightarrow^* \left(X_i^0, X_i^1\right)_{i \in [n]}$

1. For each $i \in [n]$ sample uniformly random $X_i^0, X_i^1 \in_R \{0,1\}^\kappa$ and return the list $\left(X_i^0, X_i^1\right)_{i \in [n]}$.

$\mathsf{Garb}\left(g, \mathrm{g}, \left(X_i^0, X_i^1\right)_{i \in [t]}\right) \rightarrow \left(\left(X_g^0, X_g^1\right), P_g\right)$

1. If $\mathrm{g} = \mathrm{NAND}$ do as follows:
   a) Let $X_g^0 = \mathsf{KDF}\left(X_1^1, X_2^1, \ldots, X_t^1; (\mathtt{key}, g, 0)\right)$ and $X_g^1 = \mathsf{KDF}\left(X_1^0; (\mathtt{key}, g, 1)\right)$.
   b) Next let $P[i] = X_g^1 \oplus \mathsf{KDF}\left(X_i^0; (\mathtt{inte}, g, i)\right)$ for all $i \in [2; t]$ and set $P_g = (P[i])_{i=2}^t$.
2. If instead $\mathrm{g} = \mathrm{XOR}$ do as follows:
   a) Let $X_g^0 = \bigoplus_{i=1}^t X_i^0$ and $X_g^1 = X_1^1 \oplus \left(\bigoplus_{i=2}^t X_i^0\right)$.
   b) Next let $P[i] = X_1^0 \oplus X_1^1 \oplus X_i^0 \oplus X_i^1$ for all $i \in [2; t]$ and set $P_g = (P[i])_{i=2}^t$.
3. Return $\left(\left(X_g^0, X_g^1\right), P_g\right)$.

$\mathsf{Eval}\left(g, \mathrm{g}, \vec{v}, (X_i)_{i \in [t]}, P_g\right) \rightarrow X_g$

1. If $\mathrm{g} = \mathrm{NAND}$ do as follows:
   a) If $\vec{v} = 1^t$ then set $X_g = \mathsf{KDF}\left(X_1, X_2, \ldots, X_t; (\mathtt{key}, g, 0)\right)$. If instead $\vec{v}_1 = 0$ then set $X_g = \mathsf{KDF}\left(X_1; (\mathtt{key}, g, 1)\right)$. Otherwise find the first $i \in [2; t]$ s.t. $\vec{v}_i = 0$, parse $(P[i])_{i=2}^t \leftarrow P_g$ and set $X_g = P[i] \oplus \mathsf{KDF}\left(X_i; (\mathtt{inte}, g, i)\right)$.
   b) Return X_g.
2. If instead $\mathrm{g} = \mathrm{XOR}$ do as follows:
   a) Parse $(P[i])_{i=2}^t \leftarrow P_g$.
   b) Let S be the set of $i \in [2; t]$ for which it is true that $\vec{v}_i = 1$.
   c) Return $X_g = \left(\bigoplus_{i \in [t]} X_i\right) \oplus \left(\bigoplus_{i \in S} P[i]\right)$.

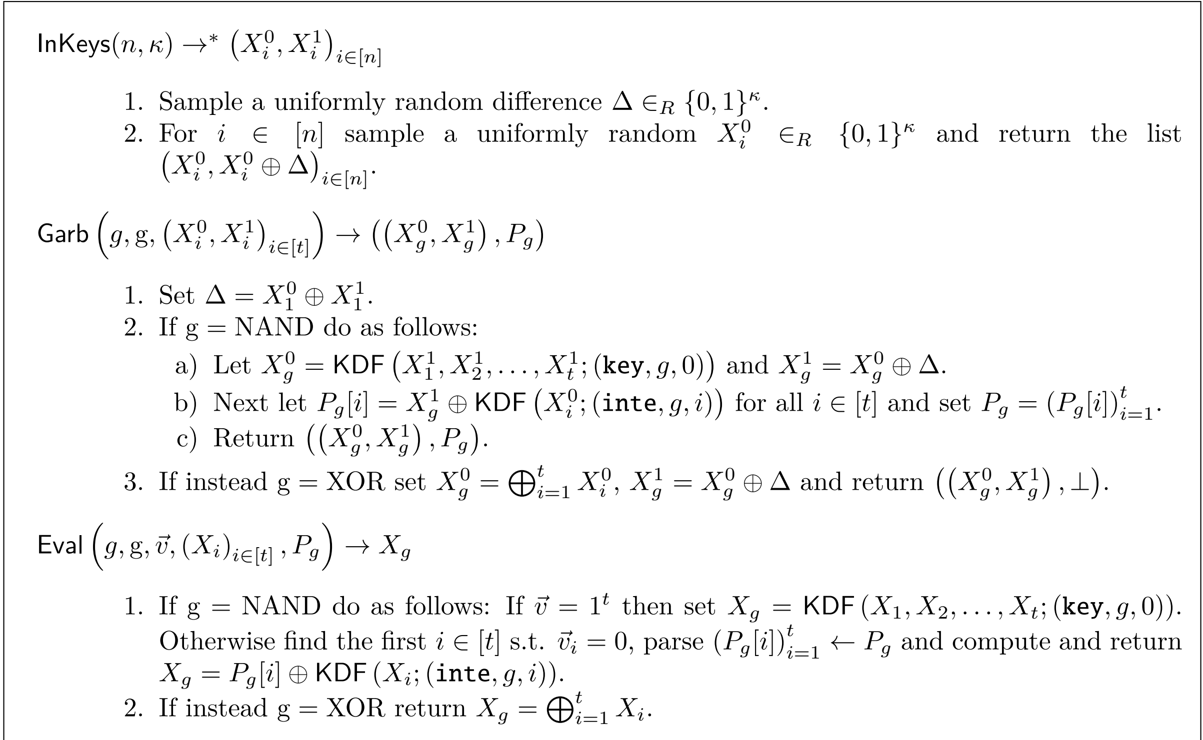
Figure 3.3: Garbling GRR1 - Without free-XOR

1. Compute $(F, e, d) \leftarrow \mathsf{Gb}\left(1^\kappa, f; r\right)$.
2. Parse $(n, q, I, W, G) \leftarrow f$, $(n, q, I, W, G, P) \leftarrow F$, and $\left(X_1^0, X_1^1, \ldots, X_n^0, X_n^1\right) \leftarrow e$.
3. For each $g \in [n+1; w]$ let $t = I(g)$ and do as follows:
   a) If $G(g, \cdot) \equiv \mathrm{XOR}$ then set $X_g^0 = \bigoplus_{i=1}^t X_{W(g,i)}^0$ and $X_g^1 = X_g^0 \oplus X_{W(g,1)}^0 \oplus X_{W(g,1)}^1$.
   b) Else set $X_g^0 = \mathsf{KDF}\left(X_{W(g,1)}^1, \ldots, X_{W(g,t)}^1; (\mathtt{key}, g, 0)\right)$ and $X_g^1 = \mathsf{KDF}\left(X_{W(g,1)}^0; (\mathtt{key}, g, 1)\right)$.
4. Return X_w^1.

Next consider the extractor when using free-XOR (the scheme in Fig. 3.2 composed with Fig. 3.4). We define it as follows:

$\mathrm{Ext}(r, f, y = 1) \rightarrow Y'$**:**

1. Compute $(F, e, d) \leftarrow \mathsf{Gb}\left(1^\kappa, f; r\right)$.
2. Parse $(n, q, I, W, G) \leftarrow f$, $(n, q, I, W, G, P) \leftarrow F$, and $\left(X_1^0, X_1^1, \ldots, X_n^0, X_n^1\right) \leftarrow e$.
3. Set $\Delta = X_1^0 \oplus X_1^1$.
4. For each $g \in [n+1; w]$ let $t = I(g)$ and do as follows:
   a) If $G(g, \cdot) \equiv \mathrm{XOR}$ then set $X_g^0 = \bigoplus_{i=1}^t X_{W(g,i)}^0$ and $X_g^1 = X_g^0 \oplus \Delta$.



$\mathsf{InKeys}(n, \kappa) \to^* \left(X_i^0, X_i^1\right)_{i \in [n]}$

1. Sample a uniformly random difference $\Delta \in_R \{0,1\}^\kappa$.
2. For $i \in [n]$ sample a uniformly random $X_i^0 \in_R \{0,1\}^\kappa$ and return the list $\left(X_i^0, X_i^0 \oplus \Delta\right)_{i \in [n]}$.

$\mathsf{Garb}\left(g, \mathrm{g}, \left(X_i^0, X_i^1\right)_{i \in [t]}\right) \to \left(\left(X_g^0, X_g^1\right), P_g\right)$

1. Set $\Delta = X_1^0 \oplus X_1^1$.
2. If g = NAND do as follows:
   a) Let $X_g^0 = \mathsf{KDF}\left(X_1^1, X_2^1, \ldots, X_t^1; (\mathtt{key}, g, 0)\right)$ and $X_g^1 = X_g^0 \oplus \Delta$.
   b) Next let $P_g[i] = X_g^1 \oplus \mathsf{KDF}\left(X_i^0; (\mathtt{inte}, g, i)\right)$ for all $i \in [t]$ and set $P_g = (P_g[i])_{i=1}^t$.
   c) Return $\left(\left(X_g^0, X_g^1\right), P_g\right)$.
3. If instead g = XOR set $X_g^0 = \bigoplus_{i=1}^t X_i^0$, $X_g^1 = X_g^0 \oplus \Delta$ and return $\left(\left(X_g^0, X_g^1\right), \perp\right)$.

$\mathsf{Eval}\left(g, \mathrm{g}, \vec{v}, (X_i)_{i \in [t]}, P_g\right) \to X_g$

1. If g = NAND do as follows: If $\vec{v} = 1^t$ then set $X_g = \mathsf{KDF}\left(X_1, X_2, \ldots, X_t; (\mathtt{key}, g, 0)\right)$. Otherwise find the first $i \in [t]$ s.t. $\vec{v}_i = 0$, parse $(P_g[i])_{i=1}^t \leftarrow P_g$ and compute and return $X_g = P_g[i] \oplus \mathsf{KDF}\left(X_i; (\mathtt{inte}, g, i)\right)$.
2. If instead g = XOR return $X_g = \bigoplus_{i=1}^t X_i$.

Figure 3.4: Garbling GRR2 - With free-XOR

   b) Else set $X_g^0 = \mathsf{KDF}\left(X_{W(g,1)}^1, \ldots, X_{W(g,t)}^1; (\mathtt{key}, g, 0)\right)$ and $X_g^1 = X_g^0 \oplus \Delta$.

5. Return X_w^1.

Now, from correctness and these extractors we see that the schemes have verifiability, as we verify by regenerating each garbled gate, and hence a verified garbled gate is correctly generated. A bit more formally, assume for the sake of contradiction that there exists a PPT adversary $\mathcal{A}$ who can generate (F, f, e, d, r) such that he wins the $\mathrm{Ver}_{\mathcal{G}}^{\mathcal{A}}(1^\kappa)$ game. This must necessarily mean that $\mathsf{Ve}(F, f, e, d, r) = 1$. This again must mean that $(F, e, d) \leftarrow^* \mathsf{Gb}(1^\kappa, f)$, so if there exists an $x \in \{0,1\}^n$ such that $\mathrm{Ext}(r, f, 1) \neq \mathsf{Ev}(F, \mathsf{En}(e, x))$ then it must mean that the extractor, at some wire g computes a "wrong key", X_g^0 or X_g^1. However, both for the GRR1 and GRR2 schemes these are restored from F in accordance with each correctly generated gate.

This takes care of the demands of correctness (Definition 6) and verifiability (property ver) of a secure privacy-free garbling scheme, as defined in Definition 7. What remains is authenticity (property aut): In the following we reduce this to the security of the KDF used.

Theorem 4. *If the* KDF *used in the garbling scheme of Fig. 3.2 composed with Fig. 3.3 is secure according to Definition 8, then* $\mathrm{Adv}_{\mathcal{G}}^{\mathrm{aut}}(\mathcal{A}, \kappa)$ *is negligible in* κ.

Proof. For notational convenience we are going to focus on the case with fan-in 2. The proof idea generalizes immediately.

A NAND gate with input keys L^0, L^1 for the left wire and R^0, R^1 for the right wire and

gate identifier g is garbled as follows:

$$O^1 \leftarrow \mathsf{KDF}\left(L^0; (\mathtt{key}, g, 1)\right) , \tag{3.1}$$
$$O^0 \leftarrow \mathsf{KDF}\left(L^1, R^1; (\mathtt{key}, g, 0)\right) , \tag{3.2}$$
$$A \leftarrow \mathsf{KDF}\left(R^0; (\mathtt{inte}, g, 2)\right) , \tag{3.3}$$
$$P \leftarrow A \oplus O^1 \quad \text{(with label } (\mathtt{garb}, g)) . \tag{3.4}$$

The output keys are (O^0, O^1). The garbled gate is just P.

A XOR gate with input keys L^0, L^1 for the left wire and R^0, R^1 for the right wire and gate identifier g is garbled as follows:

$$O^0 \leftarrow L^0 \oplus R^0 \quad \text{(with label } (\mathtt{key}, g, 0)) , \tag{3.5}$$
$$O^1 \leftarrow L^1 \oplus R^0 \quad \text{(with label } (\mathtt{key}, g, 1)) , \tag{3.6}$$
$$P \leftarrow L^0 \oplus L^1 \oplus R^0 \oplus R^1 \quad \text{(with label } (\mathtt{garb}, g)) . \tag{3.7}$$

The output keys are (O^0, O^1) and the garbled gate is just P.

Besides this, the circuit garbling just consists of reusing the appropriate output keys as input keys to later gates. A garbled circuit F consists of, amongst other, a garbled gate for each of the q internal wires, $P = (P_{n+1}, \ldots, P_w)$, in topological order. For each garbled gate P_g, let $L_g^0 = X_{W(g,1)}^0$ and $L_g^1 = X_{W(g,1)}^1$ be the corresponding keys on the left input wire, let $R_g^0 = X_{W(g,2)}^0$ and $R_g^1 = X_{W(g,2)}^1$ be the corresponding keys on the right input wire, and let $O_g^0 = X_g^0$ and $O_g^1 = X_g^1$ be the output keys. (If $g \leq n$ then we let $O_g^0 = X_g^0$ and $O_g^1 = X_g^1$ for the input keys X_g^0 and X_g^1.)

We can assume w.l.o.g. that the last gate is the output gate. For encoding information $e = (X_1^0, X_1^0, \ldots, X_n^0, X_n^1)$ and a plaintext input $x \in \{0,1\}^n$, let $X = (X_1^{x_1}, \ldots, X_n^{x_n})$ be the garbled version of x. For $i \in [n]$ let v_i be the i'th input bit, i.e. $v_i = x_i$. For $g \in [n+1; w]$, let v_g be the bit we get by computing plaintext gate number g on the bits for its input wires, that is $v_g = G\left(g, \left(v_{W(g,1)}, v_{W(g,2)}\right)\right)$ in accordance with Fig. 3.2. This defines a *plaintext evaluation* $\vec{v} = (v_1, \ldots, v_n, v_{n+1}, \ldots, v_w)$. For $g \in [n+1; w]$, let $X_g = O_g^{v_g}$. This defines a *garbled evaluation* $X^x = (X_1, \ldots, X_n, X_{n+1}, \ldots, X_w)$. The scheme is constructed such that a correct garbled circuit F and X allows one to compute $X_w = O_w^{f(x)}$. We have to prove that from a randomly generated P and X one cannot also efficiently compute $O_w^{1-f(x)}$. For this, it is sufficient to prove that one cannot efficiently compute $\left(g, O_g^{1-v_g}\right)$ for any $g \in [w]$ with non-negligible probability.

We do the proof by a simple reduction to the game KDF in Fig. 3.1. It is easy to see that the garbling and the keys learned by the evaluator in the scheme can be computed by queries to the game KDF in such a way that all the keys $O_g^{1-v_g}$ are uncompromised. In more detail, the reduction runs as follows:

Input keys: For each $i \in [n]$ and $b \in \{0,1\}$ output $(\mathtt{fresh\ key}, (\mathtt{key}, i, b))$ to define a fresh random key $X_i^b \in_R \{0,1\}^\kappa$. Then for each $i \in [n]$, output $(\mathtt{leak}, (\mathtt{key}, i, x_i))$ to add $X_i^{x_i}$ to the set of values to leak. Let $X = (X_1^{x_1}, \ldots, X_n^{x_n})$. Now for each input wire both keys are defined in the game KDF.

Internal gates: Iteratively go through all the gates. Specifically for each $g \in [n+1; w]$ we do as follows, depending on whether or not gate g is a NAND or XOR gate:

NAND gate: Call the plaintext value on the left input wire $l_g = v_{W(g,1)}$, call the plaintext value on the right input wire $r_g = v_{W(g,2)}$, and call the plaintext value on the output wire v_g. Call the keys on these wires $\left(L_g^0, L_g^1\right)$, $\left(R_g^0, R_g^1\right)$, and $\left(O_g^0, O_g^1\right)$ respectively. Thus $\left(L_g^0, L_g^1\right) = \left(X_{W(g,1)}^0, X_{W(g,1)}^2\right)$, $\left(R_g^0, R_g^1\right) = \left(X_{W(g,2)}^0, X_{W(g,2)}^1\right)$, and $\left(O_g^0, O_g^1\right) = \left(X_g^0, X_g^1\right)$. The first four of these keys are defined in the game KDF and we are given $L_g^{l_g}$ and $R_g^{r_g}$ before our guess. We should define $\left(O_g^0, O_g^1\right)$ in the game and make sure we learn $O_g^{v_g}$ before our guess. We use `derive` commands to define $O_g^1 = \mathsf{KDF}\left(L_g^0; (\mathtt{key}, g, 1)\right)$, $O_g^0 = \mathsf{KDF}\left(L_g^1, R_g^1; (\mathtt{key}, g, 0)\right)$, and $A_g = \mathsf{KDF}\left(R_g^0; (\mathtt{inte}, g, 2)\right)$. Then we use a `linear` command to define $P_g = A_g \oplus O_g^1$ (with label $(\mathtt{garb}, g)$). Then we add P_g to the set of values to leak by outputting $(\mathtt{leak}, (\mathtt{garb}, g))$. This is a correct garbling, so when we are later given $L_g^{l_g}$ and $R_g^{r_g}$, we can use them to compute $O_g^{v_h}$ by computing the garbled gate on $\left(L_g^{l_g}, R_g^{r_g}\right)$.

XOR gate: We proceed as for NAND gates, except for the specific commands issued: We use `linear` commands to define $O_g^0 = L_g^0 \oplus R_g^0$ (under identifier $(\mathtt{key}, g, 0)$), $O_g^1 = L_g^1 \oplus R_g^0$ (under identifier $(\mathtt{key}, g, 1)$) and $P_g = L_g^0 \oplus L_g^1 \oplus R_g^0 \oplus R_g^1$ (under identifier $(\mathtt{garb}, g)$). Then we add P_g to the set of values to leak by outputting $(\mathtt{leak}, (\mathtt{garb}, g))$. This is a correct garbling, so we later use it to compute $O_g^{v_g}$ by computing the garbled gate on $\left(L_g^{l_g}, R_g^{r_g}\right)$.

End: After having handled all the gates, we issue the `end` command and learn the input keys $X_i = X_i^{x_i}$ for $i \in [n]$, along with the garbled gates P_g for $g \in [n+1; w]$. Using these we can evaluate the garbled circuit and thus learn the value $X_g = O_g^{v_g}$ for all $g \in [n+1; w]$. We then give $X^x = (X_1, \ldots, X_w)$ to the adversary.

Guess: If the adversary outputs $\left(g, X_g^{1-v_g}\right)$ for any $g \in [w]$ then we output $\left(\mathtt{guess}, (\mathtt{key}, g, 1 - v_g), X_g^{1-v_g}\right)$.

It is clear that we win the guessing game exactly when $(\mathtt{key}, g, 1 - v_g)$ is uncompromised and $X_g^{1-v_g}$ is the correct "other" key for wire i supplied by the adversary – we call $X_g^{v_g}$ the *known key* and we call $X_g^{1-v_g}$ the *other* key. We call a key X_g^b *compromised* if the label $(\mathtt{key}, g, b)$ is compromised as defined by the KDF game. We call gate P_g *compromised* if the *other* key $X_g^{1-v_g}$ is compromised as defined by the KDF game.

It is sufficient to prove that $(\mathtt{key}, g, 1 - v_g)$ is uncompromised for all $g \in [w]$. It is clear that whether $(\mathtt{key}, g, 1 - v_g)$ is uncompromised does not depend on the strategy of the adversary, only the structure of the circuit, the nature of our garbling scheme and the input x. Hence, if for a fixed circuit and fixed input x some $(\mathtt{key}, g, 1 - v_g)$ is sometimes compromised, then it is always compromised. Hence, if any $(\mathtt{key}, g, 1 - v_g)$ can be compromised, then there exists a first gate j such that before executing the commands corresponding to gate j, no identifier $(\mathtt{key}, g, 1 - v_g)$ was compromised, and after executing the commands corresponding to gate j, some identifier $(\mathtt{key}, g, 1 - v_g)$ is compromised, where $g \le j$. Furthermore, among the commands executed for gate j there is a first command that leads to a compromise of a gate. We call this command *patient zero*. We first show that patient zero is not a `derive` command. Then we show that it is not a linear command followed by a leak command. Then we are done.

Assume first that patient zero is a `derive` command. We use several times that a `derive` command, when it is the last command to have been executed, cannot compromise any other key than its output key. When patient zero is a `derive` command, then gate j must be a NAND gate, as there are no `derive` commands executed in the construction of XOR gates. Recall that we issue the `derive` commands (3.1), (3.2), and (3.3), as part of a NAND gate, and then we leak P_j. Assume that $l_j = 0$. In that case $O_j^1 = \mathsf{KDF}\left(L_j^0; (\texttt{key}, j, 1)\right)$ is a known key and hence cannot be a compromised *other* key. We can also assume that L_j^1 is uncompromised (as it is an *other* key and we are at patient zero), and hence the *other* output key $O_j^0 = \mathsf{KDF}\left(L_j^1, R_j^1; (\texttt{key}, j, 0)\right)$ will clearly be uncompromised after executing the command. Assume then that $r_j = 0$. In that case the other output key is again $O_j^0 = \mathsf{KDF}\left(L_j^1, R_j^1; (\texttt{key}, j, 0)\right)$, and now R_j^1 is uncompromised. The command $A_j = \mathsf{KDF}\left(R_j^0; (\texttt{inte}, j, 2)\right)$ can therefore never be the patient zero compromising an output key, as A_j is not an output key.

Before we prove that patient zero cannot be a linear command we change the system that we analyze by replacing the processing of all NAND gates by the following commands: First we execute $(\texttt{fresh key}, (\texttt{key}, j, 0))$, $(\texttt{fresh key}, (\texttt{key}, j, 1))$, and $(\texttt{fresh key}, (\texttt{inte}, j, 2))$ to define the values O_j^0, O_j^1, and A_j respectively. Then we compute $P_j = A_j \oplus O_j^1$, and leak P_j by issuing the commands $(\texttt{linear}, (\texttt{garb}, j), (\texttt{inte}, j, 2), (\texttt{key}, j, 1))$, and $(\texttt{leak}, (\texttt{garb}, j))$ in that order. In addition we leak $O_j^{v_j}$. If $r_j = 0$ such that R_j^0 is a known key, then we also leak A_j. So, we essentially skip all `derive` commands and simulate their effect on the system by leaking the produced known keys. Since we could compute $O_j^{v_j}$ before the change, it was compromised before the change. It is also compromised after the change, as we now leak it. Similarly for A_j. Hence, the set of compromised identifiers is the same before and after the introduced changes, *at least right after the gate has been handled*. As a consequence, we have not changed whether or not some *other* key later gets compromised.[6] Furthermore, notice that since we have already showed that patient zero could not be a `derive` command this change does not affect the adversary's advantage. We therefore just have to prove that in the modified system, no *other* key gets compromised. Since there are no `derive` commands left, this is simple linear algebra.

Assume that patient zero is $P_j = A_j \oplus O_j^1$. Since A_j is a fresh key and only occurs in this equation, if A_j is uncompromised adding this equation cannot change whether an output key is compromised or not.[7] Hence it must be the case that A_j is compromised. Since A_j is fresh and occurs in no other equation, this can only have happened because we leaked it earlier. Hence R_j^0 is a known key. So, $l_j = 0$ and hence $v_j = 1$. Therefore O_j^1 is a known key and hence already compromised. Hence $P_j = A_j \oplus O_j^1$ will compromise A_j, but since A_j occurs in no other equation, this does not further change the status of any variable. We can therefore assume in the following that we process all NAND gates, with index g, as follows: Call $(\texttt{fresh key}, (\texttt{key}, g, 0))$, $(\texttt{fresh key}, (\texttt{key}, g, 1))$, and $(\texttt{leak}, (\texttt{key}, g, v_g))$ to first define the key O_g^0, O_g^1 and then leak $O_g^{v_g}$. This does not change whether or not there will be a patient zero. We can even make further changes. We once and for all create a global key Δ through the call $(\texttt{fresh key}, \texttt{delta})$. Then we execute each NAND gate as follows: Call $(\texttt{fresh key}, (\texttt{key}, g, 0))$, $(\texttt{linear}, (\texttt{key}, g, 1), (\texttt{key}, g, 0), \texttt{delta})$, and $(\texttt{leak}, (\texttt{key}, g, v_g))$ to define the keys O_g^0 and O_g^1 respectively and leak $O_g^{v_g}$. Similarly we can create the input keys X_i^0 and $X_i^1 = X_i^0 \oplus \Delta$ by

[6] Note that if eventually an *other* key gets compromised, then the introduced changes *will* have an effect. When we use `derive` commands, one compromised *other* key leads to many compromised *other* keys. When we use fresh key commands, a compromised *other* key might not have an avalanche effect. However, we are proving that the number of compromised *other* keys is 0, and hence using one system or the other is equally good.

[7] If O_j^1 is uncompromised then A_j goes from uncompromised to compromised, but A_j is not an output key, and clearly no other key than A_j can change status by this equation.

calling $(\texttt{fresh key}, (\texttt{key}, i, 0))$, and $(\texttt{linear}, (\texttt{key}, i, 1), (\texttt{key}, i, 0), \texttt{delta})$ respectively for $i \in [n]$. This will only *add* equations to the system, and hence if there was a patient zero in the system before the change there will also be a patient zero in the system after the change.

Assume then that patient zero is a linear command from a XOR gate, again with index j. We process such a gate as follows: Compute $O_j^0 \leftarrow L_j^0 \oplus R_j^0$ (with label $(\texttt{key}, j, 0)$), $O_j^1 \leftarrow L_j^1 \oplus R_j^0$ (with label $(\texttt{key}, j, 1)$), and $P_j \leftarrow L_j^0 \oplus L_j^1 \oplus R_j^0 \oplus R_j^1$ (with label $(\texttt{garb}, j)$) using the **linear** command, and leak P_j using the **leak** command. Notice that $L_j^0 \oplus L_j^1 \oplus R_j^0 \oplus R_j^1 = \Delta \oplus \Delta = 0$. Hence leaking P_j does not change the status of any key. We can therefore assume that we process XOR gates as follows: Compute $O_j^0 \leftarrow L_j^0 \oplus R_j^0$ and $O_j^1 \leftarrow L_j^1 \oplus R_j^0$ using the **linear** command.

After all the changes to the system we now "garble" as follows: First call

$$\Delta \leftarrow^* (\texttt{fresh key}, \texttt{delta}) .$$

Then for each input key, $i \in [n]$, do:

$$\begin{aligned} X_i^0 &\leftarrow^* (\texttt{fresh key}, (\texttt{key}, i, 0)) , \\ X_i^1 &\leftarrow (\texttt{linear}, (\texttt{key}, i, 1), (\texttt{key}, i, 0), \texttt{delta}) , \\ X_i^{x_i} &\leftarrow (\texttt{leak}, (\texttt{key}, i, x_i)) . \end{aligned}$$

For each NAND gate, with index g, do:

$$\begin{aligned} O_g^0 &\leftarrow^* (\texttt{fresh key}, (\texttt{key}, g, 0)) , \\ O_g^1 &\leftarrow (\texttt{linear}, (\texttt{key}, g, 1), (\texttt{key}, g, 0), \texttt{delta}) , \\ O_g^{v_g} &\leftarrow (\texttt{leak}, (\texttt{key}, g, v_g)) . \end{aligned}$$

Finally, for each XOR gate, with index g, do:

$$\begin{aligned} O_g^0 &\leftarrow (\texttt{linear}, (\texttt{key}, g, 0), (\texttt{key}, W(g,1), 0) , (\texttt{key}, W(g,2), 0)) , \\ O_g^1 &\leftarrow (\texttt{linear}, (\texttt{key}, g, 0), (\texttt{key}, W(g,1), 1) , (\texttt{key}, W(g,2), 0)) , \\ O_g^{v_g} &\leftarrow (\texttt{leak}, (\texttt{key}, g, v_g)) . \end{aligned}$$

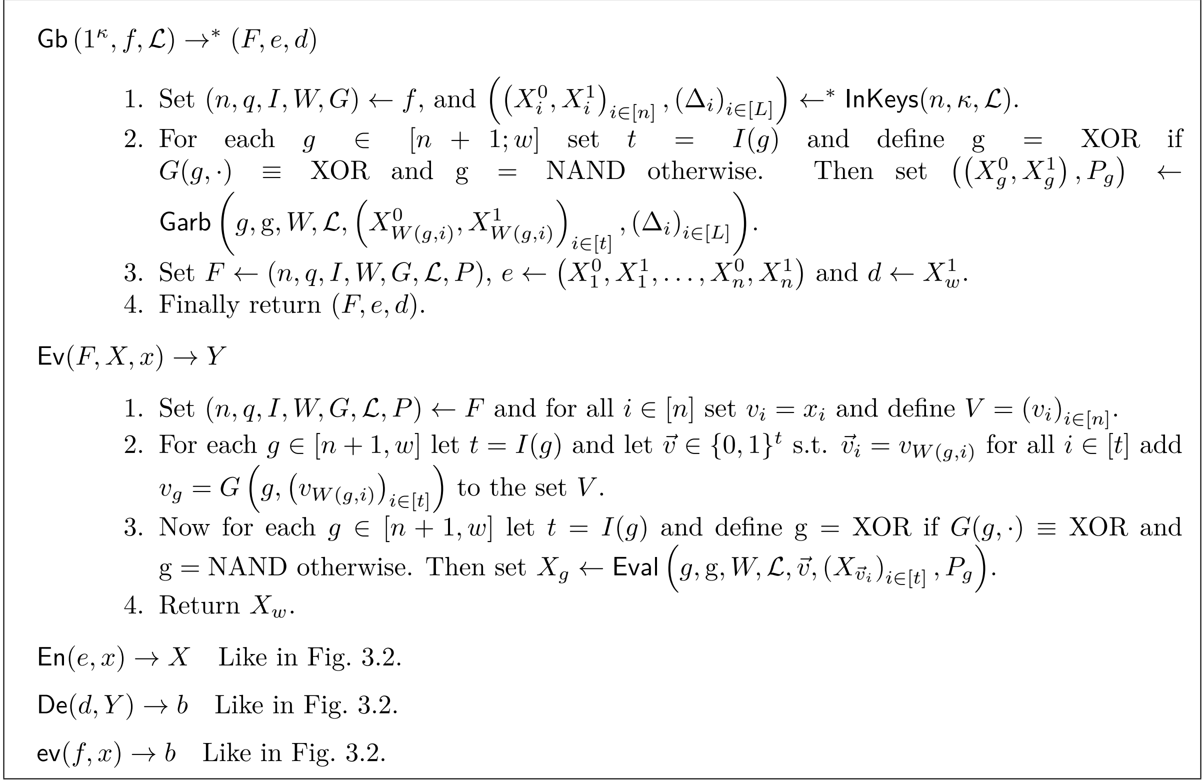
It is then fairly straight-forward to see that there are no compromised *other* key. In particular, it is trivial to see that if an other key would be compromised in this system, then the free-XOR scheme from [KS08] would trivially be insecure, as the system of equations created by the free-XOR scheme is a super set of the system created by the above commands. We therefore refer to [KS08] for the details of why the free-XOR trick is secure. □

Notice that we can use a subset of this proof to prove security of our free-XOR privacy-free garbling scheme, since the free-XOR already implements the global difference Δ. Specifically we have the following corollary:

Corrollary 1. *If the* KDF *used in the garbling scheme of Fig. 3.2 composed with Fig. 3.4 is secure according to Definition 8, then* $\text{Adv}_{\mathcal{G}}^{\text{aut}}(\mathcal{A}, \kappa)$ *is negligible in* κ.

3.3 Privacy-Free FleXOR

In [KMR14] Kolesnikov *et al.* introduced a generalization and optimization of the free-XOR approach which allows to weaken the security assumption needed for free-XOR and/or limit the



$\mathsf{Gb}(1^\kappa, f, \mathcal{L}) \rightarrow^* (F, e, d)$

1. Set $(n, q, I, W, G) \leftarrow f$, and $\left(\left(X_i^0, X_i^1\right)_{i\in[n]}, (\Delta_i)_{i\in[L]}\right) \leftarrow^* \mathsf{InKeys}(n, \kappa, \mathcal{L})$.
2. For each $g \in [n+1; w]$ set $t = I(g)$ and define $\mathrm{g} = \mathrm{XOR}$ if $G(g, \cdot) \equiv \mathrm{XOR}$ and $\mathrm{g} = \mathrm{NAND}$ otherwise. Then set $\left(\left(X_g^0, X_g^1\right), P_g\right) \leftarrow \mathsf{Garb}\left(g, \mathrm{g}, W, \mathcal{L}, \left(X^0_{W(g,i)}, X^1_{W(g,i)}\right)_{i\in[t]}, (\Delta_i)_{i\in[L]}\right)$.
3. Set $F \leftarrow (n, q, I, W, G, \mathcal{L}, P)$, $e \leftarrow \left(X_1^0, X_1^1, \ldots, X_n^0, X_n^1\right)$ and $d \leftarrow X_w^1$.
4. Finally return (F, e, d).

$\mathsf{Ev}(F, X, x) \rightarrow Y$

1. Set $(n, q, I, W, G, \mathcal{L}, P) \leftarrow F$ and for all $i \in [n]$ set $v_i = x_i$ and define $V = (v_i)_{i\in[n]}$.
2. For each $g \in [n+1, w]$ let $t = I(g)$ and let $\vec{v} \in \{0,1\}^t$ s.t. $\vec{v}_i = v_{W(g,i)}$ for all $i \in [t]$ add $v_g = G\left(g, \left(v_{W(g,i)}\right)_{i\in[t]}\right)$ to the set V.
3. Now for each $g \in [n+1, w]$ let $t = I(g)$ and define $\mathrm{g} = \mathrm{XOR}$ if $G(g, \cdot) \equiv \mathrm{XOR}$ and $\mathrm{g} = \mathrm{NAND}$ otherwise. Then set $X_g \leftarrow \mathsf{Eval}\left(g, \mathrm{g}, W, \mathcal{L}, \vec{v}, (X_{\vec{v}_i})_{i\in[t]}, P_g\right)$.
4. Return X_w.

$\mathsf{En}(e, x) \rightarrow X$ Like in Fig. 3.2.

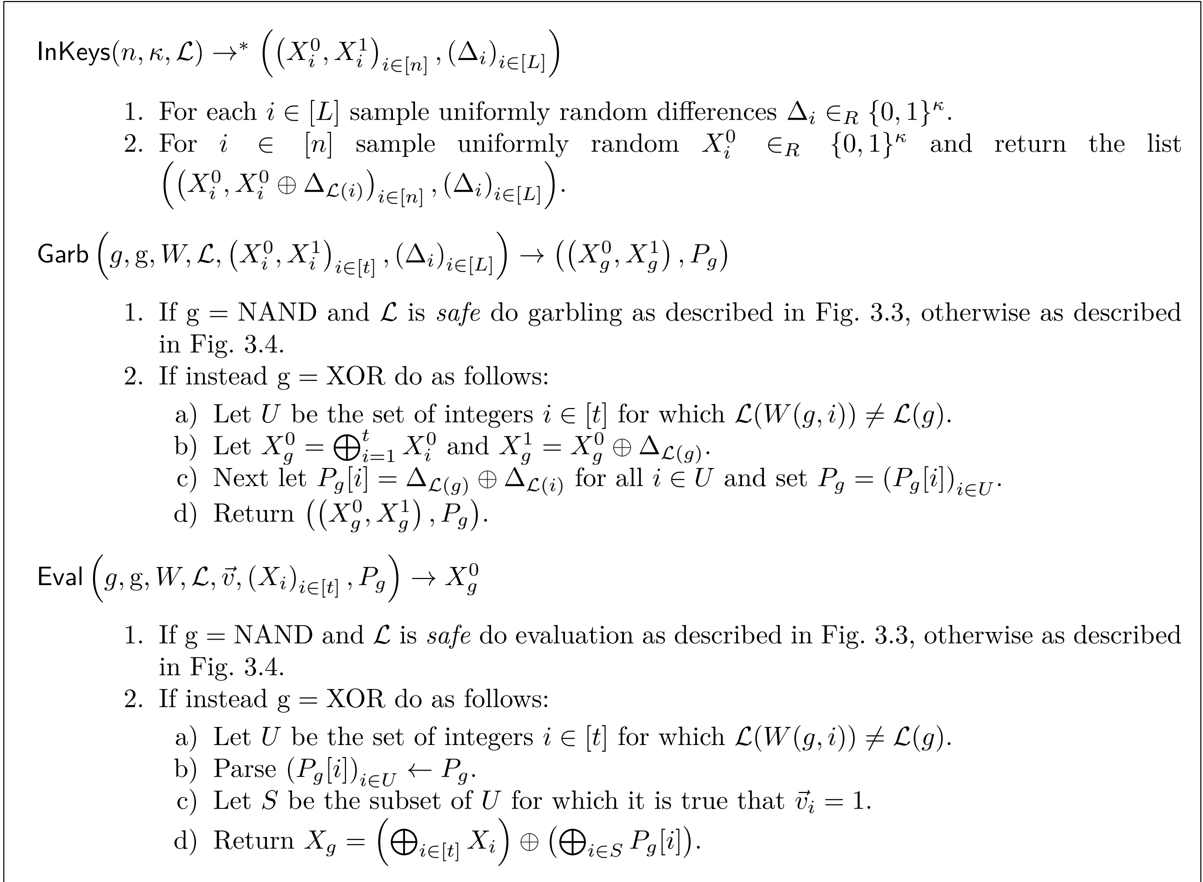
$\mathsf{De}(d, Y) \rightarrow b$ Like in Fig. 3.2.

$\mathsf{ev}(f, x) \rightarrow b$ Like in Fig. 3.2.

Figure 3.5: Privacy-Free FleXOR Garbling

amount of ciphertexts used to garble non-XOR gates. In their schemes (only considering fan-in 2 gates) non-XOR gates are constructed exactly as one would in a regular garbling scheme, but XOR gates are constructed differently and, depending on a wire ordering of the circuit, consists of either 0, 1 or 2 ciphertexts. When the garbling scheme used implements aggressive row reduction (i.e., GRR1) this yields an overall smaller size for most garbled circuits compared to the size of garbled circuits constructed using the free-XOR approach.

Their overall idea can be seen as a generalization of the free-XOR approach: the 1-key on each wire is defined to be the 0-key XOR'ed with some difference. However, unlike in the free-XOR approach, where there is only a single difference associated with a whole garbled circuit, they use several distinct differences. The amount of distinct differences in a fleXOR circuit is based on the topology of the circuit garbled. In particular the wires of the circuit are grouped into equivalence classes, each based on one particular difference. The classes are then constructed such that there is at most one non-XOR gate using a specific difference on its output wire. This makes it possible to use the aggressive row reduction approach and thus having both the 0- and 1-output key of a non-XOR gate be directly derived from its input keys. A XOR gate is then constructed by up to four ciphertexts, each working on a single key of an input wire to a XOR gate. That is, one for the left 0-, left 1-, right 0-, and right 1-input keys of a particular XOR gate. A ciphertext simply encrypts a single output key with the same semantic as its single input key, but under another difference. Thus, depending on the topology, we might need zero ciphertexts, two for the left wire and/or two for the right wire. Using the ciphertexts it is possible for the evaluator to "change" the difference of a given XOR gate input wire, such that he can evaluate it as in the free-XOR approach. Finally, two of the four ciphertexts can be eliminated using row



$\mathsf{InKeys}(n, \kappa, \mathcal{L}) \to^* \left(\left(X_i^0, X_i^1\right)_{i \in [n]}, (\Delta_i)_{i \in [L]} \right)$

1. For each $i \in [L]$ sample uniformly random differences $\Delta_i \in_R \{0,1\}^\kappa$.
2. For $i \in [n]$ sample uniformly random $X_i^0 \in_R \{0,1\}^\kappa$ and return the list $\left(\left(X_i^0, X_i^0 \oplus \Delta_{\mathcal{L}(i)}\right)_{i \in [n]}, (\Delta_i)_{i \in [L]} \right)$.

$\mathsf{Garb}\left(g, \mathrm{g}, W, \mathcal{L}, \left(X_i^0, X_i^1\right)_{i \in [t]}, (\Delta_i)_{i \in [L]}\right) \to \left(\left(X_g^0, X_g^1\right), P_g\right)$

1. If g = NAND and $\mathcal{L}$ is *safe* do garbling as described in Fig. 3.3, otherwise as described in Fig. 3.4.
2. If instead g = XOR do as follows:
   a) Let U be the set of integers $i \in [t]$ for which $\mathcal{L}(W(g,i)) \neq \mathcal{L}(g)$.
   b) Let $X_g^0 = \bigoplus_{i=1}^t X_i^0$ and $X_g^1 = X_g^0 \oplus \Delta_{\mathcal{L}(g)}$.
   c) Next let $P_g[i] = \Delta_{\mathcal{L}(g)} \oplus \Delta_{\mathcal{L}(i)}$ for all $i \in U$ and set $P_g = (P_g[i])_{i \in U}$.
   d) Return $\left(\left(X_g^0, X_g^1\right), P_g\right)$.

$\mathsf{Eval}\left(g, \mathrm{g}, W, \mathcal{L}, \vec{v}, (X_i)_{i \in [t]}, P_g\right) \to X_g^0$

1. If g = NAND and $\mathcal{L}$ is *safe* do evaluation as described in Fig. 3.3, otherwise as described in Fig. 3.4.
2. If instead g = XOR do as follows:
   a) Let U be the set of integers $i \in [t]$ for which $\mathcal{L}(W(g,i)) \neq \mathcal{L}(g)$.
   b) Parse $(P_g[i])_{i \in U} \leftarrow P_g$.
   c) Let S be the subset of U for which it is true that $\vec{v}_i = 1$.
   d) Return $X_g = \left(\bigoplus_{i \in [t]} X_i\right) \oplus \left(\bigoplus_{i \in S} P_g[i]\right)$.

Figure 3.6: Garbling - Using fleXOR

reduction (one of the left wire ciphertexts and one of the right wire ciphertexts).

Here we propose a variant of fleXOR which combines their ideas with non-oblivious gate evaluation, leading to significant improvements in terms of computation complexity. Before we can describe our privacy-free fleXOR construction we need a few definitions. These are taken almost verbatim from [KMR14]. We assume familiarity with their construction and direct the reader to their paper if that is not the case.

Definition 11 (Wire Ordering)**.** A *wire ordering* of a Boolean circuit f is a function $\mathcal{L}$ that assigns an integer to each wire in f. Without loss of generality, we assume that $im(\mathcal{L}) = \{1, \ldots, L\}$ for some integer L, and we denote $|\mathcal{L}| = L$. We say a wire ordering $\mathcal{L}$ is *safe* if:

- For each NAND gate with output wire g, and each wire j where there exists a directed path in the circuit that contains wire j before wire g, we have $\mathcal{L}(g) > \mathcal{L}(j)$.

- For each value $\ell \in im(\mathcal{L})$, there is at most one NAND gate whose output wire g satisfies $\mathcal{L}(g) = \ell$.

We say that a topological ordering of gates in a circuit f is *safety-respecting* of $\mathcal{L}$ if every NAND gate g appears earlier in the ordering than any other gate j satisfying $\mathcal{L}(g) = \mathcal{L}(j)$.

Formal description. We describe the privacy-free fleXOR protocol for gates of fan-in t in Fig. 3.5 and Fig. 3.6. Notice that the description in Fig. 3.5 is essentially the same as the one for the privacy-free scheme we described in Fig. 3.2, except for the fact that we include the wire ordering $\mathcal{L}$ needed in order for the garbling scheme to know which Δ's should be used for which wires. Regarding the specificities of the garbling, described in Fig. 3.6, see that the garbling of NAND gates is exactly the same as in Fig. 3.3 and Fig. 3.4, depending on whether or not the wire ordering is safe. That is, the scheme first checks whether or not a gate is a XOR or NAND gate. If it is a NAND gate then the garbling is the same as in Fig. 3.3 if $\mathcal{L}$ is *safe*, and the same as in Fig. 3.4 if $\mathcal{L}$ is not safe. The functions for encoding, decoding and plain evaluation are the same as for the previous schemes, and thus follow the description of Fig. 3.2.

Regarding XOR gates, we garble them essentially as in Fig. 3.3 but, since the offsets of the wires are chosen during the InKeys procedure, the Garb procedure can only define the 0-key corresponding to the output wire. Then, as in Fig. 3.3, the Garb procedure computes and outputs the XOR of the offsets between the inputs and output wire, but only for the wires that belong to the set U, that is those for which $\mathcal{L}(j) \neq \mathcal{L}(g)$, which means that the Δ used for the 1-key on wire j is different from the Δ used on the output wire of the gate g. This in turn means that we must associate a ciphertext in order to "adjust" the key on wire j.

Regarding evaluation: for NAND gates the scheme again does the same as in Fig. 3.3 and Fig. 3.4 depending on whether or not the wire ordering is safe or not, respectively. For XOR gates the scheme first defines (in step a) the set of input wires for which $\mathcal{L}(j) \neq \mathcal{L}(g)$, U, and parses the garbled gate P_g to its ciphertexts, $(P_g[i])_{i \in U}$. Then in step c the scheme identifies the subset $S \subset U$ of the input wires for which it is true that the input value for wire j is equal to 1 and finally, in step d it computes the output key by XOR'ing all input keys and the adjustments for all the wires belonging to the set S.

Security. Like for our other privacy-free garbling schemes, correctness and verifiability follows relatively straightforwards from the constructions. The proof of authenticity follows from the one for the scheme in Fig. 3.3 (since the fleXOR variant is a generalization of the schemes described in Fig. 3.3, for which some input wires happen to have the same offset as the output wire) and from the assumption on the wire ordering. We refer to [KMR14] for more details.

3.4 Efficiency Improvements

Our garbling schemes differ in performance in terms of communication and computation overhead. It is natural to ask which one is the most efficient one. Like most interesting questions, the answer is not as simple as one might want, and to answer which garbling scheme offers the best performances one must define the price of communication vs. computation. The ultimate answer depends on the actual hardware setting (CPU, network) on which the protocol is to be run and can only be determined empirically.

In Table 3.2 we benchmark our garbling scheme against the best previous garbling schemes, on a number of circuits. The circuits used are due to Smart and Tillich and are publicly available [ST12]. Note however that the numbers in our tables depend on the actual circuits being used, meaning that it might be possible to find different circuits that compute the same functions but that are more favorable to one or another garbling scheme. Finding such circuits requires non-trivial heuristics and manual work (e.g., [BP12]), as there is evidence that finding such circuits is computationally hard [Fin14, KMR14].

Communication (amortized # of ciphertexts per gate)

Circuit	# of Gates		Private			Privacy-free			Saving
	AND	XOR	GRR2	GRR1	FleXOR	GRR1	free-XOR	fleXOR	
DES	18124	1340	2.0	2.79	1.89	1.0	1.86	**0.96**	49%
AES	6800	25124	2.0	0.64	0.72	1.0	**0.43**	0.51	33%
SHA-1	37300	24166	2.0	1.82	1.39	1.0	1.21	**0.78**	44%
SHA-256	90825	42029	2.0	2.05	1.56	1.0	1.37	**0.87**	44%

Computation (amortized # of encryptions per gate for garbler/evaluator)

Circuit	# of Gates		Private			Privacy-free	Saving
	AND	XOR	GRR2	GRR1	FleXOR	All	
DES	18124	1340	4.0/1.0	3.72/**0.93**	3.78/0.96	**2.79/0.93**	25%/0%
AES	6800	25124	4.0/1.0	0.85/**0.21**	1.44/0.51	**0.64/0.21**	25%/0%
SHA-1	37300	24166	4.0/1.0	2.43/**0.61**	2.78/0.78	**1.82/0.61**	25%/0%
SHA-256	90825	42029	4.0/1.0	2.73/**0.68**	3.11/0.87	**2.05/0.68**	25%/0%

Table 3.2: Comparison with other garbling schemes on some circuit examples from [ST12] in terms of communicational and computational overhead. The fleXOR scheme used is based on a "safe" topological ordering (see [KMR14]). The number in each cell shows the amortized number of ciphertexts to be sent in the **Communication** table and the amortized number of calls to a KDF per gate that the constructor/evaluator need to perform in the **Computation** table. Note that we do not count the non cryptographic operations in this table (polynomial interpolation in private GRR2, XOR of strings in all others). We ignore inversion gates as they can be pulled inside other gates. The "Saving" column is computed against the previously best solution.

Still, no previous garbling scheme performs better than *all* of our proposed schemes, therefore while the actual saving factor might change, one of our schemes will always outperforms the rest.

3.5 Zero-Knowledge Arguments of Knowledge from Garbled Circuits

As we have previously mentioned the original motivation for our study of privacy-free garbled circuits was the setting of ZKAoK of non-algebraic statements. For concreteness we here include the protocol of [JKO13], which does just that, using garbled circuits. We furthermore see that in this protocol we can use a privacy-free garbled circuit instead of a general garbled circuit.

In Figure 3.7 a sketch of the ZKAoK protocol proposed by Jawurek *et al.* [JKO13] is shown. The protocol needs a stronger notion of OT than we have previously used. Specifically we need the sender to be *committing* to her messages when she inputs these to the OT subprotocol. At some point after the receiver inputs his choice bit and receives his choice of message, the sender may open to both the messages she gave as input. Hence we call this notion of OT for

Let $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev}, \mathsf{Ve})$ be a verifiable privacy-free garbling scheme and let L be an NP language and $c \in L$ be some element in the language and $f_c : \{0,1\}^n \to \{0,1\}$ be the verification function that outputs 1 when x is a valid witness for c. Both the prover and verifier have common input c and a security parameter κ. In addition the prover has input $x = (x_1, x_2, \ldots, x_n)$.

1. For all $i \in [n]$ the prover calls $\mathcal{F}_{\mathrm{COT}}$, giving as input to the i'th call the choice bit x_i.
2. The verifier then runs $\mathsf{Gb}(1^\kappa, f_c; r) \to (F, e, d)$, passes $e \to \left(X_1^0, X_1^1, \ldots, X_n^0, X_n^1\right)$ and commits to r using $\mathcal{F}_{\mathrm{COM}}$.
3. The verifier then calls $\mathcal{F}_{\mathrm{COT}}$ for all $i \in [n]$ giving as input to the i'th call the pair of messages $\left(X_i^0, X_i^1\right)$.
4. The prover then learns the list $(X_i^{x_i})_{i \in [n]}$ from $\mathcal{F}_{\mathrm{COT}}$.
5. The verifier then sends F to the prover.
6. The prover runs $\mathsf{Ev}\left(F, (X_i^{x_i})_{i \in [n]}, x\right) \to Y$.
7. The prover commits to Y using the functionality $\mathcal{F}_{\mathrm{COM}}$.
8. The verifier then opens up all the messages she gave as input to $\mathcal{F}_{\mathrm{COT}}$. Thus the prover learns the list of values $\left(X_1^0, X_1^1, \ldots, X_n^0, X_n^1\right) \to e$.
9. The prover then runs $\mathsf{Ve}(F, f_c, e, Y, r) \to b$. If $b = 0$ then the prover terminates the protocol otherwise he accepts and opens the commitment to Y to the verifier.
10. The verifier accepts if the value Y she received from $\mathcal{F}_{\mathrm{COM}}$ is equal to the value Y she computed during the circuit generation.

Figure 3.7: Informal Description of Jawurek *et al.* ZKAoK from garbled circuits.

committing OT and denote its functionality by $\mathcal{F}_{\mathrm{COT}}$. For a discussion of the exact requirements along with suggestions to possible realizations we point the reader to [JKO13].

The zero-knowledge protocol proceeds as follows: The prover (acting as the receiver in the OT) uses the bits of his witness x as choice bits in the OT while the verifier (acting as the sender in the OT) uses as input all the pairs of keys of the garbled circuits. The verifier also sends the garbled circuit F. Now if the prover uses a valid witness, he can evaluate the garbled circuit and compute the output key corresponding to the output bit 1. However, instead of disclosing this key at this stage, the prover commits to it and waits for the verifier to prove that she constructed the garbled circuit correctly (and acted honestly in the OT protocols as well). If this check goes through, the prover opens the commitment and the verifier accepts the proof if the commitment contains the key corresponding to the output bit 1. The main ideas behind the proof of security in [JKO13] are as follows: soundness (the verifier accepts only if the statement is true) is achieved thanks to the *authenticity* property of garbled circuits. At the same time the protocol is zero-knowledge (the verifier learns *only* that the statement is true) because the prover verifies that she generates the garbled circuit honestly before disclosing any information.

Using a "Weak" KDF

Notice that the randomness used to generate the garbled circuit is completely revealed to the prover at the end of the protocol, thus the encryption used to garble the gates only needs to remain secure throughout the execution of the protocol (in contrast to most secure computation settings where the garbling scheme needs to remain secure long after the execution of the protocol

is complete in order to ensure privacy of the input). Therefore one could imagine the following optimization: to garble the circuit, use a "weak" KDF such that it is reasonable to assume that the prover cannot break it in time 10 seconds, and let the verifier accept the proof iff the prover sends the commitment C before 10 seconds have passed. This might allow to use shorter keys and faster KDFs, resulting in better computation and communication complexity.

Part III

Secure Computation Using Garbled Circuits

Chapter 4

Improved Cut-and-Choose of Garbled Circuits

In this chapter we present a consolidation of the work done in [FN13] and [FJN14]. Both these papers present a protocol for doing maliciously secure SFE based on cut-and-choose of garbled circuits. The protocol of the first paper presents a "majority rules" approach to selecting the correct output of the evaluation circuits along with a new way of ensuring consistency of A's input, based on a circuit augmentation. The protocol of the second paper takes its departure in the work of the first paper, but improves on the "majority rules" approach by using the recent idea of "forge-and-lose", which eliminates around a factor 3 of the garbled circuits that need to be constructed and evaluated. The protocol also introduces a new way to realize the "forge-and-lose" approach, which avoids an auxiliary secure two-party computation protocol and does not rely on any number theoretic assumptions, which otherwise has been needed. The second protocol also improves on the circuit augmentation approach to ensure consistency of A's input. Both protocols are constructed to parallelize well in a same instruction, multiple data (SIMD) framework. We present SIMD implementations using a GPU as a massive SIMD device of both protocols. The findings of both protocols compare favorably with all previous implementations of maliciously secure two-party computation.

Finally, we prove the second of the two protocols is universally composable-secure against a malicious adversary assuming access to oblivious transfer, commitment, and coin-tossing functionalities in the non-programmable random oracle model.

Related, concurrent and future work. We have already covered most related work in regards to SFE and garbling in the malicious setting. However, it is appropriate to mention that some of the work previous to [FN13, FJN14] which has been carried out considering massively parallel implementations in the area of secure computation. In [SP11] a cluster of either CPUs or GPUs were used to execute 3072 semi-honestly secure protocols for 1-out-of-2 OT followed by gate garbling/evaluation in parallel. In [KSS12] the authors used up to 512 cores of the Ranger cluster in the Texas Advanced Computing Center to do OTs along with circuit garbling/evaluation in parallel to achieve malicious security using the cut-and-choose approach.

In regards to concurrent work, in [HMSG13] the authors investigate the construction and evaluation of garbled circuits on the GPU. However, their approach requires the transmission of four ciphertexts per garbled gate (unlike three, which we use) since this made it possible for them to parallelize the construction of *all* garbled gates.

It should also be noted that the use of polynomials to implement the forge-and-lose ap-

proach, by leaking a trapdoor of sorts if different circuits evaluate to different output in [FJN14] is inspired by [HKE13]. In their approach, the authors use a verifiable secret sharing scheme to ensure that two secret values associated with each output wire (one for bit 0 and one for bit 1) can always be reconstructed if at least one evaluation circuit is correct. However, whereas they need some verifiability, we can manage with standard Shamir secret sharing [Sha79]. Furthermore, the rest of their protocol is quite different from ours. For example, their protocol is based on a form of dual execution where both parties need to *both* construct *and* evaluate $\Theta(s)$ garbled circuits, meaning that the total communication complexity (and construction/evaluation of garbled circuits) is around a factor two of [FJN14, Bra13, Lin13, AMPR14].

Furthermore, concurrent to [FJN14] Afshar *et al.* [AMPR14] suggest a way of using polynomial interpolation as an add-on optimization to limit communication complexity of their forge-and-lose protocol.

We note that the recent garbling scheme of Zahur *et al.* [ZRE15], requiring only two ciphertext per garbled gate, could most likely directly yield significant improvements on our implementations. The same goes for implementing the more efficient OT extension of [KOS15].

Outline

We start by going through a high level description of our protocols in Section 4.1. We then give a detailed description of some of the primitives needed in Section 4.2. Then in Section 4.3 a fully detailed description of the most efficient of our two protocols is given and in Section 4.4 we give a full proof of security. Afterwards we discuss our implementations, by first giving a description of the overall structure of our computation device of choice: the GPU, in Section 4.5 and then our implementation details along with experimental results in Section 4.6.

4.1 Our Protocols

We now go through a high level description of the steps in our protocols. We first show the less efficient protocol of [FN13] and then discuss how to change it in order to get the more efficient protocol of [FJN14]. Assume we have access to a free-XOR garbling scheme, an OT functionality, a coin-tossing functionality along with a commitment scheme supporting both standard commitments (computationally hiding/binding) and verifiable commitments as defined in Chapter 2.

We let A garble ℓ versions of f using a binary key size preserving, structure free, free-XOR gate garbling scheme with the prv.ind and ver properties and leakage function $\mathbf{\Phi}_{\mathsf{xor}}$ as defined in Chapter 2. We call these garbled circuits F_j where $j \in [\ell]$. For each of these garbled circuits we associate a distinct global value $\Delta_j \in \{0,1\}^\kappa$, which is used to enforce the constraint that for any wire, i, in the garbled circuit F_j, we have that $K^1_{i,j} = K^0_{i,j} \oplus \Delta_j$ in correspondence with a free-XOR gate garbling scheme.

At an informal level our first protocol can then be summarized with the phases *Setup*, *Oblivious Transfer*, *Commitment*, *Cut-and-Choose*, and *Evaluation*. These are all very similar to other protocols based on cut-and-choose of garbled circuits, e.g., [LP15, LP12, SS11, SS13].

Setup The parties agree on the functionality to be computed, denoted by $f(x) = y$ where $x = x_{\mathsf{A}} \| x_{\mathsf{B}}$ with $x_{\mathsf{A}} \in \{0,1\}^{n_{\mathsf{A}}}$, $x_{\mathsf{B}} \in \{0,1\}^{n_{\mathsf{B}}}$ and $y \in \{0,1\}^m$. Let $\tilde{n_{\mathsf{B}}} = \max(4 \cdot (n+s-1), 8 \cdot s)$ and have B sample a random binary matrix $M^{\mathsf{Sec}} \in \{0,1\}^{(n+s-1) \times \tilde{n_{\mathsf{B}}}}$ and a random input $\tilde{x_{\mathsf{B}}}$ of $\tilde{n_{\mathsf{B}}}$ bits under the constraint that $M^{\mathsf{Sec}} \cdot \tilde{x_{\mathsf{B}}} = x_{\mathsf{B}} \| \beta$ where $\beta \in_R \{0,1\}^{n_{\mathsf{A}}+s-1}$. Furthermore, let A sample a uniformly random bitstring $\alpha \in_R \{0,1\}^s$ and let $\tilde{x_{\mathsf{A}}} = x_{\mathsf{A}} \| \alpha$

and similarly let $\tilde{n_\mathsf{A}} = n_\mathsf{A} + s$. Now, we have B send M^{Sec} to A and they both agree on a function, $\tilde{f}$ with M^{Sec} embedded such that

$$\tilde{f}(\tilde{x_\mathsf{A}} \| \tilde{x_\mathsf{B}}) = y \| \tau = f(x_\mathsf{A} \| x_\mathsf{B}) \| ((M^{\mathsf{In}} \cdot x_\mathsf{A}) \oplus \alpha) \ ,$$

where $\tau \in \{0,1\}^s$ and $M^{\mathsf{In}}[i,j] = \beta[i+j-1]$. That is, $\tilde{f}$ on $\tilde{x}$ computes the same output as f on x, but with the extra output τ.

Oblivious Transfer

1. For each $j \in \ell$ A constructs a garbled circuit computing $\tilde{f}$, denoted by $(F_j, e_j, d_j) \leftarrow \mathsf{Gb}\left(1^\kappa, \tilde{f}; r_j\right)$. Denote the global difference of the j'th garbled circuit by Δ_j.
2. A and B then engage in batch OT such that B learns the ℓ keys in correspondence with each of his input bits $\tilde{x_\mathsf{B}}$.

Commitment A sends the ℓ garbled circuits and their corresponding decoding information to B. She then commits to her own choice of input keys.

Cut-and-Choose

1. A and B use a coin-tossing protocol to agree on a random subset π of size $\ell/2$ of the garbled circuits, called the *check circuits*. A then sends the encoding information and the randomness used to construct each of these circuits to B.
2. B then verifies that the check circuits were correctly constructed using the verification function Ve.

Evaluation A opens the input keys in correspondence with her input $\tilde{x_\mathsf{A}}$ for all the evaluation circuits to B. B then evaluates the garbled circuits and decodes the output. If the τ part of the output is different in any of the evaluations then he aborts. Otherwise he takes the majority of occurring outputs to be the output of the computation.

Our second protocol is very similar to the first, except that we augment the function to be computed even more and add a few more phases: *Polynomial Setup*, *Augmentation* and *Reconstruction*. The *Polynomial Setup* is added to construct the polynomials we need to associate with the keys on the output wires. The *Augmentation* is added to augment the garbled circuit to ensure consistency of A's input and to make a sufficient amount of random output keys. Finally, the *Reconstruction* is added to make it possible for B to use these polynomials to reconstruct A's true input, if she tries to cheat. Thus our second protocol consists of the following phases *Setup*, *Polynomial Setup*, *Oblivious Transfer*, *Commitment*, *Augmentation*, *Cut-and-Choose*, *Evaluation* and *Reconstruction*. Which we now describe.

Setup The parties agree on the functionality to be computed, denoted by $f(x) = y$ where $x = x_\mathsf{A} \| x_\mathsf{B}$ with $x_\mathsf{A} \in \{0,1\}^{n_\mathsf{A}}$, $x_\mathsf{B} \in \{0,1\}^{n_\mathsf{B}}$ and $y \in \{0,1\}^m$. Let $\bar{n_\mathsf{B}} = \max(4 \cdot n_\mathsf{B}, 8 \cdot s)$ and have B sample a random binary matrix $M^{\mathsf{Sec}} \in \{0,1\}^{n_\mathsf{B} \times \bar{n_\mathsf{B}}}$ and a random input $\bar{x_\mathsf{B}}$ of $\bar{n_\mathsf{B}}$ bits under the constraint that $M^{\mathsf{Sec}} \cdot \bar{x_\mathsf{B}} = x_\mathsf{B}$. Furthermore, let A sample a uniformly random string $\alpha \in_R \{0,1\}^s$ and let $\bar{x_\mathsf{A}} = x_\mathsf{A} \| \alpha$ and similarly let $\bar{n_\mathsf{A}} = n_\mathsf{A} + s$. Now, we have B send M^{Sec} to A and they both agree on a function, $\bar{f}$ with M^{Sec} embedded such that $\bar{f}(\bar{x_\mathsf{A}} \| \bar{x_\mathsf{B}}) = y = f(x_\mathsf{A} \| M^{\mathsf{Sec}} \cdot \bar{x_\mathsf{B}})$ where $\bar{x} = \bar{x_\mathsf{A}} \| \bar{x_\mathsf{B}}$. That is, $\bar{f}$ on $\bar{x}$ computes the same output as f on x.

Polynomial Setup

1. A randomly selects $p = 6s + 7$ polynomials, denoted P_i for $i \in [p]$, of degree at most $\ell/2$ from the finite field $\mathbb{F}_{2^\kappa}$.
2. A computes ℓ points on each of the polynomials, achieving a total of $\ell \cdot (6s+7)$ points. She then commits to each point j on each polynomial i using a verifiable commitment scheme and sends these commitments to B.
3. A and B complete a cut-and-choose procedure on the random polynomials to select $\lfloor 1.18s + 2.18 \rfloor$ of these for *checking* and $\lceil 4.82s + 4.82 \rceil$ for *evaluation*. For each of the check polynomials A sends the openings to the points committed to in the previous step. B then uses the points to interpolate the polynomials and verifies that they are in fact all polynomials of degree at most $\ell/2$. If not, he aborts.

Oblivious Transfer

1. For each $j \in \ell$ A constructs a garbled circuit computing $\bar{f}$, denoted by $(F_j, e_j, d_j) \leftarrow^* \mathsf{Gb}\left(1^\kappa, \bar{f}\right)$. We furthermore denote the global difference of the j'th garbled circuit by Δ_j.
2. A and B then engage in batch OT such that B learns the ℓ keys in correspondence with each of his input bits $\bar{x_\mathsf{B}}$.

Commitment A sends the ℓ garbled circuits and their corresponding decoding information to B. She then commits to her own choice of input keys and also makes verifiable commitments to the concatenation of the 0- and 1-keys on each of her input wire in each of the garbled circuits.

Augmentation

1. B randomly samples a universal hash function $\mathbb{H}^{\mathsf{In}}$ from a family of universal hash functions mapping n_A bits to s bits and a universal hash function $\mathbb{H}^{\mathsf{Out}}$ mapping m bits to $\lceil 4.82s + 4.82 \rceil$ bits from another family of universal hash functions.
2. B sends a description of these hash functions to A who augments all of the garbled circuits with $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ and $\mathbb{H}^{\mathsf{Out}}$ using only XOR operations of respectively her input keys and the output keys of the garbled circuits. Where $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ means that $\mathbb{H}^{\mathsf{In}}$ is computed on the first n_A bits of A's input and the result of this is then XOR'ed with the last s bits of A's input.
3. A then makes verifiable commitments to the 0- and 1-output keys of the $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ augmentation. A also constructs, and sends, a "link" of each of the output keys of $\mathbb{H}^{\mathsf{Out}}$ to the corresponding polynomial points.

Cut-and-Choose

1. A and B use a coin-tossing protocol to agree on a random subset π of size $\ell/2$ of the garbled circuits, called the *check circuits*. A then sends the encoding information and randomness used to construct each of the check circuits.
2. B then uses the encoding information to check the verifiable commitments to concatenation of the 0- and 1-keys of A's input wires. He then verifies the check circuits were correctly constructed using the verification function Ve.

3. B then verifies the $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ augmentation by using the input keys to compute the output keys of the augmentation locally and then checking the output keys against the verifiable commitments.
4. Finally, he computes the output keys of $\mathbb{H}^{\mathsf{Out}}$, using the output keys of the garbled circuits. He then uses these keys on the "links" from the augmentation phase to learn the $\ell/2$ points on each of the $\lceil 4.82s + 4.82 \rceil$ polynomials remaining from the set-up phase. He then checks these points using the verifiable commitments. If any of the checks above fail he aborts.

Evaluation A opens the input keys in correspondence with her input $\bar{x_{\mathsf{A}}}$ for all the evaluation circuits to B. B then evaluates $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$. If the semantic value of any of the evaluations of $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ are discrepant, or if any of the verifiable commitments on the output keys of this augmentation are incorrectly constructed, he aborts. Otherwise he evaluates the garbled circuits. He then uses the verifiable commitments on the output wires of the garbled circuits to find the semantic meaning of the output keys he learns. If any of the semantic values on the output wires of the evaluated garbled circuits are inconsistent then he proceeds with the *Reconstruction* phase. Otherwise he "simulates" the *Reconstruction* phase using garbage data (to avoid timing attacks) and in the end outputs the only semantic output value he learned.

Reconstruction

1. B computes the augmented output $\mathbb{H}^{\mathsf{Out}}$ for each of the evaluated garbled circuits. Since he now knows the semantic value on each garbled circuit's output wires, along with the specification of $\mathbb{H}^{\mathsf{Out}}$, he can compute, in plain, the semantic values he expects to find after evaluating $\mathbb{H}^{\mathsf{Out}}$. Any augmented circuit where this is not the case he discards. From the remaining circuits he computes the points on the remaining polynomials from the *Polynomial Setup* phase using the links from the *Augmentation* phase. He then uses the $\ell/2$ polynomial points from the cut-and-choose phase along with the discrepant keys of an augmented output wire in two circuits to learn $\ell/2+1$ points on a polynomial for a given wire. He then does polynomial interpolation on these points to find all the ℓ points of this wire, i.e. one for each garbled circuit. He then uses the links to find the 0-key on that wire in all of the evaluation circuits. Hence B can learn Δ_j for at least one garbled circuit (because the output of that wire is discrepant and thus at least one garbled circuit will have 1-key output on that wire). He continues in this manner for each discrepant output wire until he learns Δ_j for all the non-discarded evaluation circuits.
2. Using these Δ's along with A's input keys and the verifiable commitments to the keys of the input wires, B extracts the plain values of A's input in all the evaluation circuits. For each input, B evaluates $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ on the plain input and discards the circuits where the plain output from $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ does not match the semantic output previously computed by the garbled circuit. He then uses A's plain input for one of the remaining circuits and computes the function f in plain to learn the correct output.[1] If there are no remaining circuits then B terminates.

[1] Note that to avoid selective failure attacks, B must not reveal to A whether the output was obtained from this step or from the above *Evaluation* phase.

4.2 Building Blocks

The OT Extension

We use the approach from [NNOB12] as the base of our OT extension. However, we make a few modifications to reduce as many operations as possible to parallel computable hashes of short bitstrings.

Assuming the existence of random oracles and a secure implementation of a κ-bit 1-out-of-2 OT as an ideal resource, the protocol is UC secure against a malicious adversary. For the rest of this section we let $\bar{n_{\mathsf{B}}}$ be the amount of bits in B's input for the augmented circuit.

On a high level the OT extension can be explained by the follow steps when A wishes to obliviously transfer one of two possible strings to B:

Setup A samples $\Theta(\kappa)$ random bits and B samples $\Theta(\kappa)$ pairs of random seeds of κ bits each.

Seed OT The parties execute $\Theta(\kappa)$ random OTs using their seeds, in *reverse* order. That is, B acts as the sender, using as input to each of the OTs his pairs of seeds and A acts as the receiver, using one of her random bits as input to each OT.

Extension The parties use a hash function, $\mathsf{H}(\cdot)$, to extend the seeds to long pseudorandom strings. That is, B has $\Theta(\kappa)$ pairs of seeds which he extends and A has $\Theta(\kappa)$ seeds which she extends.

Adjustment For each of the seed OTs B computes a string that is the XOR of his true input bits and both the 0 and 1 choice of each extended OTs. He sends these strings to A. She XOR's each of these strings onto each of the extended strings she learned, but only in the cases where her choice bit was 1.

Verification The parties do a verification step to ensure that B XOR'ed the same choices onto each of the adjustments he sent in *Adjustment*. This involves sacrificing half of the seed OTs.

Tilt-your-head The parties "tilt their heads" to reverse the role of sender and receiver. That is, viewing each of the OTs as a row in a matrix, they simply transpose the matrix. B, however only does this for the string of each OT corresponding to the 0 choice.

Privacy Amplification There is the potential of leaked information so the parties hash each row of the transposed matrices and let each row of the matrices define a new, random, OT.

More formally the modified OT extension goes as follows:

Setup B chooses $\left\lceil\frac{8}{3}\kappa\right\rceil$ pairs of seeds, each consisting of κ random bits. That is, for each $i \in \left[\left\lceil\frac{8}{3}\kappa\right\rceil\right]$ we let $(l_i^0, l_i^1) \in_R \{0,1\}^\kappa \times \{0,1\}^\kappa$ be the i'th seed pair. A samples a $\left\lceil\frac{8}{3}\kappa\right\rceil$ random bit string, $\Gamma \in_R \{0,1\}^{\left\lceil\frac{8}{3}\kappa\right\rceil}$.

Seed OT A and B then run $\left\lceil\frac{8}{3}\kappa\right\rceil$ seed OTs where, for $i \in \left[\left\lceil\frac{8}{3}\kappa\right\rceil\right]$, Bob offers (l_i^0, l_i^1) and Alice selects $\Gamma[i]$, and receives $l_i^{\Gamma[i]}$.

Extension For each of the $i \in \left[\left\lceil \frac{8}{3}\kappa \right\rceil\right]$ pairs of random bits Bob computes the following two vectors of $\bar{x_{\mathsf{B}}}$ bits, using $\mathsf{ID}_{i,j}$ as a unique ID:

$$L_i^0 = \mathsf{H}\left(l_i^0 \| \mathsf{ID}_{i,0}\right) \| \mathsf{H}\left(l_i^0 \| \mathsf{ID}_{i,1}\right) \| \ldots \| \mathsf{H}\left(l_i^0 \| \mathsf{ID}_{i,\bar{n_{\mathsf{B}}}/\kappa}\right),$$
$$L_i^1 = \mathsf{H}\left(l_i^1 \| \mathsf{ID}_{i,0}\right) \| \mathsf{H}\left(l_i^1 \| \mathsf{ID}_{i,1}\right) \| \ldots \| \mathsf{H}\left(l_i^1 \| \mathsf{ID}_{i,\bar{n_{\mathsf{B}}}/\kappa}\right) .$$

In the same manner A extends each of her outputs of the OT from their original length of κ bits, into strings of $\bar{n_{\mathsf{B}}}$ bits. Thus, Alice computes

$$L_i^{\Gamma[i]} = \mathsf{H}\left(l_i^{\Gamma[i]} \| \mathsf{ID}_{i,0}\right) \| \mathsf{H}\left(l_i^{\Gamma[i]} \| \mathsf{ID}_{i,1}\right) \| \ldots \| \mathsf{H}\left(l_i^{\Gamma[i]} \| \mathsf{ID}_{i,\bar{n_{\mathsf{B}}}/\kappa}\right) .$$

Adjustment

1. Now, for each $i \in \left[\left\lceil \frac{8}{3}\kappa \right\rceil\right]$ B computes a bitstring, $\lambda_i = L_i^0 \oplus L_i^1 \oplus \bar{x_{\mathsf{B}}}$, and sends these to A.
2. For each $i \in \left[\left\lceil \frac{8}{3}\kappa \right\rceil\right]$ A computes a bitstring as follows

$$L'^{\Gamma[i]}_i = L_i^{\Gamma[i]} \oplus (\Gamma[i] \cdot \lambda_i) = L_i^0 \oplus (\Gamma[i] \cdot \bar{x_{\mathsf{B}}}).$$

Verification A then picks a uniformly random permutation

$$\pi : \left[\left\lceil \frac{8}{3}\kappa \right\rceil\right] \to \left[\left\lceil \frac{8}{3}\kappa \right\rceil\right] ,$$

where for all i it holds that $\pi(\pi(i)) = i$, and sends these to B. Furthermore, define $S(\pi) = \{i | i \leq \pi(i)\}$, that is, for each pair, the smallest index is in $S(\pi)$. For all the $\left\lceil \frac{4}{3}\kappa \right\rceil$ indexes $i \in S(\pi)$ do the following:

1. A computes $d_i = \Gamma[i] \oplus \Gamma[\pi(i)]$ and sends these to B.
2. A and B both compute $Z_i = \left(L'^{\Gamma[i]}_i \oplus L'^{\Gamma[\pi(i)]}_{\pi(i)}\right)$. This is possible for B since d_i uniquely determines the way to compute Z_i, i.e. if he should XOR L_i^0 with $\bar{x_{\mathsf{B}}}$.

For all $i \in S(\pi)$, A and B concatenate the Z_i strings, call A's result Z^A and B's result Z^B. They then check that $Z^A = Z^B$ using the following subprotocol:

1. A chooses a random string $r \in_R \{0,1\}^\kappa$.
2. She then computes $\mathsf{H}(Z^A \| r) \to c$ and sends this to B.
3. B then sends Z^B to A, who checks that $Z^A = Z^B$, if this is so she then sends Z^A and r to B.
4. B then computes $\mathsf{H}(Z^A \| r) \to c'$.
5. B then checks if $c' = c$ and that $Z^A = Z^B$.
6. If all checks are successful then the strings are equal, and the protocol continues, otherwise the parties abort.

Tilt-your-head For each $i \in \left[\left\lceil \frac{4}{3}\kappa \right\rceil\right]$ and for each $j \in [n_{\mathsf{B}}]$ A defines $\bar{L}_j$ to be the string consisting of the j'th bits from all the strings ${L'}_i^{\Gamma[i]}$, i.e.

$$\bar{L}_j = {L'}_1^{\Gamma[1]}[j] \| {L'}_2^{\Gamma[2]}[j] \| \dots \| {L'}_{\lceil \frac{4}{3}\kappa \rceil}^{\Gamma[\lceil \frac{4}{3}\kappa \rceil]}[j] \ .$$

This means that she gets $\bar{n_{\mathsf{B}}}$ keys consisting of $\left\lceil \frac{4}{3}\kappa \right\rceil$ bits. For each $i \in \left[\left\lceil \frac{4}{3}\kappa \right\rceil\right]$ and for each $j \in [\bar{n_{\mathsf{B}}}]$ B sets $\tilde{L}_j$ to be the string consisting of the j'th bits from all the strings L_i^0, i.e.

$$\tilde{L}_j = L_2^0[j] \| L_2^0[j] \| \dots \| L_{\lceil \frac{4}{3}\kappa \rceil}^0[j] \ .$$

Privacy amplification

1. A lets $\bar{\Gamma}$ be the string consisting of all the bits $\Gamma[i]$ for $i \in S(\pi)$, i.e. for $i \in \left[\left\lceil \frac{4}{3}\kappa \right\rceil\right]$ we have $\bar{\Gamma}[i] = \Gamma[S(\pi)]$.
2. B now computes $X_j = \mathsf{H}\left(\tilde{L}_j\right)$ and achieves $\{X_0, \dots, X_{\bar{x_{\mathsf{B}}}}\}$.
3. A computes $X_j^0 = \mathsf{H}\left(\bar{L}_j\right)$ and $X_j^1 = \mathsf{H}\left(\bar{L}_j \oplus \bar{\Gamma}\right)$ and achieves $\left\{ \left(X_1^0, X_1^1\right), \dots, \left(X_{\bar{n_{\mathsf{B}}}}^0, X_{\bar{n_{\mathsf{B}}}}^1\right)\right\}$.

If the parties have been honest it should be the case that for each $j \in [\bar{n_{\mathsf{B}}}]$ we have $X_j = X_j^{\bar{x_{\mathsf{B}}}[j]}$.

Security of the Modified OT Extension.

The overall correctness and security of the modified OT extension follow from the correctness and security of the original OT extension [NNOB12] (which is UC-secure in the random-oracle, OT-hybrid model). However, we do make a few changes to this protocol. In the following we will specify these changes and sketch why they do not compromise the security of the protocol. The changes between the modified OT extension and the OT extension from [NNOB12] are the non-random construction of $\bar{x_{\mathsf{B}}}$ and the use of hashing to construct the strings L_i^0, L_i^1. In [NNOB12] $\bar{x_{\mathsf{B}}}$ needs to be picked uniformly random. The OTs could then later be adjusted to specific choices, both in regards to the message content and the choice of whether B should learn message 0 or message 1. It is furthermore the case that in [NNOB12] the seeds are extended through a PRG instead of a random oracle.

Non-random $\bar{x_{\mathsf{B}}}$. We need $\bar{x_{\mathsf{B}}}$ to be non-random in order for B's output of the OT extension to be consistent with his input bits for the garbled circuits. We do this as part of the OT protocol in order to eliminate the need for sending a "padding" which would be the normal approach in order to change random OTs into OTs of specific bitstrings and choices. By embedding this in the extension itself we save some rounds of communication and make the whole OT extension phase simpler by eliminating these post processing steps. Intuitively it does not compromise the security for B as $\bar{x_{\mathsf{B}}}$ is one-time-padded with the random strings L_i^0 and L_i^1 in *Adjustment*. In fact, whether $\bar{x_{\mathsf{B}}}$ is random or not, the string sent to A will still be random, as one of the elements (either L_i^0 or L_i^1) will at this point be unknown to her, and since they are random so will the string λ_i be in her view. It does not compromise the security for A either, because A's vector Γ is random, and thus that B has no idea if $\bar{x_{\mathsf{B}}}$ will be XORed onto a ${L'}_i^{\Gamma[i]}$ in the *Adjustment* steps.

Construction of L_i^0 and L_i^1. In [NNOB12] a pseudorandom generator is used to extend a random string of length κ to a pseudorandom string of length $\bar{n_{\mathsf{B}}}$. However, our approach is based on invocations of hash functions on a common seed concatenated with a unique ID. This is clearly secure in the random-oracle model.

Links

We use a technical tool which we call *links*. A *link scheme* is a pair (G, F) of PPT algorithms. The *link generator* G takes as input two values $J, K \in \{0,1\}^\kappa$, called the *joints*, and outputs a *link* $L \in \{0,1\}^*$. The *follower* allows to take one joint of a link and compute the other joint. To be more precise, for all $J, K \in \{0,1\}^\kappa$ and $L \leftarrow^* \mathsf{G}(J, K)$ it holds with probability 1 that $\mathsf{F}(\texttt{forwards}, L, J) = K$ and $\mathsf{F}(\texttt{backwards}, L, K) = J$.

As for security we want that a link leaks no other information than the ability to compute one joint from the other. In particular, to a party which does not know how to compute J nor K, the link should not reveal any information on J or K. We will model this by requiring that an adversary who cannot compute J nor K cannot distinguish a link of J and K from a link of any two other values.

Definition 12. We call a distribution $(J_1, K_1, J_2, K_2, \ldots, J_\ell, K_\ell, \zeta) \leftarrow D$ on $(\{0,1\}^\kappa)^{2\ell} \times \{0,1\}^*$ with $\ell \in \text{poly}(\kappa)$ *hard* if it holds for all PPT $\mathcal{B}$ that it wins the following game with negligible probability in κ: First sample $(J_1, K_1, J_2, K_2, \ldots, J_\ell, K_\ell, \zeta) \leftarrow D$ and give ζ to $\mathcal{B}$. Run $\mathcal{B}$ to produce $S \subset \{0,1\}^*$, where the set is represented by a list of its elements. Then $\mathcal{B}$ wins iff $J_i \in S$ or $K_i \in S$ for some i. We call a class $\mathcal{D}$ of distributions hard if all $D \in \mathcal{D}$ are hard. We call (G, F) a *secure link scheme* for $\mathcal{D}$ if it holds for all PPT $\mathcal{A}$ and all distributions $D \in \mathcal{D}$ that $\mathcal{A}$ wins the following game with probability negligibly close to $\frac{1}{2}$: Sample $(J_1, K_1, J_2, K_2, \ldots, J_\ell, K_\ell, \zeta) \leftarrow D$ and give ζ to $\mathcal{A}$. Sample uniformly random $(J_1', K_1', J_2', K_2', \ldots, J_\ell', K_\ell') \in (\{0,1\}^\kappa)^{2\ell}$. Sample a uniformly random bit $b \in_R \{0,1\}$. If $b = 0$, let $(J_1'', K_1'', J_2'', K_2'', \ldots, J_\ell'', K_\ell'') = (J_1, K_1, J_2, K_2, \ldots, J_\ell, K_\ell)$. Otherwise, let $(J_1'', K_1'', J_2'', K_2'', \ldots, J_\ell'', K_\ell'') = (J_1', K_1', J_2', K_2', \ldots, J_\ell', K_\ell')$. For $i \in [\ell]$, sample $L_i \leftarrow^* \mathsf{G}(1^\kappa, J_i'', K_i'')$. Input $(L_1, \ldots, L_\ell)$ to $\mathcal{A}$. Run $\mathcal{A}$ to produce a guess b'. The adversary wins iff $b' = b$.

We can build a link scheme from a hash function $\mathsf{H} : (\{0,1\}^\kappa)^2 \to \{0,1\}^\kappa$ as follows: On input $\mathsf{G}(J, K)$, sample uniformly random salts $r_1, r_2 \in_R \{0,1\}^\kappa$. Then output $L = (r_1, r_2, h_1, h_2) = (r_1, r_2, \mathsf{H}(J, r_1) \oplus K, \mathsf{H}(K, r_2) \oplus J)$. On input $\mathsf{F}(\texttt{forwards}, (r_1, r_2, h_1, h_2), J)$, output $\mathsf{H}(J, r_1) \oplus h_1 = K$. Similarly for the other direction, i.e. $\mathsf{F}(\texttt{backwards}, (r_1, r_2, h_1, h_2), K) = \mathsf{H}(K, r_2) \oplus h_2 = J$.

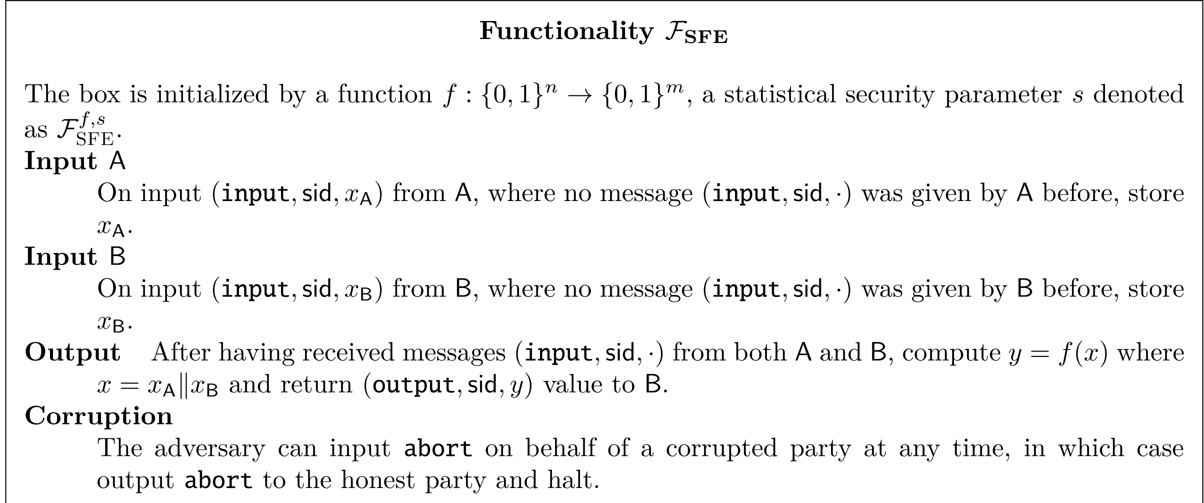
It is straight forward to prove the above scheme secure in the random oracle model for the class of *all* hard distributions. Namely, until an adversary $\mathcal{A}$ queries some $\mathsf{H}(\cdot, r_{1,i})$ or $\mathsf{H}(\cdot, r_{2,i})$, the value $(r_{1,i}, r_{2,i}, \mathsf{H}(J_i, r_{1,i}) \oplus K_i, \mathsf{H}(K_i, r_{2,i}) \oplus J_i)$ is uniformly random in its view. So, turn it into an adversary $\mathcal{B}$ by giving it uniformly random values $(r_{1,i}, r_{2,i}, h_{1,i}, h_{2,i})$ as input, and then add each J_i queried to some $\mathsf{H}(\cdot, r_{1,i})$ or $\mathsf{H}(\cdot, r_{2,i})$ to the set S, and then at the end output S.

Some Sub-Functionalities and Their Implementations

Before we can formally specify our protocol we need to introduce some ideal functionalities. We then describe some of the ideal functionalities that are used to realize our protocol in the hybrid model, and show how to realize them.

Secure Two-Party Function Evaluation

In Fig. 4.1 our ideal functionality for secure function evaluation is presented. In particular notice that, as discussed early, it reflects that B is the only party learning output.



Functionality $\mathcal{F}_{\mathbf{SFE}}$

The box is initialized by a function $f : \{0,1\}^n \to \{0,1\}^m$, a statistical security parameter s denoted as $\mathcal{F}_{\text{SFE}}^{f,s}$.

Input A
On input $(\texttt{input}, \mathsf{sid}, x_{\mathsf{A}})$ from A, where no message $(\texttt{input}, \mathsf{sid}, \cdot)$ was given by A before, store x_{A}.

Input B
On input $(\texttt{input}, \mathsf{sid}, x_{\mathsf{B}})$ from B, where no message $(\texttt{input}, \mathsf{sid}, \cdot)$ was given by B before, store x_{B}.

Output After having received messages $(\texttt{input}, \mathsf{sid}, \cdot)$ from both A and B, compute $y = f(x)$ where $x = x_{\mathsf{A}} \| x_{\mathsf{B}}$ and return $(\texttt{output}, \mathsf{sid}, y)$ value to B.

Corruption
The adversary can input `abort` on behalf of a corrupted party at any time, in which case output `abort` to the honest party and halt.

Figure 4.1: The ideal functionality, $\mathcal{F}_{\text{SFE}}$, for secure function evaluation for two parties. The box reflects that only B gets output.

Single Choice Batch Oblivious Transfer

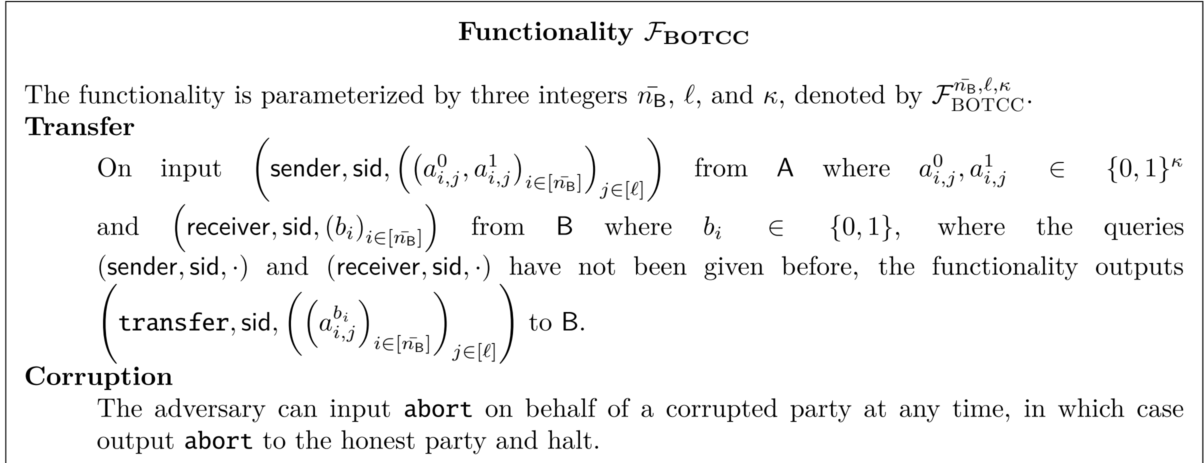
For each circuit that A garbles, oblivious transfers must be carried out that lets B receive one key for each of his input wires without revealing to A if he asked for a 0- or 1-key.

We here consider a functionality that allows $\bar{n_{\mathsf{B}}} \cdot \ell$ OTs to be carried out in one "batch". In addition, while the functionality does not make any restrictions on the keys A inputs except that they are of κ bits, it does force B to use the same selection bit for all of the ℓ circuits. For this reason we will refer to the functionality as *batch oblivious transfer with consistent choice*. We denote it $\mathcal{F}_{\text{BOTCC}}^{\bar{n_{\mathsf{B}}},\ell,\kappa}$ and it is outlined in Fig. 4.2. Note that $\mathcal{F}_{\text{BOTCC}}^{\bar{n_{\mathsf{B}}},\ell,\kappa}$ should not be confused with the batch single choice cut-and-choose functionality used in [LP12] that also integrates a cut-and-choose step.

Given access to a random oracle, Fig. 4.3 shows how to realize $\mathcal{F}_{\text{BOTCC}}^{\bar{n_{\mathsf{B}}},\ell,\kappa}$ efficiently in the $\mathcal{F}_{\text{OT}}$-hybrid model in the presence of a malicious and static adversary. The ideal functionality of $\mathcal{F}_{\text{OT}}$ is shown in Fig. 2.5 in Chapter 2 and can for example be realized as described in [PVW08].[2]

The protocol works by A picking $\bar{n_{\mathsf{B}}}$ pairs of κ random seeds (r_i^0, r_i^1 for $i \in [\bar{n_{\mathsf{B}}}]$) which are then used as input to $\bar{n_{\mathsf{B}}}$ instances of $\mathcal{F}_{\text{OT}}^{\kappa}$. For these B gives as input one of his input bits, b_i. Both A and B extend each of the seeds to strings of $\kappa \cdot \ell$ bits. A will then use the extended seeds as one-time pads of her true input and send a pair of "correction" values to B which he can use to learn the outputs he is supposed to, but only for one of the messages per OT.

[2] We could always realize the functionality by $\bar{n_{\mathsf{B}}}$ repeated executions of $\mathcal{F}_{\text{OT}}$ where each message was a string of $\ell \cdot \kappa$ bits. However, depending on the protocol used to realize $\mathcal{F}_{\text{OT}}$ this could be quite inefficient as group operations on group elements of at least $\ell \cdot \kappa$ bits might be needed.



Functionality $\mathcal{F}_{\mathbf{BOTCC}}$

The functionality is parameterized by three integers $\bar{n_\mathsf{B}}$, ℓ, and κ, denoted by $\mathcal{F}_{\text{BOTCC}}^{\bar{n_\mathsf{B}},\ell,\kappa}$.

Transfer

On input $\left(\mathsf{sender}, \mathsf{sid}, \left(\left(a^0_{i,j}, a^1_{i,j}\right)_{i\in[\bar{n_\mathsf{B}}]}\right)_{j\in[\ell]}\right)$ from A where $a^0_{i,j}, a^1_{i,j} \in \{0,1\}^\kappa$ and $\left(\mathsf{receiver}, \mathsf{sid}, (b_i)_{i\in[\bar{n_\mathsf{B}}]}\right)$ from B where $b_i \in \{0,1\}$, where the queries $(\mathsf{sender}, \mathsf{sid}, \cdot)$ and $(\mathsf{receiver}, \mathsf{sid}, \cdot)$ have not been given before, the functionality outputs $\left(\mathtt{transfer}, \mathsf{sid}, \left(\left(a^{b_i}_{i,j}\right)_{i\in[\bar{n_\mathsf{B}}]}\right)_{j\in[\ell]}\right)$ to B.

Corruption

The adversary can input `abort` on behalf of a corrupted party at any time, in which case output `abort` to the honest party and halt.

Figure 4.2: The ideal functionality $\mathcal{F}_{\text{BOTCC}}^{\bar{n_\mathsf{B}},\ell,\kappa}$ (with ℓ circuits, $\bar{n_\mathsf{B}}$ input bits from B, and keys of κ bits).

Theorem 5. *Assuming the hash function $\mathsf{H}(\cdot)$ has κ bits output and is a random oracle then the protocol Π_{BOTCC} in Fig. 4.3 UC-securely realizes the $\mathcal{F}_{\text{BOTCC}}$ functionality in the $\mathcal{F}_{\text{OT}}$-hybrid model against any PPT static and malicious adversary.*

Proof. We start by considering that A (sender) is corrupted and thus under the control of the adversary, which we denote by $\mathcal{A}$. We now describe a simulator S_A simulating the interaction between $\mathcal{A}$ and an honest B (receiver), using the functionality $\mathcal{F}_{\text{BOTCC}}^{\bar{n_\mathsf{B}},\ell,\kappa}$:

1. S_A gets the messages $(\mathsf{sender}, (\mathsf{sid}, i), r^0_i, r^1_i)$ for $i \in \bar{n_\mathsf{B}}$ where $r^0_i, r^1_i \in \{0,1\}^\kappa$ from $\mathcal{A}$'s calls to $\mathcal{F}^\kappa_{\text{OT}}$ and returns to $\mathcal{A}$ the acknowledgment message (sid, i) for $i \in [\bar{n_\mathsf{B}}]$.

2. S_A then computes for $i \in [\bar{n_\mathsf{B}}]$ the pairs:
$$\left(r'^0_{i,j}, r'^1_{i,j}\right) = \left(\mathsf{H}\left(r^0_i \| \mathsf{ID}_{i,j}\right), \mathsf{H}\left(r^1_i \| \mathsf{ID}_{i,j}\right)\right) \in \{0,1\}^\kappa \times \{0,1\}^\kappa \ .$$

3. Next S_A receives the differences needed to change the extended inputs into $\mathcal{A}$'s "true" input. That is, the pairs $\left(\delta^0_{i,j}, \delta^1_{i,j}\right)$ for $i \in [\bar{n_\mathsf{B}}]$ and $j \in [\ell]$.

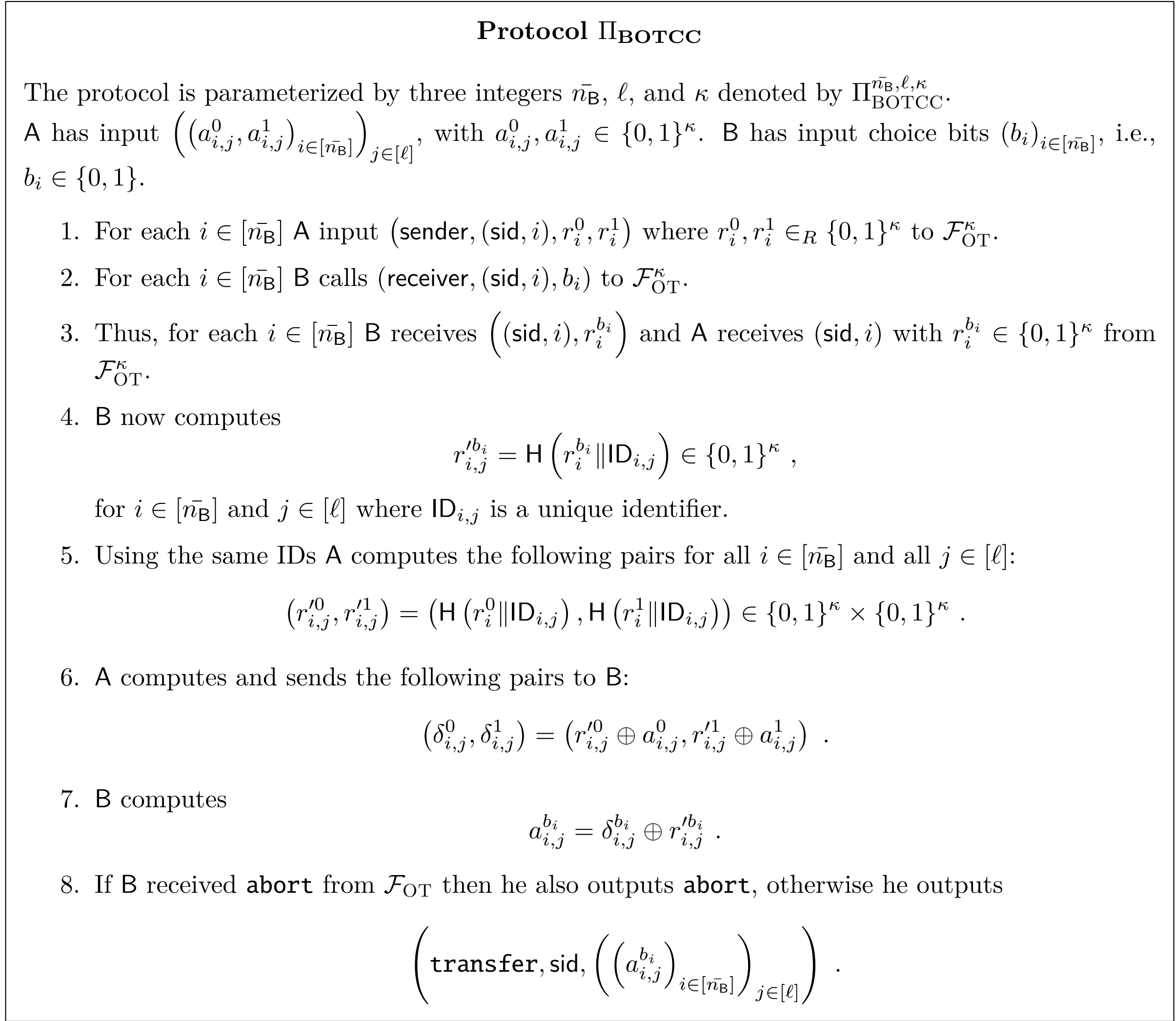
4. S_A then computes $\left(a^0_{i,j}, a^1_{i,j}\right) = \left(\delta^0_{i,j} \oplus r'^0_{i,j}, \delta^1_{i,j} \oplus r'^1_{i,j}\right)$.

5. S_A inputs $\left(\mathsf{sender}, \mathsf{sid}, \left(\left(a^0_{i,j}, a^1_{i,j}\right)_{i\in[\bar{n_\mathsf{B}}]}\right)_{j\in[\ell]}\right)$ to the functionality $\mathcal{F}_{\text{BOTCC}}^{\bar{n_\mathsf{B}},\ell,\kappa}$.

6. If $\mathcal{A}$ ever inputs `abort` to $\mathcal{F}^\kappa_{\text{OT}}$ or sends messages of incorrect format then S_A inputs `abort` to $\mathcal{F}_{\text{BOTCC}}^{\bar{n_\mathsf{B}},\ell,\kappa}$.

It is easy to see that the real and ideal executions are indistinguishable. First notice that in both the real (hybrid) and ideal (simulated) world B outputs a list of random elements $\left(a^{b_i}_{i,j}\right)$ in correspondence with $\mathcal{A}$'s input $a^0_{i,j}$ when $b_i = 0$ and $a^1_{i,j}$ when $b_i = 1$. Next see that all the messages that S_A gives to $\mathcal{A}$ are from the same distribution as the ones in the real protocol execution with an honest B.

Now consider a corrupt B, controlled by the adversary, denoted by $\mathcal{B}$. We now describe a simulator S_B for the execution:



Protocol $\Pi_{\mathbf{BOTCC}}$

The protocol is parameterized by three integers $\bar{n_{\mathsf{B}}}$, ℓ, and κ denoted by $\Pi^{\bar{n_{\mathsf{B}}},\ell,\kappa}_{\text{BOTCC}}$.
A has input $\left(\left(a^0_{i,j}, a^1_{i,j}\right)_{i\in[\bar{n_{\mathsf{B}}}]}\right)_{j\in[\ell]}$, with $a^0_{i,j}, a^1_{i,j} \in \{0,1\}^\kappa$. B has input choice bits $(b_i)_{i\in[\bar{n_{\mathsf{B}}}]}$, i.e., $b_i \in \{0,1\}$.

1. For each $i \in [\bar{n_{\mathsf{B}}}]$ A input $\left(\mathsf{sender}, (\mathsf{sid}, i), r^0_i, r^1_i\right)$ where $r^0_i, r^1_i \in_R \{0,1\}^\kappa$ to $\mathcal{F}^\kappa_{\text{OT}}$.
2. For each $i \in [\bar{n_{\mathsf{B}}}]$ B calls $(\mathsf{receiver}, (\mathsf{sid}, i), b_i)$ to $\mathcal{F}^\kappa_{\text{OT}}$.
3. Thus, for each $i \in [\bar{n_{\mathsf{B}}}]$ B receives $\left((\mathsf{sid}, i), r^{b_i}_i\right)$ and A receives (sid, i) with $r^{b_i}_i \in \{0,1\}^\kappa$ from $\mathcal{F}^\kappa_{\text{OT}}$.
4. B now computes
$$r'^{b_i}_{i,j} = \mathsf{H}\left(r^{b_i}_i \| \mathsf{ID}_{i,j}\right) \in \{0,1\}^\kappa ,$$
for $i \in [\bar{n_{\mathsf{B}}}]$ and $j \in [\ell]$ where $\mathsf{ID}_{i,j}$ is a unique identifier.
5. Using the same IDs A computes the following pairs for all $i \in [\bar{n_{\mathsf{B}}}]$ and all $j \in [\ell]$:
$$\left(r'^0_{i,j}, r'^1_{i,j}\right) = \left(\mathsf{H}\left(r^0_i \| \mathsf{ID}_{i,j}\right), \mathsf{H}\left(r^1_i \| \mathsf{ID}_{i,j}\right)\right) \in \{0,1\}^\kappa \times \{0,1\}^\kappa .$$
6. A computes and sends the following pairs to B:
$$\left(\delta^0_{i,j}, \delta^1_{i,j}\right) = \left(r'^0_{i,j} \oplus a^0_{i,j}, r'^1_{i,j} \oplus a^1_{i,j}\right) .$$
7. B computes
$$a^{b_i}_{i,j} = \delta^{b_i}_{i,j} \oplus r'^{b_i}_{i,j} .$$
8. If B received `abort` from $\mathcal{F}_{\text{OT}}$ then he also outputs `abort`, otherwise he outputs
$$\left(\mathtt{transfer}, \mathsf{sid}, \left(\left(a^{b_i}_{i,j}\right)_{i\in[\bar{n_{\mathsf{B}}}]}\right)_{j\in[\ell]}\right) .$$

Figure 4.3: The protocol Π_{BOTCC} realizing $\mathcal{F}_{\text{BOTCC}}$ in the $\mathcal{F}_{\text{OT}}$-hybrid, random oracle model.

1. S_B gets the messages $(\mathsf{receiver}, (\mathsf{sid}, i), b_i)$ for $i \in \bar{n_{\mathsf{B}}}$ where $b_i \in \{0,1\}$ from $\mathcal{B}$'s calls to $\mathcal{F}^\kappa_{\text{OT}}$.

2. S_B then picks $\bar{n_{\mathsf{B}}}$ uniformly random seeds $r^{b_i}_i \in_R \{0,1\}^\kappa$ and returns $\left((\mathsf{sid}, i), r^{b_i}_i\right)$ for $i \in [\bar{n_{\mathsf{B}}}]$ to $\mathcal{B}$.

3. It then extends these seeds using the random oracle:
$$r'^{b_i}_{i,j} = \mathsf{H}\left(r^{b_i}_i \| \mathsf{ID}_{i,j}\right) \in \{0,1\}^\kappa .$$

4. S_B then inputs $\left(\mathsf{receiver}, \mathsf{sid}, (b_i)_{i\in[\bar{n_{\mathsf{B}}}]}\right)$ to $\mathcal{F}^{\bar{n_{\mathsf{B}}},\ell,\kappa}_{\text{BOTCC}}$ and receives back $\left(\mathtt{transfer}, \mathsf{sid}, \left(\left(a^{b_i}_{i,j}\right)_{i\in[\bar{n_{\mathsf{B}}}]}\right)_{j\in[\ell]}\right)$.

5. It then computes $\delta_{i,j}^{b_i} = a_{i,j}^{b_i} \oplus r_{i,j}'^{b_i}$ and samples $\delta_{i,j}^{1-b_i}$ uniformly at random from $\{0,1\}^\kappa$ for $i \in [\bar{n_\mathsf{B}}]$ and $j \in [\ell]$. It sends the pairs $\left(\delta_{i,j}^0, \delta_{i,j}^1\right)$ for $i \in [\bar{n_\mathsf{B}}]$ and $j \in [\ell]$ to $\mathcal{B}$.

Again it is easy to see that the real and ideal executions are indistinguishable. Again $\mathcal{B}$ will receive the data from the same distribution in both the real and simulated case. In particular notice that $r_{i,j}'^{b_i} \oplus \delta_{i,j}^{b_i} = a_{i,j}^{b_i}$ like in the real world. For $\delta_{i,j}^{1-b_i}$ we notice that in the simulation it is uniformly random distributed over $\{0,1\}^\kappa$, whereas in the real world it is the XOR of $a_{i,j}^{1-b_i}$ and $\mathsf{H}\left(r_{i,j}^{1-b_i} \| \mathsf{ID}_{i,j}\right)$. However, since H is a random oracle this will be a uniformly random value. The only way that $\mathcal{B}$ will be able to distinguish the simulation and the real world is if he queries $\mathsf{H}\left(r_{i,j}^{1-b_i} \| \mathsf{ID}_{i,j}\right)$. However, he never learns anything in relation to $r_{i,j}^{1-b_i}$ because it is otherwise only used in $\mathcal{F}_{\mathrm{OT}}^{\kappa}$, which is ideal. □

If B has many bits of input to the functionality he wishes to compute with A, i.e., if $\bar{n_\mathsf{B}}$ is larger than $c\kappa$ for some constant c, then we can use an *OT extension* to limit the amount of heavy cryptographic operations needed to realize the $\mathcal{F}_{\mathrm{BOTCC}}^{\bar{n_\mathsf{B}},\ell,\kappa}$ functionality.

Secret Sharing

For our protocol we use a variation of Shamir's Secret Sharing scheme [Sha79] in order to make it possible to reconstruct keys the evaluator is not in possession of. We will secret share a key of κ bits in blocks of length l bits (we use $l = 8$ in our implementation). With this in hand we describe the simple polynomial interpolation methods we will use in our protocol in Fig. 4.4.

Initialized with the same parameters (ℓ, κ, l) from both A and B, with $2^l > \ell$ and l dividing κ. If $l < \kappa$, the values will be secret shared, interpolated and checked in κ/l blocks.

Construct

1. A constructs a random polynomial in $\mathbb{F}_{2^l}$ of degree $\ell/2$ by choosing $\ell/2+1$ random values in $\{0,1\}^l$ which will be the coefficients. Call this polynomial P.
2. Using P compute and return the following ℓ pairs $(i, P(i))_{i \in [\ell]}$.

Reconstruct$\left(\psi, (i, P(i))_{i \in \psi}\right)$

1. If ψ is not a subset of $[\ell]$ of size at least $\ell/2+1$ then return `abort`. Otherwise take the $\ell/2+1$ first pairs of $(i, P(i))_{i \in \psi}$ (call this set ψ') and do interpolation on these points. Call the polynomial that results from this interpolation P'.
2. Return $(i, P'(i))_{i \in [\ell]}$

Verify$\left(\psi, (i, P(i))_{i \in \psi}\right)$

1. If ψ is not a subset of $[\ell]$ of at least $\ell/2+1$ entries then output `abort`. Otherwise take the $\ell/2+1$ first pairs of $(i, P(i))_{i \in \psi}$ (call this set ψ') and do interpolation on these points. Call the polynomial that results from this interpolation P'.
2. For the values in ψ not used in the interpolation use the interpolated polynomial P to verify the values lie on this polynomial. That is, for all $i \in \psi \backslash \psi'$ check if $P(i) =^? P'(i)$. If this is not true for some i return `reject` otherwise return `accept`.

Figure 4.4: The methods for polynomial manipulation.

4.3 The Full Protocol

We here give the full protocol for our second and most efficient scheme only. We have chosen to only give the details of this as it builds on top of our first protocol. We denote the protocol by Π_{SFE}. Afterwards we give a UC proof of security in Section 4.4.

We describe $\Pi_{\mathrm{SFE}}^{f,s}$ in the $\mathcal{F}_{\mathrm{COM}}$-, $\mathcal{F}_{\mathrm{BOTCC}}^{\bar{n_\mathsf{B}},\ell,\kappa}$-, $\mathcal{F}_{\mathrm{CT}}$-hybrid model, assuming that (G,F) is a secure link scheme in accordance with Definition 12 and that $\mathcal{G}$ is a binary key size preserving, structure free free-XOR gate garbling scheme with the prv.ind and ver properties and leakage function $\mathbf{\Phi}_{\mathsf{xor}}$ in accordance with the definitions from Section 2.4.

Setup

1. We denote A's input by x_A and B's input by x_B. Furthermore, if a party ever sends something inconsistent with what he/she is supposed to, then the receiving party outputs `abort` and halts.
2. Let $\bar{n_\mathsf{B}} = \max(4 \cdot n_\mathsf{B}, 8 \cdot s)$ and let ℓ be the smallest even integer satisfying $\ell - \frac{1}{2}\log(\ell) + \log\left(\frac{2\sqrt{2\pi}}{e^2}\right) \geq s$. Furthermore, let $p = 6 \cdot s + 7$.
3. B chooses a random matrix $M^{\mathsf{Sec}} \in \{0,1\}^{n_\mathsf{B} \times \bar{n_\mathsf{B}}}$ and a random bit vector $\bar{x_\mathsf{B}} \in \{0,1\}^{\bar{n_\mathsf{B}}}$ under the constraint that $M^{\mathsf{Sec}} \cdot \bar{x_\mathsf{B}} = x_\mathsf{B}$ and sends M^{Sec} to A.
4. Based on M^{Sec} the parties agree[3] on the function $\bar{f}(\bar{x_\mathsf{A}} \| \bar{x_\mathsf{B}})$ whose output is equal to $f\left(x_\mathsf{A} \| M^{\mathsf{Sec}} \cdot \bar{x_\mathsf{B}}\right) = y$ where A gives $\bar{x_\mathsf{A}} \in \{0,1\}^{n_\mathsf{A}+s}$ as input and B gives $\bar{x_\mathsf{B}} \in \{0,1\}^{\bar{n_\mathsf{B}}}$ as input and B receives $y_\mathsf{B} = y \in \{0,1\}^m$ as output.

Polynomial Setup

1. A and B initialize Fig. 4.4 with input (ℓ, κ, κ).
2. A executes **Construct** in Fig. 4.4 p times to construct polynomials of degree at most $\ell/2$ from the finite field $\mathbb{F}_{2^\kappa}$. We call these polynomials $P_1, P_2, \ldots, P_p$. For $i \in [p]$ and $j \in [\ell]$ she then computes the points $P_i(j)$. For each $i \in [p]$ the ℓ points $(P_i(j))_{j\in[\ell]}$ thus uniquely define the polynomial P_i of degree at most $\ell/2$.
3. Using $\mathcal{F}_{\mathrm{COM}}$, A verifiably commits to $P_i(j)$ for all $i \in [p]$ and $j \in [\ell]$. Formally, she calls $\mathcal{F}_{\mathrm{COM}}\left(\mathsf{vc}, \mathsf{A}, \mathsf{sid}, \mathsf{ID}_{i,j}^{\mathsf{poly}}, P_i(j)\right)$ and B receives $\left(\mathsf{vc}, \mathsf{A}, \mathsf{sid}, \mathsf{ID}_{i,j}^{\mathsf{poly}}\right)$ from $\mathcal{F}_{\mathrm{COM}}$ for all $i \in [p]$ and $j \in [\ell]$.
4. B chooses a challenge ψ, that is, a random set of distinct elements from $[p]$ such that $|\psi| = \lfloor 1.18s + 2.18 \rfloor$, and sends ψ to A.
5. A opens the commitments to all the points corresponding to the received challenge. That is, for each $i \in \psi$ and $j \in [\ell]$ she calls $\mathcal{F}_{\mathrm{COM}}\left(\mathsf{open}, \mathsf{A}, \mathsf{sid}, \mathsf{ID}_{i,j}^{\mathsf{poly}}\right)$ and in turn B learns $P_i(j)$.
6. B now verifies that for each $i \in \psi$ the points $(P_i(j))_{j\in[\ell]}$ define a polynomial of degree at most $\ell/2$ by computing $\mathbf{Verify}(\psi, (i, P_i)_{i\in\psi})$. If any of these checks fail B outputs `abort` and halts. Otherwise define the set $\chi = [p]\backslash\psi$, which is then the indices of the remaining polynomials.

Oblivious Transfer

1. A garbles ℓ copies of $\bar{f}$ using $\mathcal{G}$. Denote the garbling as

$$\{(F_j, e_j, d_j)\}_{j\in[\ell]} \leftarrow \mathsf{Gb}\left(1^\kappa, \bar{f}; r_j\right) .$$

[3] This is easy to do based on a deterministic algorithm.

Furthermore, let

$$\left(X^0_{1,j}, X^1_{1,j}, \ldots, X^0_{\bar{n},j}, X^1_{\bar{n},j}\right) \leftarrow e_j \text{ and } (Y^0_{1,j}, Y^1_{1,j}, \ldots, Y^0_{m,j}, Y^1_{m,j}) \leftarrow d_j \ ,$$

for $j \in [\ell]$.

2. A then calls $\mathcal{F}^{\bar{n_\mathsf{B}},\ell,\kappa}_{\text{BOTCC}}$ with

$$\left(\mathsf{sender}, \mathsf{sid}, \left(\left(X^0_{\bar{n_\mathsf{A}}+i,j}, X^1_{\bar{n_\mathsf{A}}+i,j}\right)_{i\in[\bar{n_\mathsf{B}}]}\right)_{j\in[\ell]}\right) \ ,$$

and B inputs $\left(\mathsf{receiver}, \mathsf{sid}, (\bar{x_\mathsf{B}}[i])_{i\in[\bar{n_\mathsf{B}}]}\right)$. Thus B learns the output

$$\left(\mathtt{transfer}, \mathsf{sid}\left(\left(X^{\bar{x_\mathsf{B}}[i]}_{\bar{n_\mathsf{A}}+i,j}\right)_{i\in[\bar{n_\mathsf{A}}]}\right)_{j\in[\ell]}\right) \ .$$

Commitment

1. A sends the garbled circuits $(F_j)_{j\in[\ell]}$ and decoding information $(d_j)_{j\in[\ell]}$ to B.
2. A then uses $\mathcal{F}_{\text{COM}}$ to make verifiable commitments to the concatenation of the 0- and 1-key on each of her augmented input wires for each garbled circuit. She does so by calling $\left(\mathsf{vc}, \mathsf{A}, \mathsf{sid}, \mathsf{ID}^{\mathsf{A-order}}_{i,j}, X^0_{i,j} \| X^1_{i,j}\right)$ on $\mathcal{F}_{\text{COM}}$ for all $i \in [\bar{n_\mathsf{A}}]$ and $j \in [\ell]$.
3. She also commits to her augmented input by calling $\left(\mathtt{commit}, \mathsf{A}, \mathsf{sid}, \mathsf{ID}^{\mathsf{A-in}}_{i,j}, X^{\bar{x_\mathsf{A}}[i]}_{i,j}\right)$ on $\mathcal{F}_{\text{COM}}$ for all $i \in [\bar{n_\mathsf{A}}]$ and $j \in [\ell]$.

Augmentation

1. Now B samples three uniformly random bit vectors $\beta \in_R \{0,1\}^{n_\mathsf{A}+s-1}$, $\beta_a \in_R \{0,1\}^{\lceil m+4.82s+3.82\rceil}$ and $\beta_b \in_R \{0,1\}^{\lceil 4.82s+4.82\rceil}$. The first two vectors define the matrices $M^{\mathsf{In}} \in \{0,1\}^{s\times n_\mathsf{A}}$ and $M^{\mathsf{Out}} \in \{0,1\}^{\lceil 4.82s+4.82\rceil\times m}$ by having $M^{\mathsf{In}}[i,j] = \beta[i+j-1]$, respectively $M^{\mathsf{Out}}[i,j] = \beta_a[i+j-1]$.
2. B sends the three vectors β, β_a and β_b to A who then defines the matrices M^{In} and M^{Out} similarly.
3. A augments the garbled circuits to compute $\left(M^{\mathsf{In}} \cdot x_\mathsf{A}\right) \oplus \alpha$ where x_A and α are in column form. That is, she computes the keys $\bar{Y}^0_{i,j} = \left(\bigoplus_{l=1}^{n_\mathsf{A}} M^{\mathsf{In}}[i,l] \cdot X^0_{l,j}\right) \oplus X^0_{n_\mathsf{A}+i,j}$, for all $i \in [s]$ and all circuits $j \in [\ell]$.
4. She then constructs verifiable commitments to the output keys of M^{In}. That is, she calls $\mathcal{F}_{\text{COM}}\left(\mathsf{vc}, \mathsf{A}, \mathsf{sid}, \mathsf{ID}^{\mathsf{aug0}}_{i,j}, \bar{Y}^0_{i,j}\right)$ and $\mathcal{F}_{\text{COM}}\left(\mathsf{vc}, \mathsf{A}, \mathsf{sid}, \mathsf{ID}^{\mathsf{aug1}}_{i,j}, \bar{Y}^1_{i,j}\right)$ for all $i \in [s]$ and $j \in [\ell]$.
5. Next, A augments the garbled circuits further such that they compute $(M^{\mathsf{Out}} \cdot y) \oplus \beta_b$ where y is the binary output in column form. That is, letting $\Delta_j = Y^0_{1,j} \oplus Y^1_{1,j}$ she computes the keys $\tilde{Y}^0_{i,j} = \left(\bigoplus_{l=1}^{m} M^{\mathsf{Out}}[i,l] \cdot Y^0_{l,j}\right) \oplus (\beta_b[i] \cdot \Delta_j)$, for all $i \in [\lceil 4.82s + 4.82\rceil]$ and all circuits $j \in [\ell]$.
6. A then computes the links $L_{i,j} = \mathsf{G}\left(P_{\chi[i]}(j), \tilde{Y}^0_{i,j}\right)$ for $i \in [\lceil 4.82s + 4.82\rceil]$ and $j \in [\ell]$. A then sends these links to B.

Cut-and-choose

1. A and B invoke $\mathcal{F}_{\text{CT}}$ by calling $(\texttt{flip}, \mathsf{sid}, \kappa)$ and B inputs $\texttt{continue}$ so they both get random coin tosses. They then use these as input to a deterministic algorithm to randomly select $\ell/2$ elements from $[\ell]$. Call these $\ell/2$ elements π and define $\phi = [\ell]\backslash\pi$. We say that the set π is the *check set* and that ϕ is the *evaluation* set.
2. For the check circuits A sends to B the encoding information and the randomness used to construct the check circuits. Thus she sends $(r_j, e_j)_{j\in\pi}$.
3. B uses this to verify the commitments to the concatenation of the 0- and 1-key on A's input wires. That is, he parses $e_j \to \left(\bar{X}^0_{1,j}, \bar{X}^1_{1,j}, \dots, \bar{X}^0_{\bar{n_\mathsf{A}},j}, \bar{X}^1_{\bar{n_\mathsf{A}},j}\right)$ for $j \in \pi$. He then calls $\mathcal{F}_{\text{COM}}\left(\mathsf{check}, \mathsf{A}, \mathsf{sid}, \mathsf{ID}^{\mathsf{A-order}}_{i,j}, \bar{X}^0_{i,j}\|\bar{X}^1_{i,j}\right)$ for each $i \in [\bar{n_\mathsf{A}}]$, $j \in \pi$. If the functionality ever return $\perp$ then he outputs $\texttt{abort}$ and halts. He also checks that the encoding information is consistent with the keys he learned through the OTs, that is, that $X^{\bar{x_\mathsf{B}}[i]}_{\bar{x_\mathsf{A}}+i,j} = \bar{X}^{\bar{x_\mathsf{B}}[i]}_{\bar{x_\mathsf{A}}+i,j}$ for $i \in [\bar{n_\mathsf{B}}]$ and $j \in \pi$. If not then he outputs $\texttt{abort}$ and halts.
4. B then verifies the garbled circuits by calling $\mathsf{Ve}\left(F_j, \bar{f}, e_j, d_j, r_j\right)$ for $j \in \pi$. If $\perp$ is ever given as output then he outputs $\texttt{abort}$ and halts.
5. B also checks that the augmentation computing a digest on A's input has been constructed correctly by checking the verifiable commitments on the output keys of the check circuits. That is, he computes $\Delta_j = \bar{X}^0_{1,j} \oplus \bar{X}^1_{1,j}$ for all $j \in \pi$. He then computes $\bar{Y}^0_{i,j} = \left(\bigoplus_{l=1}^{n_\mathsf{A}} M^{\mathsf{In}}[i,l]\cdot \bar{X}^0_{l,j}\right) \oplus \bar{X}^0_{n_\mathsf{A}+i,j}$, $\bar{Y}^1_{i,j} = \bar{Y}^0_{i,j} \oplus \Delta_j$ and then calls $\mathcal{F}_{\text{COM}}\left(\mathsf{check}, \mathsf{A}, \mathsf{sid}, \mathsf{ID}^{\mathsf{aug0}}_{i,j}, \bar{Y}^0_{i,j}\right)$ and $\mathcal{F}_{\text{COM}}\left(\mathsf{check}, \mathsf{A}, \mathsf{sid}, \mathsf{ID}^{\mathsf{aug1}}_{i,j}, \bar{Y}^1_{i,j}\right)$ for all $i \in [s]$ and $j \in \pi$. If $\mathcal{F}_{\text{COM}}$ ever returns $\perp$ he outputs $\texttt{abort}$ and halts.
6. In the same manner B then uses all the output keys (which are computed through the method Ve) from the garbled circuits to compute the augmented output. That is, he computes $\tilde{Y}^0_{i,j} = \left(\bigoplus_{l=1}^{m} M^{\mathsf{Out}}[i,l]\cdot Y^0_{l,j}\right) \oplus (\beta_b[i]\cdot\Delta_j)$, $\tilde{Y}^1_{i,j} = \tilde{Y}^0_{i,j} \oplus \Delta_j$ and uses these values to compute $P_{\chi[i]}(j)$ via the link $L_{i,j}$. That is he computes $\mathsf{F}\Big(\texttt{backwards}, L_{i,j}, \tilde{Y}^0_{i,j}\Big) = P_{\chi[i]}(j)$. He then verifies these values by calling $\mathcal{F}_{\text{COM}}\left(\mathsf{check}, \mathsf{ID}^{\mathsf{poly}}_{\chi[i],j}, P_{\chi[i]}(j)\right)$ for all $i \in [\lceil 4.82s + 4.82\rceil]$ and $j \in \pi$.
7. If any of the above checks fail or if $\mathcal{F}_{\text{COM}}$ returns $\perp$ on any of the calls, B outputs $\texttt{abort}$ and halts.

Evaluation

1. A now opens her input keys by calling $\mathcal{F}_{\text{COM}}\left(\mathsf{open}, \mathsf{A}, \mathsf{sid}, \mathsf{ID}^{\mathsf{A-in}}_{i,j}\right)$ for all $i \in [\bar{n_\mathsf{A}}]$ and $j \in \phi$ from which B in turn learns $X^{\bar{x_\mathsf{A}}[i]}_{i,j}$.
2. Using A's input keys B evaluates the augmentation for consistency checking of A's input for all $j \in \phi$. Thus he computes $\bar{Y}_{i,j} = \left(\bigoplus_{l=1}^{n_\mathsf{A}} M^{\mathsf{In}}_{i,l}\cdot X^{\bar{x_\mathsf{A}}[l]}_{l,j}\right) \oplus X^{\bar{x_\mathsf{A}}[n_\mathsf{A}+i]}_{n_\mathsf{A}+i,j}$ for all $i \in [s]$ and $j \in \phi$. He then uses the verifiable commitments to find the bit each key $\bar{Y}_{i,j}$ represents. Thus he calls $\mathcal{F}_{\text{COM}}\left(\mathsf{check}, \mathsf{A}, \mathsf{sid}, \mathsf{ID}^{\mathsf{aug0}}_{i,j}, \bar{Y}_{i,j}\right)$ and $\mathcal{F}_{\text{COM}}\Big(\mathsf{check}, \mathsf{A}, \mathsf{sid}, \mathsf{ID}^{\mathsf{aug1}}_{i,j}, \bar{Y}_{i,j}\Big)$ and sets the bit $\bar{y}_{i,j} := 0$ if the first query returns $\top$, and $\bar{y}_{i,j} := 1$ if instead the second query returns $\top$. If both queries return $\top$ or both return $\perp$ then he outputs $\texttt{abort}$ and halts. He then verifies that $\bar{y}_{i,j} = \bar{y}_{i,j'}$ for all $i \in [s]$ and $j, j' \in \phi$ and if any check fails he outputs $\texttt{abort}$ and halts.

3. B now uses A's input keys along with his own input keys, which he learned in the *Oblivious Transfer* phase, to evaluate all garbled circuits in the set ϕ. First, he lets

$$X_j \leftarrow \left(X_{1,j}^{x_{\mathsf{A}}[1]}, \ldots, X_{\bar{n_{\mathsf{A}}},j}^{x_{\mathsf{A}}[\bar{n_{\mathsf{A}}}]}, X_{\bar{n_{\mathsf{A}}}+1,j}^{\bar{x_{\mathsf{B}}}[1]}, X_{n_{\mathsf{A}}+\bar{n_{\mathsf{B}}},j}^{x_{\mathsf{B}}[\bar{n_{\mathsf{B}}}]}\right) ,$$

for each $j \in \phi$. (The first $\bar{n_{\mathsf{A}}}$ values he learned from A's opening to $\mathsf{ID}^{\mathsf{A-in}}$, the last $\bar{n_{\mathsf{B}}}$ he learned from the OTs.)

4. Next he computes $Y_j \leftarrow \mathsf{Ev}(F_j, X_j)$ for each $j \in \phi$.

5. For each $j \in \phi$ he lets $y_j \leftarrow \mathsf{De}(d_j, Y_j)$. If $y_j = \perp$ he discards F_j.

6. If $y_{\phi[1]}[i] = y_j[i]$ for all $j \in \phi \backslash \phi[1]$ then B accepts and returns this output. If not then B proceeds with the *Reconstruction* phase.[4]

Reconstruction

1. B computes $\tilde{y}_j = \left(M^{\mathsf{Out}} \cdot y_j\right) \oplus \beta_b$ and lets $\tilde{y}_{i,j} = \tilde{y}_j[i]$ for all $i \in [[\lceil 4.82s + 4.82 \rceil]]$ and $j \in \phi$.

2. Similarly he computes the augmented output keys as $\tilde{Y}_{i,j}^{\tilde{y}_{i,j}} = \left(\bigoplus_{l=1}^{m} M^{\mathsf{Out}}[i,l] \cdot Y_{l,j}^{y_j[l]}\right)$ for all $i \in [[\lceil 4.82s + 4.82 \rceil]]$ and $j \in \phi$.

3. Now for each augmented output wire, evaluating to a semantic 0, B computes the values $P_{\chi[i]}(j)$ for all $i \in [[\lceil 4.82s + 4.82 \rceil]]$ and $j \in \phi$. That is, he computes $P_{\chi[i]}(j) = \mathsf{F}\left(\texttt{backwards}, L_{i,j}, \tilde{Y}_{i,j}^{\tilde{y}_{i,j}}\right)$ for all $i \in [[\lceil 4.82s + 4.82 \rceil]]$ and $j \in \phi$ where $\tilde{y}_{i,j} = 0$. For each of these values, he calls $\mathcal{F}_{\text{COM}}\left(\mathsf{check}, \mathsf{A}, \mathsf{sid}, \mathsf{ID}_{\chi[i],j}^{\mathsf{poly}}, P_{\chi[i]}(j)\right)$. If $\mathcal{F}_{\text{COM}}$ returns $\perp$ on any of these instances he discards the associated circuit j.

4. B now takes the remaining garbled circuits that evaluate to different values, and looks at all the augmented output wires that evaluate differently between at least two of these garbled circuits, to find a new polynomial point. That is, define $U \subseteq \phi$ to be the set of circuits which has not been discarded and define $W \subseteq [[\lceil 4.82s + 4.82 \rceil]]$ such that for each $i \in W$ there exists a pair of indices $j, j' \in U$ where $\tilde{y}_{i,j} \neq \tilde{y}_{i,j'}$. Now for each $i \in W$ do the following:[5]

   a) Find the first element $j \in U$ for which it is true that $\tilde{y}_{i,j} = 0$. Now using link $L_{i,j}$ he calls $\mathcal{F}_{\text{COM}}\left(\mathsf{check}, \mathsf{A}, \mathsf{sid}, \mathsf{ID}_{i,j}^{\mathsf{poly}}, \mathsf{F}\left(\texttt{backwards}, L_{i,j}, \tilde{Y}_{i,j}\right)\right)$. If $\mathcal{F}_{\text{COM}}$ returns $\perp$ then take the next element (if it exist) $j \in U$ where $\tilde{y}_{i,j} = 0$ and call $\mathcal{F}_{\text{COM}}\left(\mathsf{check}, \mathsf{A}, \mathsf{sid}, \mathsf{ID}_{i,j}^{\mathsf{poly}}, \mathsf{F}\left(\texttt{backwards}, L_{i,j}, \tilde{Y}_{i,j}\right)\right)$ and continue taking elements (if they exist) until $\mathcal{F}_{\text{COM}}$ returns $\top$.

   b) When $\mathcal{F}_{\text{COM}}$ returns $\top$ do polynomial interpolation using the list of pairs $S = \left((l, P_i(l))_{l \in \pi}, \left(j, \mathsf{F}\left(\texttt{backwards}, L_{i,j}, \tilde{Y}_{i,j}\right)\right)\right)$ by calling **Reconstruct**$(\pi \cup \{j\}, S)$. Then the result should be $(l, P_i(l))_{l \in [\ell]}$. For each $l \in [\ell] \backslash \{\pi \cup \{j\}\}$ call $\mathcal{F}_{\text{COM}}\left(\mathsf{check}, \mathsf{A}, \mathsf{sid}, \mathsf{ID}_{i,l}^{\mathsf{poly}}, P_i(l)\right)$.

   c) Next, for each element $j' \in U$ for which it is true that $\tilde{y}_{i,j} = 1$, compute the values $\Delta_{j'} = \mathsf{F}(\texttt{forwards}, L_{i,j}, P_i(j')) \oplus \tilde{Y}_{i,j'}$.

[4]In order to avoid timing attacks B should always do the *Reconstruction*. If no issues have arisen then he should do it for simulated values.

[5]The following should be done for each j being an index of an evaluation circuit in order to avoid timing attacks.

5. Now B should know Δ_j for all $j \in \phi$ where the garbled circuit has not been discarded. If he does not, then he outputs `abort` and halts.
6. Using Δ_j and A's input keys B can find A's input for all circuits calling $\mathcal{F}_{\text{COM}}\Big(\mathsf{check}, \mathsf{A}, \mathsf{sid}, \mathsf{ID}_{i,j}^{\mathsf{A-order}}, X_{i,j}^{x_{\mathsf{A}}[i]} \| X_{i,j}^{x_{\mathsf{A}}[i]} \oplus \Delta_j\Big)$ and $\mathcal{F}_{\text{COM}}\left(\mathsf{check}, \mathsf{ID}_{i,j}^{\mathsf{A-order}}, X_{i,j}^{x_{\mathsf{A}}[i]} \oplus \Delta_j \| X_{i,j}^{x_{\mathsf{A}}[i]}\right)$, if the first call returns $\top$, A's bit $x_{\mathsf{A}}[i] = 0$, if the second call returns $\top$ then $x_{\mathsf{A}}[i] = 1$, otherwise he outputs `abort` and halts.
7. If A's input bits are the same in all the remaining garbled circuits B computes $\mathsf{ev}(f, x_{\mathsf{A}} \| x_{\mathsf{B}}) \rightarrow y$ and finally outputs y. However, if there is a discrepancy he evaluates $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ in plain and compares the output with the semantic meaning of the output of $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ he found in *Evaluation*. He discards any circuit where the plain and the semantic output are discrepant. Finally, he uses A's input of any remaining circuit to evaluate the function f in plain and outputs the result.

4.4 Proof of Security

Before continuing with the actual proof we need the following preliminaries about universal hash functions:

Universal Hash Functions

Definition 13 ((Binary) universal hash function)**.** A family of functions $\mathcal{H} = \{h : \{0,1\}^q \to \{0,1\}^w\}$ is a family of universal hash functions if

$$\forall x, y \in \{0,1\}^q, x \neq y \ : \ \mathsf{Pr}_{h \in_R \mathcal{H}}[h(x) = h(y)] \leq 2^{-w} \ .$$

Furthermore, we say that a family of universal hash functions have the *uniform difference property* if

$$\forall x, y \in \{0,1\}^q, x \neq y \ : \ \forall z \in \{0,1\}^w \ : \ \mathsf{Pr}_{h \in_R \mathcal{H}}[h(x) \oplus h(y) = z] = 2^{-w} \ .$$

We say that the family is *pairwise independent* if

$$\forall x, y \in \{0,1\}^q, x \neq y \ : \ \forall z_1, z_2 \in \{0,1\}^w \ : \ \mathsf{Pr}_{h \in_R \mathcal{H}}[h(x) = z_1 \wedge h(y) = z_2] = 2^{-2w} \ .$$

Finally, we say that a family is *uniform* if

$$\forall x \in \{0,1\}^q, \forall z \in \{0,1\}^w \ : \ \mathsf{Pr}_{h \in_R \mathcal{H}}[h(x) = z] = 2^{-w} \ .$$

From the definition notice that pairwise independence is the strongest type of family as it implies both uniformity and the uniform difference property. Furthermore, notice that if the family is uniform then it also satisfies the uniform difference property, but the converse is not necessarily true.

Definition 14 ($\mathbb{H}^{\mathsf{In}}$ (formal))**.** Let $\beta \in \{0,1\}^{n_{\mathsf{A}}+s-1}$ and $M^{\mathsf{In}} \in \{0,1\}^{s \times n_{\mathsf{A}}}$ be a matrix with each entry defined as $M^{\mathsf{In}}[i,j] = \beta[i+j-1]$ for $i \in [s]$, $j \in [n_{\mathsf{A}}]$. Now define the function $h_\beta : \{0,1\}^{n_{\mathsf{A}}} \to \{0,1\}^s$ as $h_\beta(x) = M^{\mathsf{In}} \cdot x = z$ when queried on β. Finally define a family of functions $\mathbb{H}^{\mathsf{In}} = \{h_\beta(\cdot)\}_{\beta \in \{0,1\}^{n_{\mathsf{A}}+s-1}}$.

Lemma 7. $\mathbb{H}^{\mathsf{In}}$ *is a universal hash function with the uniform difference property.*

Proof. First notice that it is enough to show the uniform difference property by the following:

$$\forall x, y \in \{0,1\}^{n_{\mathsf{A}}}, \ x \neq y \ : \ \forall z \in \{0,1\}^s \ : \ \mathsf{Pr}_{h \in_R \mathcal{H}}[h(x) \oplus h(y) = z] = 2^{-s} \Rightarrow$$
$$\forall x, y \in \{0,1\}^{n_{\mathsf{A}}}, \ x \neq y \ : \ \forall z \in \{0,1\}^s \ : \ \mathsf{Pr}_{h \in_R \mathcal{H}}[h(x) = z \oplus h(y)] = 2^{-s} \ .$$

This must in particular be true when $z = 0^s$ and so $\forall x, y \in \{0,1\}^{n_{\mathsf{A}}}, \ x \neq y \ : \ \mathsf{Pr}_{h \in_R \mathcal{H}}[h(x) = h(y)] = 2^{-s} \leq 2^{-s}$, which meets the definition of a universal hash function.

Now for the uniform difference property we need to show that

$$\forall z \in \{0,1\}^s : \mathsf{Pr}\left[\left(M^{\mathsf{In}} \cdot x\right) \oplus \left(M^{\mathsf{In}} \cdot x'\right) = z\right] \ ,$$

when each entry of M^{In} is defined as $M^{\mathsf{In}}[i,j] = \beta[i+j-1]$, $\beta \in_R \{0,1\}^{n_{\mathsf{A}}+s-1}$ and $x, x' \in \{0,1\}^{n_{\mathsf{A}}}$ where $x \neq x'$. First notice that matrix multiplication is linear, so these are equivalent:

$$\left(M^{\mathsf{In}} \cdot x\right) \oplus \left(M^{\mathsf{In}} \cdot x'\right) = z$$
$$M^{\mathsf{In}} \cdot (x \oplus x') = z \ .$$

Thus it is enough to show that $\mathsf{Pr}[M^{\mathsf{In}} \cdot (x \oplus x') = z] = 2^{-s}$ for $x \neq x'$ and any z. We do this by showing that for any choice of $(x \oplus x') \neq 0$ there exist $2^{n_{\mathsf{A}}-1}$ choices of β for each of the 2^s possible elements in the range of the function. Thus when β is chosen at random (from $\{0,1\}^{n_{\mathsf{A}}+s-1}$) there is uniform probability of hitting each of the elements in the function's range, in particular of hitting z.

Now set $\tilde{x} = x \oplus x'$ and see that given β, $M^{\mathsf{In}} \cdot \tilde{x}$ can be computed as follows:

$$M^{\mathsf{In}} \cdot \tilde{x} = \begin{pmatrix} \beta[1] & \beta[2] & \dots & \beta[n_{\mathsf{A}}] \\ \beta[2] & \beta[3] & \dots & \beta[n_{\mathsf{A}}+1] \\ \vdots & \vdots & \ddots & \vdots \\ \beta[s] & \beta[s+1] & \dots & \beta[n_{\mathsf{A}}+s-1] \end{pmatrix} \cdot \begin{pmatrix} \tilde{x}[1] \\ \tilde{x}[2] \\ \vdots \\ \tilde{x}[n_{\mathsf{A}}] \end{pmatrix} .$$

That is, as an $s \times n_{\mathsf{A}}$ binary matrix times a column vector of n_{A} bits. Notice that we can rewrite this computation as follows:

$$\begin{pmatrix} \beta[1] & \beta[2] & \dots & \beta[n_{\mathsf{A}}] \\ \beta[2] & \beta[3] & \dots & \beta[n_{\mathsf{A}}+1] \\ \vdots & \vdots & \ddots & \vdots \\ \beta[s] & \beta[s+1] & \dots & \beta[n_{\mathsf{A}}+s-1] \end{pmatrix} \cdot \begin{pmatrix} \tilde{x}[1] \\ \tilde{x}[2] \\ \vdots \\ \tilde{x}[n_{\mathsf{A}}] \end{pmatrix}$$

$$= \begin{pmatrix} \tilde{x}[1] & \tilde{x}[2] & \tilde{x}[3] & \cdots & \tilde{x}[n_{\mathsf{A}}] & 0 & 0 & 0 & \cdots & 0 \\ 0 & \tilde{x}[1] & \tilde{x}[2] & \cdots & \tilde{x}[n_{\mathsf{A}}-1] & \tilde{x}[n_{\mathsf{A}}] & 0 & 0 & \cdots & 0 \\ 0 & 0 & \tilde{x}[1] & \cdots & \tilde{x}[n_{\mathsf{A}}-2] & \tilde{x}[n_{\mathsf{A}}-1] & \tilde{x}[n_{\mathsf{A}}] & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & 0 & 0 & \tilde{x}[1] & \cdots & \tilde{x}[n_{\mathsf{A}}] \end{pmatrix}$$

$$\cdot \begin{pmatrix} \beta[1] \\ \beta[2] \\ \beta[3] \\ \vdots \\ \beta[n_{\mathsf{A}}+s-1] \end{pmatrix} .$$

A bit more specifically we view β as a column vector of $n_{\mathsf{A}}+s-1$ bits and define a binary matrix $X \in \{0,1\}^{s \times (n_{\mathsf{A}}+s-1)}$ where

$$X[i,j] = \begin{cases} \tilde{x}[j-i+1] & \text{if } 0 < j-i+1 \leq n_{\mathsf{A}} \\ 0 & \text{otherwise .} \end{cases}$$

Now see that since $\tilde{x} \neq 0^{n_{\mathsf{A}}+s-1}$ there must be at least one bit, say at position k, that is 1. Thus, we see that matrix X will be of the following form:

$$\begin{pmatrix} \tilde{x}[1] & \cdots & 1 & \cdots & \tilde{x}[n_{\mathsf{A}}] & 0 & 0 & \cdots & 0 \\ 0 & \tilde{x}[1] & \cdots & 1 & \cdots & \tilde{x}[n_{\mathsf{A}}] & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & \tilde{x}[1] & \cdots & 1 & \cdots & \tilde{x}[n_{\mathsf{A}}] \end{pmatrix} .$$

That is, the matrix is in row echelon form and does not have any all-0 rows. Thus the matrix has full rank (s) and therefore is surjective and so we can hit each of the 2^s possible s-bit outputs.

Next consider row i and column $i+k-1$. This entry will have value 1. No matter what the value is of all the other $n_{\mathsf{A}}+s-2$ entries in row i are set to, the i'th bit in the output vector, that is $(M^{\mathsf{In}} \cdot \tilde{x})[i]$ can be uniquely decided by setting $\beta[i+k-1]$ to either 0 or 1. That is, if $\beta[i+k-1]=0$ the $n_{\mathsf{A}}+s-2$ other entries in row i and the vector β computes the i'th output bit, that is $(M^{\mathsf{In}} \cdot \tilde{x})[i]$. If instead $\beta[i]=1$ the $n_{\mathsf{A}}+s-2$ other entries in row i and the vector β computes the negation of the i'th output bit, that is $1-((M^{\mathsf{In}} \cdot \tilde{x})[i])$. The point is that $\beta[i]$ can be used to hit either 0 or 1 no matter what the other bits in the matrix and the vector β are set to. This is clearly true individually for all the rows. However, as the output is a vector and we use one bit of β for each row to determine each output bit we see that for a given output vector (of s bits) we need s bits of β to uniquely decide its value, whereas the rest of the $n_{\mathsf{A}}+s-1-s=n_{\mathsf{A}}-1$ bits of β remain free no matter what output we desire and what the input is. Thus there will exist at least $2^{n_{\mathsf{A}}-1}$ choices of β to give a specific output, in particular z on any non-zero choice of $\tilde{x}$. But since we have already established that each possible output must occur once, by the fact that our new matrix description of the computation has full rank, it means that this is the case for all 2^s possible outputs. Thus there can only be exactly $2^{n_{\mathsf{A}}-1}$ choices of β for each output, and in particular z, as $2^s \cdot 2^{n_{\mathsf{A}}-1}=2^{n_{\mathsf{A}}+s-1}$ covers all the possible choices of β. ☐

Definition 15 ($\mathbb{H}^{\mathsf{Out}}$ (formal))**.** Let $\beta_a \in \{0,1\}^{\lceil m+4.82s+4.82\rceil-1}$, $\beta_b \in \{0,1\}^{\lceil 4.82s+4.82\rceil}$ and $M^{\mathsf{Out}} \in \{0,1\}^{\lceil 4.82s+4.82\rceil \times m}$ be a matrix with each entry defined as $M^{\mathsf{Out}}[i,j]=\beta_a[i+j-1]$. Now define the function

$$h_{\beta_a,\beta_b}:\{0,1\}^m \rightarrow \{0,1\}^{\lceil 4.82s+4.82\rceil} \text{ as } h_{\beta_a,\beta_b}(x)=(M^{\mathsf{Out}} \cdot x) \oplus \beta_b = z\ .$$

Finally, define a family of functions

$$\mathbb{H}^{\mathsf{Out}}=\{h_{\beta_a,\beta_b}(\cdot)\}_{\beta_a \in \{0,1\}^{\lceil m+4.82s+3.82\rceil},\beta_b \in \{0,1\}^{\lceil 4.82s+4.82\rceil}}\ .$$

Lemma 8. $\mathbb{H}^{\mathsf{Out}}$ *is a pairwise independent universal hash function.*

Proof. A proof can be found in [MNT93]. ☐

Proof of Protocol Security

We prove the protocol Π_{SFE} secure according to the standard real/ideal simulation paradigm with sequential composition [Gol04] when using a private, verifiable, binary key size preserving, structure free, free-XOR gate garbling scheme and one have a secure link scheme in accordance with Definition 12 and hybrid access to the $\mathcal{F}_{\text{BOTCC}}$, $\mathcal{F}_{\text{CT}}$, and $\mathcal{F}_{\text{COM}}$ functionalities. In the end we argue how the protocol can be proven secure in the UC model [Can01] and show how to efficiently realize it in the random oracle, $\mathcal{F}_{\text{OT}}$-hybrid model.

Finally, notice that we need that there is at least one non-XOR gate on the path from an output wire to some input wire. If that is not the case, we can simply add at most m dummy gates.

Theorem 6. *The protocol* Π_{COM} *from Section 4.3 UC-securely realizes* $\mathcal{F}_{\text{SFE}}$ *in Fig. 4.1 in the* ($\mathcal{F}_{\text{BOTCC}}$, $\mathcal{F}_{\text{CT}}$, $\mathcal{F}_{\text{COM}}$)*-hybrid model against any PPT static and malicious adversary assuming that* $\mathcal{G}$ *is a key size preserving, structure free, free-XOR gate garbling scheme with the* prv.ind

and ver *properties and leakage function* $\mathbf{\Phi}_{\mathsf{xor}}$ *in accordance with the definitions in Chapter 2 and the link scheme* (G, F) *used is secure according to Definition 12. Furthermore, assume that there is at least one non-XOR gate on the path from an output wire to some input wire.*

Proof. We split the proof up in two lemmas, one considering a corrupted A and one where B is corrupted. Theorem 6 follows directly from these two lemmas.

Corrupt A

Lemma 9. *The protocol* Π_{COM} *from Section 4.3 UC-securely realizes* $\mathcal{F}_{\text{SFE}}$ *in Fig. 4.1 in the* $(\mathcal{F}_{\text{BOTCC}}, \mathcal{F}_{\text{CT}}, \mathcal{F}_{\text{COM}})$*-hybrid model against any PPT static and malicious adversary corrupting* A *assuming that* $\mathcal{G}$ *is a key size preserving, structure free, free-XOR gate garbling scheme with the* prv.ind *and* ver *properties and leakage function* $\mathbf{\Phi}_{\mathsf{xor}}$ *in accordance with the definitions in Chapter 2 and the link scheme* (G, F) *used is secure according to Definition 12. Furthermore, assume that there is at least one non-XOR gate on the path from an output wire to some input wire.*

Proof. Before proceeding with the formal proof we give some high-level intuition. Notice first that if A tries to cheat she can only do so before the cut-and-choose phase as there is no interaction (except opening of ideal commitments) with her after this phase.

In our approach we make sure that two evaluation circuits evaluating to different results enable B to recover A's plain input for these circuits. Hence, intuitively, a cheating A must guess *exactly* the cut-and-choose challenge (in order to ensure that all check circuits are good, while all evaluation circuits are bad). In this case the optimal ratio between check and evaluation circuits is $1/2$. This is seen as a cheating A's success probability is $\binom{s}{c\cdot s}^{-1}$ when $c \leq 1$. Now, $\binom{s}{c\cdot s}^{-1}$ is clearly minimized when $c = 1/2$ since $\binom{s}{c\cdot s}$ for $0 < c \leq 1$ is maximized when $c = 1/2$.

The following lemma gives a lower bound on the replication factor ℓ required in our protocol (that is, how many circuits to garble) in order to ensure that A guesses exactly the cut-and-choose challenge with probability at most 2^{-s}.

Lemma 10. *The probability of guessing exactly the elements in a set of size* $\ell/2$ *sampled randomly from a set of* ℓ *elements is less than* 2^{-s} *when* $\log\left(\frac{2\sqrt{2\pi}}{e^2}\right) - \frac{1}{2}\log(\ell) + \ell \geq s$*.*

Proof. First notice that a cheating A can only succeed if she guesses exactly the set of check circuits in the cut-and-choose phase. For circuit replication factor ℓ there are exactly $\binom{\ell}{\ell/2}$ possible sets, of which, one is randomly selected based on the coin tossing.

$$\binom{\ell}{\frac{\ell}{2}} = \frac{\ell!}{\frac{\ell}{2}!\left(\ell - \frac{\ell}{2}\right)!} = \frac{\ell!}{\left(\frac{\ell}{2}!\right)^2}$$

We now bound this using the "inequality version" of Stirling's approximation:

$$\sqrt{2\pi\ell} \cdot \left(\frac{\ell}{e}\right)^{\ell} \cdot e^{\frac{1}{12\ell+1}} \leq \ell! \leq \sqrt{2\pi\ell} \cdot \left(\frac{\ell}{e}\right)^{\ell} \cdot e^{\frac{1}{12\ell}} \ .$$

Notice that the factors $e^{\frac{1}{12\ell+1}}$ and $e^{\frac{1}{12\ell}}$ converge towards 1 when $\ell \geq 0$. Furthermore, see that when $\ell \geq 2$ it holds that $\sqrt{2\pi} \cdot e^{\frac{1}{12\ell}} < e$. Thus we can simplify the inequality to the following simpler form, assuming wlog that $\ell > 2$:

$$\sqrt{2\pi\ell} \cdot \left(\frac{\ell}{e}\right)^{\ell} \leq \sqrt{2\pi\ell} \cdot \left(\frac{\ell}{e}\right)^{\ell} \cdot e^{\frac{1}{12\ell+1}} \leq \ell! \leq \sqrt{2\pi\ell} \cdot \left(\frac{\ell}{e}\right)^{\ell} \cdot e^{\frac{1}{12\ell}} \leq e\sqrt{\ell} \cdot \left(\frac{\ell}{e}\right)^{\ell} \ .$$

Now, replacing the numerator with the lower bound of Stirling's approximation and replacing the denominator with the upper bound of Stirling's approximation we get the following inequality:

$$\frac{\ell!}{\left(\frac{\ell}{2}!\right)^2} \geq \frac{\sqrt{2\pi\ell}\left(\frac{\ell}{e}\right)^\ell}{\left(e\sqrt{\frac{\ell}{2}}\left(\frac{\frac{\ell}{2}}{e}\right)^{\frac{\ell}{2}}\right)^2} = \frac{\sqrt{2\pi\ell}\left(\frac{\ell}{e}\right)^\ell}{e^2\frac{\ell}{2}\left(\frac{\ell}{2e}\right)^\ell} = \frac{\sqrt{2\pi\ell}\ell^\ell e^{-\ell}}{e^2 2^{-1}\ell\ell^\ell 2^{-\ell}e^{-\ell}}$$

$$= \frac{2\sqrt{2\pi\ell}2^\ell}{e^2\ell} = \frac{2\sqrt{2\pi}}{e^2}\cdot\frac{\sqrt{\ell}2^\ell}{\ell} = \frac{2\sqrt{2\pi}}{e^2}\cdot\ell^{-\frac{1}{2}}2^\ell$$

We now want to find ℓ such that $\frac{2\sqrt{2\pi}}{e^2}\cdot\ell^{-\frac{1}{2}}2^\ell \geq 2^s$. This will mean that there will be at least 2^s possible choices for the cut-and-choose phase and thus A's probability of guessing the choice will be at most 2^{-s}. Setting $c = \frac{2\sqrt{2\pi}}{e^2}$, taking the logarithm, and isolating s we get:

$$\log\left(c\ell^{-\frac{1}{2}}2^\ell\right) \geq \log\left(2^s\right)$$

$$\log(c) - \frac{1}{2}\log(\ell) + \ell \geq s$$

□

Approximating the constant we get:

$$\ell - \frac{1}{2}\log(\ell) - 0.5596 \geq s$$

When choosing the replication factor according to Lemma 10, the only problem is that B might not know which circuit is the correct one. Thus, if more than one circuit "seemingly" evaluates correctly B must be able to discover which of these evaluations are correct. He does that by restoring A's input as mentioned in the *Reconstruction* phase and then evaluating the function f in plain. We now proceed with the formal proof.

Let $\mathcal{A}$ be an adversary controlling A in the real world execution of the protocol. We construct a simulator S running in the ideal world with external access to $\mathcal{F}_{\mathrm{SFE}}$. Internally S runs $\mathcal{A}$, controlling A, along with a simulation of the interaction between $\mathcal{A}$, an honest B and the ideal functionalities ($\mathcal{F}_{\mathrm{BOTCC}}$, $\mathcal{F}_{\mathrm{CT}}$, $\mathcal{F}_{\mathrm{COM}}$). See Fig. 4.5 for an illustration of this. The simulator S works as follows:

- S starts simulating internally the real world execution of the protocol consisting of $\mathcal{A}$, B and the ideal functionalities $\mathcal{F}_{\mathrm{BOTCC}}$, $\mathcal{F}_{\mathrm{CT}}$ and $\mathcal{F}_{\mathrm{COM}}$. If an honest B would output `abort` at any time during the simulation, then S sends `abort` to $\mathcal{F}_{\mathrm{SFE}}$.
- The simulator then simulates the *Setup*, *Polynomial Setup*, and *Oblivious Transfer* phases, but simulating B's input as the all-0 string, i.e., it sets $x_{\mathsf{B}} = 0^{n_{\mathsf{B}}}$.
- When reaching the *Commitment* phase the simulator extracts the ℓ (possibly different) inputs of $\mathcal{A}$ to the ℓ garbled circuits it has constructed. It does so directly since $\mathcal{A}$ calls $\mathcal{F}_{\mathrm{COM}}$ with the keys of its input wires. We will later determine which of these inputs is the one $\mathcal{A}$ is actually committed to. In the rest of the proof we denote the i'th input bit in the j'th circuit, given by $\mathcal{A}$, by ${x'}_{\mathsf{A}i}^{j}$. When we consider the entire vector of input bits in the j'th circuit we remove the subscript, i.e., ${x'}_{\mathsf{A}}^{j}$, and when we consider the input to the ideal functionality we simply call it x'_{A}. In either case if we consider the augmented input we add a bar, i.e., $\bar{x'}_{\mathsf{A}i}^{j}$, $\bar{x'}_{\mathsf{A}}^{j}$ and $\bar{x_{\mathsf{A}}}'$.

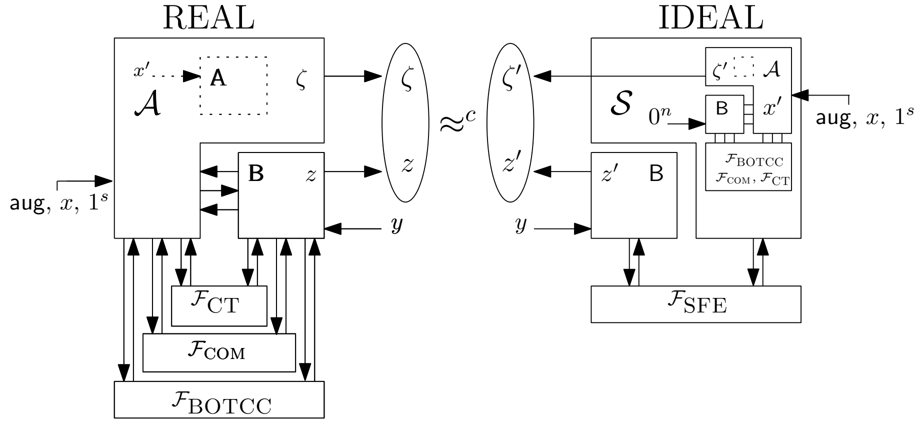


Figure 4.5: The ideal and the real world execution with corrupt A ($\mathcal{A}$).

- The simulator continues and chooses two random universal hash functions in the *Augmentation* phase and then runs the *Cut-and-Choose*, *Evaluation*, and *Reconstruction* phases.
- At the end of the *Evaluation* phase an honest B (and in this case the simulator) will have discarded any inconsistent garbled circuits received by $\mathcal{A}$. Thus what remains are garbled circuits that can be evaluated. If they all give the same output then the simulator will use one of $\mathcal{A}$'s input vectors extracted during the *Oblivious Transfer* phase to one of the evaluable garbled circuits and input such a vector to $\mathcal{F}_{\text{SFE}}$. Otherwise, it will continue simulating an honest B in the *Reconstruction* phase and in turn find $\mathcal{A}$'s input to each of the garbled circuits. In particular S will, like honest B, then evaluate the function $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ in plain and compare the plain output with the semantic meaning of the output he learned in *Evaluation* to find an input of $\mathcal{A}$ given to the correctly constructed circuits. S then uses such an input of $\mathcal{A}$ as input to $\mathcal{F}_{\text{SFE}}$. Notice that there can only be one suitable set of inputs (except with probability 2^{-s}) since having more would imply that either the verifiable commitments on the output keys of $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ were incorrectly constructed, in which case the simulator would have discarded the circuit, or $\mathcal{A}$ has guessed a collision for $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$, which can only happen with probability 2^{-s}.

Our first observation is that in the real (hybrid) world execution B's input $\bar{x_{\mathsf{B}}}$ is only used as input to the ideal functionality computing $\mathcal{F}_{\text{BOTCC}}^{\bar{n_{\mathsf{B}}},\ell,\kappa}$. Consequently, the messages that B sends during the remaining part of the protocol and hence the view of $\mathcal{A}$ do not depend on $\bar{x_{\mathsf{B}}}$. Therefore, the output of $\mathcal{A}$ in the real world execution and S in the ideal world are identical.

We must, however, prove that the *joint* distribution of $\mathcal{A}$'s and B's output in the real world execution is identical to the *joint* distribution of S's and B's output in the ideal world, except with negligible probability in κ. This involves proving the following cases:

1. Consider all steps of the protocol except for the reconstruction phase. For all these steps we must argue that B outputs `abort` with the same probability in the ideal and real worlds. In the ideal world B outputs `abort` if and only if S sends `abort` to $\mathcal{F}_{\text{SFE}}$. By construction S outputs `abort` if and only if B aborts in the simulation. But the probability that B aborts may in general depend on his input, which is x_{B} in the real execution and $0^{n_{\mathsf{B}}}$ in the simulation. So we must argue that the probability that B aborts does not depend on whether his input is x_{B} or $0^{n_{\mathsf{B}}}$.

2. In the cases where B does not abort before the reconstruction phase we must argue that, except with probability negligible in the security parameter, he does not abort during reconstruction and that his output in the real execution is $f(x'_{\mathsf{A}} \| x_{\mathsf{B}})$ where x'_{A} is the same input for $\mathcal{A}$ that S inputs to $\mathcal{F}_{\text{SFE}}$ in the ideal world. Our strategy for this is the following:

   a) We show that after a certain point in the protocol, even a malicious $\mathcal{A}$ is committed to her inputs ${x'_{\mathsf{A}}}^j$ for each of the ℓ garbled circuits.

   b) We show how S can extract ${x'_{\mathsf{A}}}^j$ for all $j \in [\ell]$ from $\mathcal{A}$.

   c) Then we show that both S and honest B will use the ${x'_{\mathsf{A}}}^j$ as input for a circuit that is correctly garbled, committed and where the verifiable commitments on the output keys of $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ are also correctly constructed.

   d) We then show that such a circuit exists except with negligible probability in the security parameter, if not he will abort.

   e) Next we show that there cannot exist several good circuits where the input $\mathcal{A}$ is committed to is different.

   f) Finally, we argue that even with a malicious $\mathcal{A}$, B outputs $f(x'_{\mathsf{A}} \| x_{\mathsf{B}})$ except with negligible probability, in particular that the real B will compute $f({x'_{\mathsf{A}}}^j \| x_{\mathsf{B}})$ for the same ${x'_{\mathsf{A}}}^j$ as the simulator, if not he will abort.

Furthermore, see that everything sent to $\mathcal{A}$ by the simulator has exactly the same distribution as in a real world execution (because of its internal simulation of $\mathcal{F}_{\text{BOTCC}}$).

Now, we say that for $j \in [\ell]$ the garbled circuit and decoding information (F_j, d_j) sent in *Commitment* is *good* iff the verification procedure accepts it: $\mathsf{Ve}\left(F_j, \bar{f}, e_j, d_j, r_j\right) \to 1$ where $\bar{f}$ is the augmented function decided in *Setup* and $e_j \to \left(X^0_{1,j}, X^1_{1,j}, \ldots, X^0_{\bar{n},j}, X^1_{\bar{n},j}\right)$, where the first $2 \cdot \bar{n_{\mathsf{A}}}$ elements are the $\bar{n_{\mathsf{A}}}$ pairs of elements $\mathcal{A}$ committed to in the verifiable commitments from *Commitment* with the IDs $\mathsf{ID}^{\mathsf{A-order}}_{i,j}$ for $i \in [\bar{n_{\mathsf{A}}}]$. It should further hold that the last $2 \cdot \bar{x_{\mathsf{B}}}$ elements are the pairs $\mathcal{A}$ used as input to the $\mathcal{F}_{\text{BOTCC}}$ functionality in *Oblivious Transfer*. A garbled circuit that is not good is called *bad*.

We call a verifiable commitment to an output key good iff the ID of the commitment reflects the key committed to. For example, the commitment with ID $\mathsf{ID}^{\mathsf{aug0}}_{i,j}$ should be made to the i'th output 0-key when computing the augmentation $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ in the j'th garbled circuit.

Proof of Case 1. We now consider all the steps in which B can output `abort`:

Polynomial Setup If an opening of one of the commitments to the polynomials, after the cut-and-choose step outputs `reject` or the polynomials are not of correct degree then B aborts. However, which polynomials are checked is completely determined by B's challenge and is thus independent of his input.

Cut-and-Choose B terminates in this phase if the verification of any of the garbled circuits fail, if $\mathcal{A}$ has not committed to the correct input keys for each of the garbled circuits, if any commitment to the output keys of the garbled circuit or $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ is bad, or if there is a problem with the verifiable commitments to the polynomial points, found using the links. That is, B first checks the verifiable commitments on the concatenation of a pair of $\mathcal{A}$'s input keys, using e_j for $j \in \pi$. He also checks that the keys he got from the *Oblivious Transfer* are consistent with the tokens in e_j. However, $\mathcal{A}$ could have given something else than the right key as input during to the OT, that is, launching a selective failure

attack. However, as proved in [LP15] if $\bar{n_{\mathsf{B}}} = \max(4 \cdot n_{\mathsf{B}}, 8 \cdot s)$ then aborting or not, will be independent of B's true input except with negligible difference when s is chosen s.t. 2^{-s} is negligible in κ. We notice that the rest of the checks in this phase are independent of B's input. In particular this includes the verification step, $\mathsf{Ve}\left(F_j, \bar{f}, e_j, d_j, r_j\right)$. To see this assume for the sake of contradiction that an adversary could construct a garbled circuit with associated encoding/decoding information such that the verification accepts, but that the output of the evaluation might depend on the input. If that was the case then we could use this adversary to win the $\text{Ver}_{\mathcal{G}}$ game with non-negligible probability. We simply take the tuple $(F_j, \bar{f_j}, e_j, d_j, r_j)$ and input to the game.
Furthermore, all of these cases depend on the cut-and-choose challenge which is chosen by coin tossing[6] and thus independent of B's input.

Evaluation If the openings of the commitments of $\mathcal{A}$'s choice of input keys outputs `reject`, if she committed to garbage, or if any of the verifiable commitments to the output keys of $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ are bad, then B will output `abort`. However, the computation of $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ is only based on $\mathcal{A}$'s input and thus is independent of B's input. Finally, B will output `abort` if he cannot evaluate any of the garbled evaluation circuits. Whether B aborts might be based on his input as he learns only one of the keys for each wire in each circuit in the *Oblivious Transfer* phase. However, if $\mathcal{A}$ tries to do a selective failure attack during the OTs she can only learn a negligible amount of information about B's true input because what B actually inputs to the protocol is $\bar{x_{\mathsf{B}}}$, which by construction seems uniformly random.

Proof of Case 2.a and 2.b. We first observe that in the protocol even a malicious $\mathcal{A}$ is committed to a value associated with each of her input wires for all the garbled circuits after the *Commitment* phase. The simulator can furthermore extract each of these values since $\mathcal{A}$ commits to these using $\mathcal{F}_{\text{COM}}$. Furthermore, the simulator can find the semantic values (if the circuits can be evaluated) for each of $\mathcal{A}$'s commitments by extracting $\mathcal{A}$'s "order commitments" (The *Commitment* phase, step 2). More specifically what S does is as follows:

- In Step 2 of the *Commitment* phase, S learns $X^0_{i,j} \| X^1_{i,j}$ for all $i \in [\bar{n_{\mathsf{A}}}]$ and $j \in [\ell]$ directly by extracting them from $\mathcal{A}$'s calls to $\mathcal{F}_{\text{COM}}$.
- Then in Step 3 of the same phase S learns $X_{i,j}^{x'^{\,j}_{\mathsf{A}i}}$ for all $i \in [\bar{n_{\mathsf{A}}}]$ and $j \in [\ell]$.
- For each $i \in [\bar{n_{\mathsf{A}}}]$ and $j \in [\ell]$ S learns the bits $\mathcal{A}$ committed to by checking if $X_{i,j}^{\bar{x'}^{\,j}_{\mathsf{A}i}} = X^0_{i,j}$ or $X_{i,j}^{\bar{x'}^{\,j}_{\mathsf{A}i}} = X^1_{i,j}$. If the first is true then $\bar{x'}^{\,j}_{\mathsf{A}i} = 0$, if the second is true then $\bar{x'}^{\,j}_{\mathsf{A}i} = 1$.

Proof of Case 2.c. Next, notice that some of these commitments may not be actual keys to their associated garbled circuit but instead just random garbage. In that case both S and B will discard their associated garbled circuits during the *Evaluation* phase (or abort the protocol during *Cut-and-Choose*). So, unless the protocol has aborted by the end of the *Evaluation* phase, at least one garbled circuit for evaluation will exist where each of the inputs committed to by $\mathcal{A}$ are actual keys for the corresponding garbled circuit. However, there might be more than one such circuit and $\mathcal{A}$ might have committed to keys with different semantic values in each of these. Still, if that is the case then the augmented output of $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ will be different for each of the circuits where she has given different inputs except with probability 2^{-s}.

[6]Failure of $\mathbb{H}^{\mathsf{Out}}$ also depends on the cut-and-choose of polynomials but we have already showed that such failure is independent of B's input.

To see this we must show that it will not be possible for $\mathcal{A}$ to find a collision on $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ with probability larger than 2^{-s}. That is, it is not possible for her to give two inputs $x_{\mathsf{A}} \| \alpha$ and $x'_{\mathsf{A}} \| \alpha' \in \{0,1\}^{\bar{n_{\mathsf{A}}}}$ with $x_{\mathsf{A}} \neq x'_{\mathsf{A}}$ such that $(M^{\mathsf{In}} \cdot x_{\mathsf{A}}) \oplus \alpha = (M^{\mathsf{In}} \cdot x'_{\mathsf{A}}) \oplus \alpha'$ with probability larger than 2^{-s}. Now remember that we have from Lemma 7 that $\mathbb{H}^{\mathsf{In}}$ computes a universal hash function with the uniform difference property, i.e., that $\mathsf{Pr}\left[\left(M^{\mathsf{In}} \cdot x_{\mathsf{A}}\right) \oplus \left(M^{\mathsf{In}} \cdot x'_{\mathsf{A}}\right) = \bar{\alpha}\right] = 2^{-s}$ for any $\bar{\alpha} \in \{0,1\}^s$. It turns out that the uniform difference property is enough to ensure $\mathcal{A}$ cannot find a collision. To see this notice that what we actually need to show is $\mathsf{Pr}\left[\left(M^{\mathsf{In}} \cdot x_{\mathsf{A}}\right) \oplus \alpha = \left(M^{\mathsf{In}} \cdot x'_{\mathsf{A}}\right) \oplus \alpha'\right] \leq 2^{-s}$ for all $x_{\mathsf{A}} \| \alpha$, $x'_{\mathsf{A}} \| \alpha' \in \{0,1\}^{\bar{n_{\mathsf{A}}}}$ with $x_{\mathsf{A}} \neq x'_{\mathsf{A}}$ when M^{In} is uniformly sampled. However, setting $\bar{\alpha} = \alpha \oplus \alpha'$ we see

$$\begin{aligned}\left(M^{\mathsf{In}} \cdot x_{\mathsf{A}}\right) \oplus \left(M^{\mathsf{In}} \cdot x'_{\mathsf{A}}\right) &= \bar{\alpha} \\ \left(M^{\mathsf{In}} \cdot x_{\mathsf{A}}\right) \oplus \left(M^{\mathsf{In}} \cdot x'_{\mathsf{A}}\right) &= \alpha \oplus \alpha' \\ \left(M^{\mathsf{In}} \cdot x_{\mathsf{A}}\right) \oplus \alpha &= \left(M^{\mathsf{In}} \cdot x'_{\mathsf{A}}\right) \oplus \alpha' \ ,\end{aligned}$$

and thus we have $\mathsf{Pr}\left[\left(M^{\mathsf{In}} \cdot x_{\mathsf{A}}\right) \oplus \alpha = \left(M^{\mathsf{In}} \cdot x'_{\mathsf{A}}\right) \oplus \alpha'\right] = 2^{-s}$ and we are done.

Next see that both B and the simulator will see if the evaluation of $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ is correct for the circuits in ϕ during the *Evaluation* phase and abort if that is not the case. However, $\mathcal{A}$ could commit to different inputs without being detected by corrupting the verifiable commitments of the output keys of $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$. Still, we force $\mathcal{A}$ to commit to the output keys before the *Cut-and-Choose* phase, so if she chooses to cheat we then have from Lemma 10 that she cannot successfully do this for all evaluation circuits and thus there will be at least one circuit for which the verifiable commitments to the output keys of $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ are correctly constructed. Furthermore, one of these must be associated with some correct input keys as the protocol would otherwise terminate during *Evaluation*.

We have now established that there must be a vector of input keys from $\mathcal{A}$ along with good verifiable commitments to the output keys of $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$. However, there might still be instances of input keys that fit their associated circuit but where $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ is incorrectly constructed, e.g., so that it gives the same digest as the correctly constructed case. In this situation we must now argue that both B and S can find out which of the garbled circuits and its corresponding $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ augmentation is correct. In particular, we have two cases:

- Either the semantic output of the garbled circuit is the same in both cases and there is no issue for B since the correct output is known in the real world and B can output this. In the simulation S can easily find out which circuit is correct since it can extract the semantic value of $\mathcal{A}$'s input keys as described above in the proof of 2.a and 2.b. The input for a correct circuit whose associated $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ augmentation is correct S inputs to $\mathcal{F}_{\mathrm{SFE}}$.

- If the semantic output of the garbled circuits are different the simulator can do the same as in the case above or it can do the same as a real-world B would; use the *Reconstruction* phase to extract ${x'_{\mathsf{A}}}^j$ for all $j \in \phi$ committed to by $\mathcal{A}$ and evaluate $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ of the disputed circuits in plain and find out which input is used in a correctly constructed circuit and then use this input to evaluate the function f in plain.

Proof of 2.d. The fact that at least one good circuit exists or both S and B will terminate the protocol follows directly from Lemma 10.

Proof of 2.e. The fact that we cannot have several correctly constructed augmented garbled circuits evaluate to different outputs stems from the fact that if they evaluate differently then $\mathcal{A}$'s input must be different. However, if that is the case then $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$, except with probability 2^{-s}, will output different values and thus both B and S will terminate the protocol by Lemma 7 and the discussion of case 2.c.

Proof of Case 2.f. We finally argue that even with a malicious $\mathcal{A}$, if x'_{A} is the semantic value of the input that $\mathcal{A}$ commits to in the *Commitment* phase, associated with a correctly constructed augmented garbled circuit, then $f(x'_{\mathsf{A}} \| x_{\mathsf{B}})$ will be the output of B if he does not abort.

To see this we only need to argue that in the real world, and despite a corrupted A, B outputs $\mathsf{ev}(f, x'_{\mathsf{A}} \| x_{\mathsf{B}})$ where x'_{A} is the input that $\mathcal{A}$ commits to in the *Commitment* Step. To show this we consider the following lemma:

Lemma 11. *Assume we have c evaluation circuits and w output wires where at most half are bad, then the probability that we cannot find the global difference for each circuit, and thus might fail the reconstruction, is at most*

$$\frac{1 + (w/2) \cdot (2^c - 1) + \sum_{i=2}^{w/2} \left(\binom{w/2}{i} \cdot c \cdot 2^{i(c-1)} \right)}{2^{c \cdot w/2}} .$$

Proof. In the following denote the bit value on the i'th wires in the j'th circuit by $b_{i,j}$. We now prove this lemma using a counting argument. First notice that the total amount of cases of c circuits with w wires are $2^{c \cdot w}$ since we can view it as a set of $c \cdot w$ binary wires where each outcome is possible. We now count the number of "bad" configurations of these wires, i.e., configurations of the polynomial points associated with the output wires in the evaluation circuits that does not make it possible to do interpolations such that we can learn the global difference used in each of the c circuits. First define a *very bad* wire to be a wire whose associated polynomial is not correctly constructed. Any very bad wire will be completely useless. Furthermore, the position of a very bad wire does not matter as all wires are equally usable in the manner that they can be used to restore Δ_j for all $j \in [c]$. Next define an *AND0 wire* to be a wire where the output bits on this wire from all c evaluation circuits AND'ed together gives 0, assuming it is not a very bad wire. Notice that for a given wire on c circuits there is only one configuration where the bits AND together to 1, that is for the all 1-string.

Notice that in the view of finding Δ_j for all $j \in [c]$ the positions of very bad wires does not matter, as any wire that is not very bad can be used to do polynomial interpolation and thus find global differences. Next see that because of the potential existence of very bad wires, the worst cases will always be when we have $w/2$ very bad wires. So we can simply consider bad cases on the remaining $w/2$ wires. The reason being that if we can restore the Δ values with only $w/2$ wires then we can always restore it with more wires. This applies no matter the configuration of the bits on the extra wires, as we can simply choose to ignore them in the restoration process. So in the following we only count configurations of the $w/2$ wires which are not very bad. Now see that all bad configurations are captured in the following, possibly intersecting, sets:

1. All configurations when we do not have any AND0 wires.
2. All configurations when we only have a single AND0 wire.

3. For all $i \in [w/2]\backslash\{1\}$, all configurations when we have i AND0 wires and $\exists j \in [c]$: $\vee_{k=1}^{i} b_{k,j} = 0$. That is, the cases where the OR of the bits on each of the i wires in any of the c circuits results in 0.

If the first case happens then we cannot do any polynomial interpolation and thus must be included. However, this case only occurs when the values on all the $w/2$ wires are 1's, i.e., only on one configuration.

For the second case, assume the AND0 wire is wire i, then if this case occurs we will only be able to do interpolation on wire i, and in turn only be able to find Δ_j for each circuit j when $b_{i,j} = 1$ (which cannot occur for all circuits since wire i must have at least one 0-bit). Now see that a wire is an AND0 if and only if it does not purely output 1-bits. Thus there are $2^c - 1$ possible ways a given wire can be of the AND0 type. Furthermore, notice that there are $w/2$ ways of selecting an AND0 wire. Finally see that when we only have a single AND0 wire, the values on the remaining wires must all be the 1-string. So we get a total of $(w/2) \cdot (2^c - 1)$ bad configurations in this case.

The third case is a bit harder to show and we consider it in an inductive manner. I.e., first consider the case of two AND0 wires, wlog wire 1 and 2, and the event that $\exists j \in [c]$: $b_{1,j} \vee b_{2,j} = 0$. This means that for circuit j we will only be able to learn 0 keys on the two wires we are able to interpolate (the AND0 wires) and thus not be able to find Δ_j. Now to count configurations see that bits only OR to 0 if all of them are 0, so it must be the case that $b_{1,j} = b_{2,j} = 0$. Now since each AND0 wire has c bits, we see that the amount of remaining configurations of these two wires are at most[7] $2^{2(c-1)}$, since each wire constitutes 2^{c-1} possible configurations (once the j'th bit has been set to 0). However, this occurs no matter which value j takes, i.e., it happens once for each of the c possible values j can take. That is at most $c \cdot 2^{2(c-1)}$ cases. Finally see that we can select two AND0 wires in $\binom{w/2}{2}$ different ways, so we get $\binom{w/2}{2} \cdot c \cdot 2^{2(c-1)}$ cases in total.

Now consider the general case where we have i AND0 wires. There must be no circuit where the bits of these i wires OR to 0. When they OR to 0 the remaining configuration of the wires are bad, i.e., at most $2^{i(c-1)}$ configurations. Again it is bad no matter for what circuit the wires OR to 0, so in total $c \cdot 2^{i(c-1)}$ cases. Also, these i AND0 wires can be chosen in $\binom{w/2}{i}$ ways. So in total $\binom{w/2}{i} \cdot c \cdot 2^{i(c-1)}$ bad configurations.

Finally we add all the cases together and get an upper bound on bad cases:

$$1 + (w/2) \cdot (2^c - 1) + \sum_{i=2}^{w/2} \left(\binom{w/2}{i} \cdot c \cdot 2^{i(c-1)} \right) .$$

Since the cases are randomly distributed (because of cut-and-choose and the universal hash function is uniform), dividing this number with the total amount of possible cases ($2^{c \cdot w/2}$) yields the desired result. □

Notice that this analysis is not tight as our argumentation only considers a subset of all good cases. That is, there are several bad cases that get counted several times.

According to Lemma 11 we can bound the probability that a corrupted $\mathcal{A}$ passes the cut-and-choose test of polynomials *and* that we cannot restore all the global differences for the evaluable circuits. When we use the following lemma we get that the amount of wires needed in order to make sure that we can find Δ_j for all $j \in \phi$ except with probability at most 2^{-s}:

[7] We count some redundant configurations here.

Lemma 12. *Assuming the polynomials on at least half of the wires are good then we achieve failure probability at most* 2^{-s} *for* $s \geq 1$ *if we have at least* $w = \lceil 4.82s + 4.82 \rceil \geq -\frac{s}{\frac{\log(3)}{2}-1} - \frac{1}{\frac{\log(3)}{2}-1}$ *output wires.*

Proof. First observe that the optimal strategy for $\mathcal{A}$, if she wishes to succeed in cheating, is to construct only two circuits that evaluate differently (assuming that the amount of circuits is less than the amount of output wires). This is so since the equation $\frac{1+(w/2)\cdot(2^c-1)+\sum_{i=2}^{w/2}\left(\binom{w/2}{i}\cdot c\cdot 2^{i(c-1)}\right)}{2^{c\cdot w/2}}$ is maximized for $c = 2$ under the constraint that $1 < c < w$, $c \in \mathbb{Z}$ and $w \geq 8$. Notice that we need $c > 1$ since otherwise there is no chance of failure as the output will be the same in all circuits. Obviously we also need $c \in \mathbb{Z}$. Finally notice that $c < w$ and $w \geq 8$ are constraints arriving as artifacts of the equation.

Under these constraints consider the probability of the worst case, i.e. that the output is discrepant on 2 circuits:

$$
\begin{aligned}
&\frac{1+(w/2)\cdot(2^2-1)+\sum_{i=2}^{w/2}\left(\binom{w/2}{i}\cdot 2\cdot 2^{i(2-1)}\right)}{2^{2\cdot w/2}} \\
&= \frac{1+3\cdot(w/2)+\sum_{i=2}^{w/2}\left(\binom{w/2}{i}\cdot 2\cdot 2^{i}\right)}{2^{w}} \\
&= \frac{1+3\cdot(w/2)+2\cdot\sum_{i=2}^{w/2}\left(\binom{w/2}{i}\cdot 2^{i}\right)}{2^{w}} \\
&= \frac{1+3\cdot(w/2)+2\cdot\left(\sum_{i=0}^{w/2}\left(\binom{w/2}{i}\cdot 2^{i}\right)-w-1\right)}{2^{w}}\ .
\end{aligned}
$$

Using the Binomial Theorem we get the following:

$$
\begin{aligned}
&\frac{1+3\cdot(w/2)+2\cdot\left(\sum_{i=0}^{w/2}\left(\binom{w/2}{i}\cdot 2^{i}\right)-w-1\right)}{2^{w}} \\
&= \frac{1+3\cdot(w/2)+2\cdot\left((1+2)^{w/2}-w-1\right)}{2^{w}} \\
&= \frac{1+3\cdot(w/2)+2\cdot 3^{w/2}-2\cdot w-2}{2^{w}} \\
&= \frac{2\cdot 3^{w/2}-(w/2)-1}{2^{w}} \\
&= 2\cdot 2^{-w}\cdot 3^{w/2}-2^{-w}\cdot(w/2)-2^{-w} \\
&= 2^{-w+1}\cdot 2^{\log(3)\cdot w/2}-2^{-w}\cdot((w/2)+1) \\
&= 2^{-w+1+\log(3)\cdot w/2}-2^{-w}2^{\log((w/2)+1)} \\
&= 2^{1+w\left(\frac{\log(3)}{2}-1\right)}-2^{-w+\log((w/2)+1)} \\
&\leq 2^{1+w\left(\frac{\log(3)}{2}-1\right)}-2^{1-w}
\end{aligned}
$$

The last step follows when $w \geq 2$. So we finally need to show that $2^{1+w\left(\frac{\log(3)}{2}-1\right)}-2^{1-w} \leq 2^{-s}$.

To do so we replace w by our hypothesis term, $w = -\frac{s}{\frac{\log(3)}{2}-1} - \frac{1}{\frac{\log(3)}{2}-1}$.

$$\begin{aligned}
2^{-s} &\geq^{?} 2^{1+\left(-\frac{s}{\frac{\log(3)}{2}-1}-\frac{1}{\frac{\log(3)}{2}-1}\right)\cdot\left(\frac{\log(3)}{2}-1\right)} - 2^{1-\left(-\frac{s}{\frac{\log(3)}{2}-1}-\frac{1}{\frac{\log(3)}{2}-1}\right)} \\
&= 2^{1+(-s-1)} - 2^{1+\frac{s}{\frac{\log(3)}{2}-1}+\frac{1}{\frac{\log(3)}{2}-1}} \\
&= 2^{-s} - 2^{1+\frac{s}{\frac{\log(3)}{2}-1}+\frac{1}{\frac{\log(3)}{2}-1}}
\end{aligned}$$

Next observe that $2^{1+\frac{s}{\frac{\log(3)}{2}-1}+\frac{1}{\frac{\log(3)}{2}-1}} \approx 2^{-4.82s-3.82}$. Thus for $s \geq 1$ we have that

$$2^{-s} - 2^{1+\frac{s}{\frac{\log(3)}{2}-1}+\frac{1}{\frac{\log(3)}{2}-1}} < 2^{-s}.$$

Finally, since the amount of evaluation circuits are less than s for $s \geq 1$ then $c < w \Rightarrow 2 \cdot s < \lceil 4.82s + 4.82 \rceil$ is clearly true and we are done. □

Next, to bound the amount of polynomials needed in the cut-and-choose challenge, remember that we need at least half of the $\lceil 4.82s+4.82 \rceil$ evaluation polynomials to be good. Thus there is a total of $\binom{p}{p - \lceil 4.82s + 4.82 \rceil}$ possible ways to choose the check polynomials. If more than half of the evaluation polynomials are "bad" then there is a total of $\binom{p - \left(\frac{\lceil 4.82s+4.82 \rceil}{2} + 1\right)}{p - \lceil 4.82s + 4.82 \rceil}$ "bad" cut-and-choose challenges. Since there is only a set of $p - \left(\frac{\lceil 4.82s+4.82 \rceil}{2} + 1\right)$ good polynomials to choose from and the check size remains $p - \lceil 4.82s + 4.82 \rceil$. Thus the probability of a "bad" cut-and-choose challenge is

$$\frac{\binom{p - \left(\frac{\lceil 4.82s+4.82 \rceil}{2} + 1\right)}{p - \lceil 4.82s + 4.82 \rceil}}{\binom{p}{p - \lceil 4.82s + 4.82 \rceil}}$$

We must now find a value for p such that this expression is less than 2^{-s}. According to the following lemma this is the case if we set $p \geq 6s + 7$.

Lemma 13.

$$\frac{\binom{6s + 7 - \left(\frac{\lceil 4.82s+4.82 \rceil}{2} + 1\right)}{6s + 7 - (\lceil 4.82s + 4.82 \rceil)}}{\binom{6s+7}{6s + 7 - (\lceil 4.82s + 4.82 \rceil}} \leq 2^{-s}$$

for $s > 1$.

Proof. Let

$$\gamma = \frac{\binom{6s + 7 - \left(\frac{\lceil 4.82s+4.82 \rceil}{2} + 1\right)}{6s + 7 - \lceil 4.82s + 4.82 \rceil}}{\binom{6s+7}{6s + 7 - \lceil 4.82s + 4.82 \rceil}}.$$

Now consider $\frac{2^{-s}}{\gamma}$. If we can show that this is an increasing function for $s > 1$, then we are done, as this means that 2^{-s} is growing faster than γ and in turn that $\gamma < 2^{-s}$. We see this by taking the derivative of $\frac{2^{-s}}{\gamma}$ and noticing this is always positive. □

Next see that at least half of the polynomials used to augment the output wires of $\mathbb{H}^{\mathsf{Out}}$ are good (the event in Lemma 12) then we have exactly the case described in the *Reconstruction* phase except if some of the links of $\mathbb{H}^{\mathsf{Out}}$ are wrong. However, since $\mathcal{A}$ sends the links before the cut-and-choose phase then Lemma 10 tells us that there will be at least one circuit where the links are correctly constructed. In turn B will always be able to recover the input x'_{A} used by $\mathcal{A}$.

Whether a garbled circuit is good or bad is determined before partitioning them into check and evaluation circuits in *Cut-and-Choose*. Thus when using a replication factor of ℓ (which is the replication factor actually used in the protocol), the random partitioning into check and evaluation circuits will ensure that there will be no good evaluation circuit with probability at most $2^{-s} \leq 2^{-\left(\ell - \frac{1}{2}\log \ell + \log\left(2\frac{\sqrt{2\pi}}{e^2}\right)\right)}$ (by Lemma 10).

This completes the proof for the case where A is corrupt. □

Corrupt B

Lemma 14. *The protocol* Π_{COM} *from Section 4.3 UC-securely realizes* $\mathcal{F}_{\mathrm{SFE}}$ *in Fig. 4.1 in the* ($\mathcal{F}_{\mathrm{BOTCC}}$, $\mathcal{F}_{\mathrm{CT}}$, $\mathcal{F}_{\mathrm{COM}}$)*-hybrid model against any PPT static and malicious adversary corrupting* B *assuming that* $\mathcal{G}$ *is a key size preserving, structure free, free-XOR gate garbling scheme with the* prv.ind *and* ver *properties and leakage function* $\mathbf{\Phi}_{\mathsf{xor}}$ *in accordance with the definitions in Chapter 2 and the link scheme* (G, F) *used is secure according to Definition 12. Furthermore, assume that there is at least one non-XOR gate on the path from an output wire to some input wire.*

Proof. The intuition for this part of the proof is quite similar to most other two-party protocols based on garbled circuits with cut-and-choose. Plainly speaking a malicious B cannot cheat since all the input he gives to the protocol is his plain input bits (to $\mathcal{F}_{\mathrm{COM}}$) and some random choices, which are used only to protect himself against a potentially malicious A.

Still, from a simulation point of view, we must argue that the simulated $\mathcal{B}$ outputs the same as $\mathcal{B}$ in the real (hybrid) execution, including the literal output of the protocol $\bar{f}(\bar{x_{\mathsf{A}}} \| \bar{x_{\mathsf{B}}}) = y$. This is not as obvious as in the case of a corrupted A and is achieved by S learning the cut-and-choose challenge directly from $\mathcal{F}_{\mathrm{CT}}$ and then constructing the evaluation garbled circuits "maliciously" such that they always output $\bar{f}(\bar{x_{\mathsf{A}}} \| \bar{x_{\mathsf{B}}}) = y$, no matter what the semantic value of the input is. In particular it will still give the correct output even if the $\bar{x_{\mathsf{A}}}$ used in the garbled circuit is the all-0 string.

The formal proof proceeds as follows: Given an adversary $\mathcal{B}$ that controls B, we construct a simulator S in a way analogous to the case where A is corrupted. Formally the simulator S works as follows:

- S starts simulating internally the real world execution of the protocol consisting of A, $\mathcal{B}$, and the ideal functionalities $\mathcal{F}_{\mathrm{BOTCC}}^{\bar{n_{\mathsf{B}}},\ell,\kappa}$, $\mathcal{F}_{\mathrm{CT}}$, and $\mathcal{F}_{\mathrm{COM}}$. If A aborts at any time during the simulation, then S sends `abort` to $\mathcal{F}_{\mathrm{SFE}}$.

- The simulator then executes the *Setup* and *Polynomial Setup* phases as an honest A would.

- When reaching the *Oblivious Transfer* phase S first extracts $\mathcal{B}$'s input $\bar{x_\mathsf{B}}'$ from his call to $\mathcal{F}^{\bar{n_\mathsf{B}},\ell,\kappa}_{\text{BOTCC}}$. S then computes $M^{\mathsf{Sec}} \cdot \bar{x_\mathsf{B}}' = x'_\mathsf{B}$ and sends x'_B to $\mathcal{F}_{\text{SFE}}$, which returns $y = f(x_\mathsf{A} \| x'_\mathsf{B})$. Then it simulates $\mathcal{F}_{\text{CT}}$ to select the partition of check and verification circuits. It then garbles all the check circuits correctly. For the evaluation circuits it defines a new function $\bar{f}'$ which computes $y = \bar{f}'(0^{n_\mathsf{A}} \| \alpha' \| \bar{x_\mathsf{B}}')$ with $\alpha' \in_R \{0,1\}^s$ where $\mathbf{\Phi}_{\mathsf{xor}}(\bar{f}) = \mathbf{\Phi}_{\mathsf{xor}}(\bar{f}')$.[8] Now it garbles the evaluation circuits using this function. Next, simulating $\mathcal{F}^{\bar{n_\mathsf{B}},\ell,\kappa}_{\text{BOTCC}}$, S returns the keys $X^{\bar{x_\mathsf{B}}'[i]}_{\bar{n_\mathsf{A}}+i,j}$ for $i \in [\bar{n_\mathsf{B}}]$ and $j \in [\ell]$ to $\mathcal{B}$.

- In the *Commitment* phase S sends the garbled circuits and their corresponding decoding information it just constructed to $\mathcal{B}$. It then makes verifiable commitments to A's order commitments by simulating $\mathcal{F}_{\text{COM}}\left(\mathsf{vc}, \mathsf{A}, \mathsf{sid}, \mathsf{ID}^{\mathsf{A-order}}_{i,j}, X^0_{i,j} \| X^1_{i,j}\right)$ and to her simulated input by simulating the calls $\mathcal{F}_{\text{COM}}\left(\mathtt{commit}, \mathsf{A}, \mathsf{sid}, \mathsf{ID}^{\mathsf{A-in}}_{i,j}, X^{\bar{x_\mathsf{A}}'[i]}_{i,j}\right)$ for $i \in [\bar{n_\mathsf{A}}]$ and $j \in [\ell]$ where $\bar{x_\mathsf{A}}' = 0^{n_\mathsf{A}} \| \alpha'$.

  The rest of the commitments in the *Commitment* phase S constructs as an honest A would.

- In the *Augmentation* phase S receives the specification for $\mathbb{H}^{\mathsf{In}}$ and $\mathbb{H}^{\mathsf{Out}}$ from $\mathcal{B}$. For both the check and evaluation circuits S then constructs the verifiable commitments to the output keys of $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ along with the links, as an honest A would.

- For *Cut-and-Choose* it outputs to $\mathcal{B}$ his part of the simulated $\mathcal{F}_{\text{CT}}$ call from *Oblivious Transfer*. It then executes the rest of *Cut-and-Choose*, *Evaluation*, and *Reconstruction* phases as an honest A would.

To prove security we must show the following:

1. We must argue that A outputs `abort` with the same probability in the real and ideal world. In particular that the probability with which she aborts does not depend on whether or not her input is $x_\mathsf{A} \| \alpha$ or $0^{n_\mathsf{A}} \| \alpha'$ with $\alpha, \alpha' \in_R \{0,1\}^s$.

2. If A does not abort then we must argue that A outputs the same in the real and ideal world.

3. We need to argue that the output of $\mathcal{B}$ is the same in the real and ideal world. In particular that $\mathcal{B}$ does not learn anything that depends on A's input x_A except $f(x_\mathsf{A} \| x'_\mathsf{B}) = y$.

Together, the steps above ensure that the *joint* distributions of $\mathcal{B}$'s output and A's output in the ideal and in the real world are indistinguishable.

Proof of Case 1. Notice that an honest A following the protocol will not abort except in the trivial cases where the adversary blocks the communication or she received garbage.

Proof of Case 2. This case is trivial since A does not receive any output.

[8] One simple way of doing this is to have the last non-XOR gates for an output wire compute a constant value. Thus no matter what the circuit will be evaluated on, the output, when decoded, will always be y. We note that if there are no non-XOR gates for each output wire we can simply insert identity gates into the real functionality f for each output wire at the beginning of the protocol.

Proof of Case 3. This is the hardest case and we must first argue that S can convince $\mathcal{B}$ to learn the true value $y = f(x_{\mathsf{A}} \| x'_{\mathsf{B}})$ with the same probability (except with negligible difference) in the real and ideal world. To do this we first observe that in the real protocol even a corrupt B is committed to a specific input $\bar{x_{\mathsf{B}}}'$ at the time he sends his input to $\mathcal{F}_{\mathrm{BOTCC}}^{\bar{n_{\mathsf{B}}},\ell,\kappa}$. Since S fully controls the simulation of $\mathcal{F}_{\mathrm{BOTCC}}^{\bar{n_{\mathsf{B}}},\ell,\kappa}$, S can therefore easily extract $\bar{x_{\mathsf{B}}}'$ from $\mathcal{B}$'s communication with $\mathcal{F}_{\mathrm{BOTCC}}^{\bar{n_{\mathsf{B}}},\ell,\kappa}$, compute $x'_{\mathsf{B}} = M^{\mathsf{Sec}} \cdot \bar{x_{\mathsf{B}}}'$ and input x'_{B} to $\mathcal{F}_{\mathrm{SFE}}$ (as done in the simulation above). Thus, in the real world, $\mathcal{B}$ learns $f(x_{\mathsf{A}} \| x'_{\mathsf{B}}) = y$ from the protocol execution. This is the same as in the ideal world since S hardcodes the evaluation garbled circuits to be exactly $f(x_{\mathsf{A}} \| x'_{\mathsf{B}}) = y$, learned from $\mathcal{F}_{\mathrm{SFE}}$.

Furthermore, $\mathcal{B}$ cannot learn anything about the semantic values of A's input keys (or any other wire keys in the circuit) for the evaluation circuits because of the prv.ind property of the garbling scheme.

We will argue about this more formally using the following template: First we consider a weaker protocol where the verifiable commitments to the order of the input keys and the output keys of M^{In} are removed and where the links on each output wire i whose output is 1 when evaluated on $\left(M^{\mathsf{Out}} \cdot \bar{f}(\bar{x_{\mathsf{A}}} \| \bar{x_{\mathsf{B}}}')\right) \oplus \beta_b$ are computed as $\mathsf{G}(P_{\chi[i]}(j), r_{i,j})$ where $r_{i,j} \in_R \{0,1\}^\kappa$. We then prove this protocol secure against a malicious $\mathcal{B}$, under the assumption that $\mathcal{G}$ has prv.ind. We then show that we can exchange these "fake" links with the real links and add the verifiable commitments without giving an adversary any non-negligible advantage in breaking the security of the protocol.

More specifically we construct a series of hybrids for the weak protocol and then use a hybrid argument to argue that if a $\mathcal{B}$ can distinguish between any pair of hybrids with non-negligible advantage then we can use him to construct a distinguisher $\mathbb{D}$ that can win the $\mathrm{PrvInd}_{\mathcal{G},\mathbf{\Phi}_{\mathsf{xor}}}^{\mathbb{D}}(1^\kappa)$ game with non-negligible advantage.

First, we formally define the weak protocol. It is exactly the same as the real protocol, except for the following changes:

Commitment

2. Nothing is done.

Augmentation

4. Nothing is done.

6. A is somehow given possession of $\left(M^{\mathsf{Out}} \cdot \mathsf{ev}(\bar{f}, \bar{x_{\mathsf{A}}} \| \bar{x_{\mathsf{B}}})\right) \oplus \beta_b = \tilde{y}$. For $i \in [\lceil 4.82s + 4.82 \rceil]$ and $j \in [\ell]$ A then computes the links $L_{i,j} = \mathsf{G}\left(P_{\chi[i]}(j), \tilde{Y}_{i,j}^0\right)$ where $\tilde{y}[i] = 0$ and $L_{i,j} = \mathsf{G}\left(P_{\chi[i]}(j), r_{i,j}\right)$ where $\tilde{y}[i] = 1$ and $r_{i,j} \in_R \{0,1\}^\kappa$. A then sends these links to B.

We define the simulation of this protocol like the simulator for the real protocol except that it of course does not make the verifiable commitments to the order of the input keys and the output keys of M^{In} and step 6. of the *Augmentation* phase is done as in the weak protocol above.

We next define a sequence of hybrids $\mathrm{H}_0, \mathrm{H}_1, \ldots, \mathrm{H}_{\ell/2}$ where H_0 is exactly the real world execution of the weak protocol and $\mathrm{H}_{\ell/2}$ is exactly the simulation of the weak protocol. The hybrids are constructed such that H_i constructs the first i of the garbled circuits for evaluation like in the simulation. That is, dishonestly constructed such that the non-XOR gates whose output wires are also circuit output wires compute a constant function in correspondence with the output given by $\mathcal{F}_{\mathrm{SFE}}$. Thus these gates evaluate to the output $\mathcal{B}$ would expect in the real

world and where the input for A is substituted with that 0-string (concatenated with a uniformly random α), we call this function $\bar{f}'$. We use the α the real A give as part of her input to the augmented circuit. We get this by cheating and inspecting $\mathcal{F}_{\text{SFE}}$ to learn $\bar{x_\mathsf{A}}\|\alpha$. We note that the α used for each garbled circuit within a given hybrid will be the same. The simulator will honestly construct the $\ell/2-i$ other garbled circuits in H_i along with their decoding information and commitments to A's real input keys like in the real world. Now see that in H_0 everything will be constructed honestly, like in the real protocol.[9] Thus, everything that is sent to $\mathcal{B}$ in this hybrid will be indistinguishable from a real world execution of the weak protocol. Now, we see that the only difference between $\text{H}_{\ell/2}$ and the simulation of the weak protocol is whether or not α is the random string chosen by A or the random string α' chosen by the simulator, and that the simulator does not cheat and extracts A's input from $\mathcal{F}_{\text{SFE}}$, it only uses the output. First notice, that since both the real α in $\text{H}_{\ell/2}$ and the α' picked in the simulation of the weak protocol are chosen uniformly at random thus these two hybrids are indistinguishable, since they are from exactly the same distribution. Furthermore, see that $\text{H}_{\ell/2}$ does not use the extracted input, so it might as well not extract it at all, just like the simulator of the weak protocol.

Now assume, for the sake of contradiction that there exists an adversary $\mathbb{D}$ that can distinguish between the real and simulated world of the weak protocol. Since we already argued that H_0 is indistinguishable from the weak protocol and $\text{H}_{\ell/2}$ is indistinguishable from the simulation of the weak protocol, then he must be able to distinguish between H_{i-1} and H_i for some $i \in [\ell/2]$. Now see that each evaluation circuit is handled independently, which means that in both H_{i-1} and H_i we have the decoding information and commitments for the first $i-1$ evaluation circuits constructed as in the simulation of the weak protocol. Similarly, we have that for the last $\ell/2-i$ evaluation circuits their decoding information and commitments to A's input keys are constructed as in the real world execution of the weak protocol. This means that we can simply focus on the one garbled circuit which is constructed as in the real world in H_{i-1} and as in the simulation in H_i. However, if the adversary can distinguish these two with non-negligible probability we can construct a distinguisher $\mathbb{D}'$ that uses $\mathbb{D}$ to win the $\text{PrvInd}^{\mathbb{D}'}_{\mathcal{G},\mathbf{\Phi}_{\mathsf{xor}}}(1^\kappa)$ game with non-negligible advantage. It does so by using $\text{PrvInd}^{\mathbb{D}'}_{\mathcal{G},\mathbf{\Phi}_{\mathsf{xor}}}(1^\kappa)$ to construct the garbled circuit in question. Specifically it inputs $\left(\bar{f}, \bar{f}', x_0, x_1\right)$ to the game. Here $x_0 = 0^{n_\mathsf{A}}\|\alpha\|\bar{x_\mathsf{B}}$ and $x_1 = x_\mathsf{A}\|\alpha\|\bar{x_\mathsf{B}}$ where α is the uniformly random A used as input (extracted by the simulator). It gets back a garbled circuit, garbled input, and decoding information. It will use this information in place of the garbled circuit, garbled input and decoding information for the circuit in question between H_{i-1} and H_i. It now runs the hybrid with $\mathbb{D}$. If $\mathbb{D}$ guesses H_{i-1} then we input bit 0 to the game, otherwise we input bit 1. We now notice that if the game has sampled bit 0, our execution is exactly the execution of H_{i-1}, if it has instead sampled bit 1, then it is exactly the execution of H_i. Thus we get the same advantage as $\mathbb{D}$ (since A does not abort). However, since we assume that $\mathcal{G}$ has prv.ind then $\mathbb{D}$ cannot do so with more than negligible advantage. Since $\ell/2$ is $\text{poly}(\kappa)$ (as we can assume $\ell \leq \kappa$), it follows from a standard hybrid argument that the weak protocol is secure.

We now consider a sequence of pairs of hybrids, each modeling a real/simulated view. The sequence will contain $\lceil 4.82s + 4.82 \rceil$ pairs. We let pair 0 be the same as the real/simulated view of the weak protocol and pair $\lceil 4.82s + 4.82 \rceil$ be as the real/simulated view of the true protocol, but where the verifiable commitments to the order of A's input keys and the verifiable commitments to the output keys of M^{In} are removed. Now in the i'th pair we change the i'th link from $\mathsf{G}(P_{\chi[i]}(j), r_{i,j})$ to $\mathsf{G}(P_{\chi[i]}(j), \tilde{Y}^0_{i,j})$ for each $j \in \phi$ (if $\tilde{y}[i] = 1$, otherwise we do nothing). We now argue that if an adversary can distinguish the i'th (for $i \in [\lceil 4.82s + 4.82 \rceil]$)

[9] We also note that since everything is constructed honestly the simulator does not inspect $\mathcal{F}_{\text{SFE}}$.

real/simulated pair with non-negligible advantage then he could also distinguish between the $i-1$'th pair. However, having already proved this is not possible for $i-1=0$, we can go through the argument sequentially and achieve a contradiction for each layer. However, we first need a lemma:

Lemma 15. *Given Δ_j for some $j \in \phi$ it is easy to distinguish between the transcript of the real/simulated weak protocol.*

Proof. Given Δ_j this is done by simply looking at the gates whose output are also output of the circuit.[10] For each of these gates we then compute all possible combinations (using the pair of keys already known) if all outputs are the same then we are clearly in the simulation (since the gates are constructed to be constant) if not, then we are clearly in the real setting.

This approach will work, except with negligible probability, because we assumed that the scheme was structure free, and thus a wrong candidate will get rejected, except with negligible probability in κ. □

First see that if $\tilde{y}[i]=0$ there is clearly nothing to show as the hybrids are exactly the same, so in the following we only consider the case of $\tilde{y}[i]=1$. See that the points $\left(P_{\chi[i]}(j)\right)_{j\in\phi}$ on the polynomial associated with output wire i has high min-entropy in the view of the distinguisher. This is because he knows only $\ell/2$ points (through the cut-and-choose) and they lie on a uniformly random sampled polynomial of degree at most $\ell/2$ and all other points are perfectly hidden using the verifiable commitments since the only other place where they are used is in the links. Hence, if we consider the distribution $(P_{\chi[i]}(1), r_{i,1}, \ldots, P_{\chi[i]}(\ell/2), r_{i,\ell/2}, \zeta)$ generated by running the protocol and letting ζ be the view of $\mathcal{B}$ with the links for wire i removed and if we let $r_{i,j}$ be the corresponding $\tilde{Y}^0_{i,j}$, then this distribution is hard, unless he can find Δ_j and thus compute the 0-key from the 1-key. However, we already proved that if one can find Δ_j with non-negligible probability then one can distinguish between the real/simulated world of the weak protocol. Hence we can replace the links $L_{i,j}=\mathsf{G}\left(P_{\chi[i]}(j), \tilde{Y}^0_{i,j}\right)$ by $\mathsf{G}\left(P_{\chi[i]}(j), r_{i,j}\right)$ for uniformly random $r_{i,j}$. This makes the hybrid layers $i-1$ and i computationally indistinguishable.

We notice that the weak protocol being considered on layer $\lceil 4.82s+4.82\rceil$, neither A nor the simulator needs to somehow become in possession of $\left(M^{\mathsf{Out}}\cdot \mathsf{ev}(\bar{f}, \bar{x_{\mathsf{A}}}\|\bar{x_{\mathsf{B}}})\right)\oplus\beta_b=\tilde{y}$. Thus we have not removed this unreal assumption.

We will now argue that since no polytime distinguisher can distinguish between the hybrids on layer $\lceil 4.82s+4.82\rceil$, then there cannot be a distinguisher that can distinguish between the real and simulated protocol. We do so in two steps: First assume we add the verifiable commitments of the order of A's input keys to the weak protocol/simulation. In this new protocol/simulation pair we have both the real and simulated execution consisting of the correct verifiable commitments, that is, in accordance with the protocol description.

Now assume, for the sake of contradiction, that an adversary can distinguish between this new real/simulation pair. If this is the case then it must be because of the verifiable commitments, since it is the only change and we have already showed that the real/simulation of the weak protocol are indistinguishable. Next see that the verifiable commitments are constructed similarly in both the real and simulated world. Furthermore, since the commitment functionality is ideal we notice that the only way the adversary can distinguish between the two settings is if he queries the commitment oracle on a pair of A's input keys (or on an output key of the computation of M^{In} that is not known). However, if he queries the functionality on a correct

[10] If a circuit output wire comes from a XOR gate then the test must be done on all of the non-XOR gates being part of the linear combination of input to such a XOR gate.

pair, say for input pair i in circuit j, then he will clearly know both the 0-key and the 1-key for this wire in this circuit (similar argument for the output keys).

However, then he can also compute the XOR of these two keys and learn Δ_j. Learning Δ_j will make it easy to distinguish between the hybrids except with negligible probability in κ (no matter if we are in the pair of hybrids of the weak protocol or the new pair) as proved in Lemma 15.

This means that everything we can do in the new pair of hybrids could be done in the previous pair, except with negligible chance in κ. Thus a distinguisher on the new pair of hybrids could efficiently be transfered to a distinguisher on the old pair. If the new distinguisher has non-negligible advantage in distinguishing the new pair, then the transformed distinguisher would also have that in the old pair. However, since we have already proved that no such distinguisher exists for the old pair then we have a contradiction.

Finally, the reason a PPT $\mathcal{B}$ cannot learn anything from the output of $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$, which will be different in the real and ideal world, is because the output is encrypted under a one-time pad, i.e., the random bits a, chosen by S which are XOR'ed with $M^{\mathsf{In}} \cdot x_{\mathsf{A}}$. So both in the real and ideal world this value will be uniformly distributed.

Hence, the above series of hybrids take us from the real execution to the simulation.

□

□

A note on the statistical security parameter. In the proof above we use that each of the bad events that a cheating A can try to cause can only happen with probability at most 2^{-s}. These events are:

1. A majority of inconsistent polynomials remains after cut-and-choose of polynomials.
    - This can only happen if she incorrectly constructs at least $\lceil 2.41 \cdot s + 3.41 \rceil$ of the polynomials.
2. *Every* check circuit is good and *every* evaluation circuit is bad.
    - This only happens if she constructs half of the circuits correctly and the other half incorrectly.
3. There is a collision of $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$.
    - This can only happen if she gives at least two different inputs to the protocol.[11]
4. A is successful in a selective failure attack.
    - This can only happen if she gives garbage for some inputs to the $\mathcal{F}_{\text{BOTCC}}$.
5. There is a collision of $\mathbb{H}^{\mathsf{Out}}$.
    - This can only happen if she constructs at least one circuit maliciously such that it computes something else than $f(x)$.

[11]There can seemingly be a collision if A maliciously constructs the verifiable commitments to the output keys of $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ for one of the circuits where she has given divergent input. However, this is not actually a collision of $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ but an incorrect circuit.

However, nothing is preventing A from trying to cause each of these events. Still, it should be noted that if A is unsuccessful in her cheating attempt of event 1, 2, or 3 then B will notice and can safely abort the protocol. Thus trying to cause more than just a single of the events 1, 2, or 3 will only increase the probability of A being caught and B terminating the protocol without the compromising security. Now notice that the list of bad events are in the order of which B will discover if they are unsuccessful (and in turn for the three first events, abort). Thus, to increase the probability of a successful cheating attempt A should try to make event 4 and 5 occur. This is so since if there is a collision of $\mathbb{H}^{\mathsf{Out}}$ B cannot terminate the protocol without possibly leaking some information on his private input and since a selective failure attack does not necessarily make B abort the protocol. She cannot combine an attack for event 4 or 5 with attack 1, 2 or 3 since if one of these are unsuccessful then B will terminate before event 4 and 5 could happen.

Now since event 4 and 5 can happen independently with probability 2^{-s} we get from the union bound that if A tries to make both occur she will be successful in having at least one of them occur with probability at most $2^{-s} + 2^{-s} = 2^{-s+1}$. This means that in order to achieve a total failure probability of at most 2^{-s} we must instead of s use $s' = s + 1$ in the part of the protocol concerned with $\mathbb{H}^{\mathsf{Out}}$ and the selective failure attack prevention. Thus M^{Sec} must be a $n_{\mathsf{B}} \times \max(4 \cdot n_{\mathsf{B}}, 8(s+1))$ matrix and $\mathbb{H}^{\mathsf{Out}}$ must have $\lceil 4.82(s+1) + 4.82 \rceil$ output wires (in turn we must also construct $6(s+1)+7$ polynomials to be certain that the success probability of guessing the cut-and-choose challenge of polynomials is still at most 2^{-s} unless we wish to reprove Lemma 13).

UC Security. We note the following about the simulator in the proof:

1. The simulator does not at any time rewind the adversary it simulates.

2. The simulator can extract the modified input of the malicious party in its simulation (and hence obtain the honest parties output from $\mathcal{F}_{\mathrm{SFE}}$).

3. The transcript of the simulated execution is generated in "real time". I.e. the simulator outputs values to the adversary at the right point during the execution and they are not changed later on.

Therefore, we immediately achieve UC security [Can01] if the underlying ideal functionalities used are UC-secure.

For example, assuming that the discrete log problem (DLP) is hard, the UC-secure oblivious transfer of [PVW08] can be used to realize the seed OTs.

Theorem 6 is stated in the hybrid model where the protocol depends on ideal functionalities for commitments, coin tossing, and oblivious transfers with consistent choice. In order to achieve a protocol that can be implemented in practice we apply the UC composition theorem, as stated in the following corollary.

Corrollary 2. *Let $\mathcal{G}$ be the GaXR garbling scheme of Fig. 2.8 where the encryption function is $\mathbb{E}^{\mathsf{H}}(K_o, K_l, K_r; T) = \mathsf{H}(K_l \| K_r \| T) \oplus K_o = C$ and decryption function $\mathbb{D}^{\mathsf{H}}(C, K_l, K_r; T) = \mathsf{H}(K_l \| K_r \| T) \oplus C$. Furthermore, let the links be implemented as discussed in Section 4.2. Then the protocol Π_{SFE} described in Section 4.3 UC-securely realizes the $\mathcal{F}_{\mathrm{SFE}}$ functionality from Fig. 4.1 in the $\mathcal{F}_{\mathrm{OT}}$-hybrid model against any PPT static and malicious adversary when the following is true: calls to $\mathcal{F}_{\mathrm{BOTCC}}$ are replaced by invocation of the protocol Π_{BOTCC} in Fig. 4.3, calls to $\mathcal{F}_{\mathrm{COM}}$ are replaced with invocations of the protocol Π_{COM} in Fig. 2.2 from Chapter 2,*

calls to $\mathcal{F}_{\mathrm{CT}}$ *are replaced by invocations of the protocol* Π_{CT} *in Fig. 2.3 from Chapter 2, and calls to* $\mathsf{H}(\cdot)$ *in the subprotocols above are done with calls to a programmable random oracle.*

Proof. We already showed in Section 2.4 that the scheme in Fig. 2.8 with encryption function $\mathbb{E}^{\mathsf{H}}(K_o, K_l, K_r; T) = \mathsf{H}(K_l \| K_r \| T) \oplus K_o = C$ and decryption function $\mathbb{D}^{\mathsf{H}}(C, K_l, K_r; T) = \mathsf{H}(K_l \| K_r \| T) \oplus C$ is a private, verifiable, key size preserving, structure free, free-XOR gate garbing scheme following Cor. 1 and Cor. 2 in the ROM. As proved in Section 2.3 the protocol Π_{COM} UC-securely realizes $\mathcal{F}_{\mathrm{COM}}$ in the programmable random oracle model. Next, consider that $\mathcal{F}_{\mathrm{BOTCC}}$ can be implemented by an invocation of Π_{BOTCC} along with a random oracle and a protocol for $\mathcal{F}_{\mathrm{OT}}$ as described in Section 4.2, where $\mathcal{F}_{\mathrm{OT}}$ is ideal. $\mathcal{F}_{\mathrm{CT}}$ can be realized using protocol Π_{CT} in the ROM. Finally, all of the above realizations are UC-secure in the presence of PPT static and malicious adversaries. Hence, by Theorem 6 and the UC-composition theorem (Theorem 1) we have that Cor. 2 follows immediately. □

Finally, we notice that the OT extension from Section 4.2 can be used for the OTs in the protocol in Fig. 4.3, again assuming an underlying OT functionality for the seed OTs.

4.5 Parallel Approach

In this section we briefly outline the structure of a CUDA GPU and discuss how to write code for it.

We assume access to a massively parallel computation device which is capable of executing the same instruction on each processor in parallel, but on different pieces of data (SIMD). This is in a sense the "minimal" way of modeling parallel computation, as a device capable of executing distinct instructions on distinct pieces of data (Multiple Instruction, Multiple Data (MIMD)) is clearly also capable of executing the same instruction on distinct pieces of data. Furthermore, our protocol does not make any assumption on whether such a device has access to shared memory between the processors, or only access to local memory. This applies completely for write privileges, but also for read privileges with only a constant memory usage penalty if shared memory is not available.

GPGPU. We decided to implement our protocol using a GPU, the motivation being that GPUs are part of practically all mid- to high-end consumer computers. Furthermore, using the GPU eliminates the security problems of outsourcing the computation to a non-local cluster. Also, assuming access to a local cluster seems to be an unrealistic assumption for many practical applications. Finally, using gaming consoles or multi-core CPUs might also have been an option. However, even the latest and best of these have orders of magnitude less cores than the latest GPUs.

CUDA. Our implementation is done using the CUDA platform which is an extension to C/C++ (or Fortran) that allows using NVIDIA GPUs for general computational tasks. This is done by making CUDA programs. Such a program does not purely run on the GPU. It consists of code which runs on the CPU, and CUDA classes which run on the GPU.

In order to motivate our specific implementation choices it is necessary to describe a general CUDA enabled GPU [KH10]: Each GPU consists of several *streaming multiprocessors* (SM), each of these again contains at least 8 *streaming processors* (SP), depending on the architecture of the GPU. Each of the SPs within a given SM always performs the same operations at a given point in time, but on different pieces of data. Furthermore, each of these SMs contains between

16 and 112 KB of *shared memory* along with at least 8 KB of constant cache, which all of the SPs within the given SM must share. For storage of variables each SM contains 8 to 128 K 32-bit registers which are shared amongst all the SPs within a given SM. Thus all the threads being executed by a given SM must share all these resources.

We now introduce some notation and concepts which are used in the general purpose GPU community and which we will also use for the rest of this thesis; a GPU is called a *device* and the non-GPU parts of a computer is called the *host*. This means that the CPU, RAM, hard drive etc., are part of the host. The code written for the host will be used to interact with the OS, that is, it will do all the IO operations needed by the CUDA program. The host code is also responsible for copying the data to and from the device, along with launching code on the device. Each procedure running on a device is called a *kernel*. Before launching a kernel the host code should complete all needed IO and copy all the data needed by the kernel to the device's RAM. The RAM of the device is referred to as *global memory*. After a kernel has terminated, the host can copy the results from the global memory of the device to its own memory, before it launches another kernel.

A kernel is more than just a procedure of code, it also contains specifications of how many times in parallel the code should be executed and any type of synchronization needed between the parallel executions. A kernel consists of code which is executed in a *grid*. A grid is a 2 (or 3) dimensional matrix of *blocks*. Each block is a 3-dimensional matrix of threads. Each thread is executed once and takes up one SP during its execution. When all the threads, of all the blocks in the grid, have been executed the kernel terminates. The threads in each block are executed in *warps*, which is a sequence of 32 threads. Thus, the threads must be partitioned into blocks in multiples of the warp size, and contain no branching.[12] The threads in a block can then be executed completely independent and in arbitrary order. The GPU scheduler tries to put as many blocks as possible onto the SMs at a given time. This makes it possible for fast context switching where the execution of a block can be interleaved if one block is waiting for data loaded from global memory, whereas an other block is ready to be executed. This means that we want as many blocks as possible resident on the SMs at the same time, known as *block occupancy*. However, even passive blocks must still use the storage of an SM (the registers and shared memory), so we must try to limit the storage needed for each block to achieve high block occupancy.

Reading global memory is done for 32 consecutive 32 bit words at a time, so to achieve the fastest execution time one should *coalescence* this data. That is, to ensure that the 32 words accessed at a given point in time by the threads in a warp are consecutive in memory. This makes it possible to load all of them in one go, and in turn significantly increase the speed of the program. This advice on memory organization is also relevant for the data in the shared memory. Finally, it is a well known fact [Cor15] that the bottleneck for most applications of the massive parallelism offered by CUDA is the memory bandwidth, thus it should always be a goal to limit the frequency of which a program accesses data in the global memory.

4.6 Implementation

As our protocols are designed to work well in the SIMD model we did our implementation in CUDA, which supports both explicit programming of SIMD execution, along with cheap hardware. More specifically we implemented our protocol in standard C using the CUDA extension by NVIDIA. As the second protocol is more or less the same as the first protocol, but involving

[12] If they do contain branching, then *all* possible branches are executed by *all* threads in the warp.

less garbled circuits and a more efficient version of the mechanism to ensure consistency of A's input along with a few bells and whistles, we will only describe the implementation details of this one. In particular it turns out that this implementation is more or less just a superset of the code used in the implementation of the first protocol.

Trading Assumptions for Efficiency

Since the description of the second protocol is given in the hybrid setting, assuming access to OTs, commitments, coin-tossing, and garbling, we need to make some decisions on the security assumptions needed in order to realize the functionalities in our implementation. We have chosen to be liberal in the security assumptions in order to achieve a more efficient solution. Specifically we implement a concrete scheme close to the one considered in Cor. 2. In particular this means that our implementation assumes the existence of a programmable random oracle and specifically we use SHA-1 to implement random oracle queries with a 160 bit output. Furthermore, we also base the OTs we need on the efficient batch OT extension, which only assumes access to $\Theta(s)$ "seed" OTs in the ROM described in Section 4.2. Finally, we also implement the verification of "circuit seeds" [GMS08] to eliminate the need of sending the garbled computation tables of the check circuits. In this approach A constructs each specific gabled circuit using the randomness of a small seed of entropy combined with a pseudo-random generator (in our case SHA-1). Then, instead of sending all the garbled circuits, she sends a hash digest of each of the garbled circuits, which is much smaller than the actual garbled circuit. If a circuit is selected as a check circuit A simply sends the seed for that circuit and B can construct the garbled circuit himself and verify them against the digest. For the evaluation circuits A sends the actual garbled circuit, which B verifies against the digests to make sure A did not try to cheat.

SIMD Implementation Optimizations

Gate Generation

Kernel structure. First, notice that we will have a case of SIMD for each of the ℓ circuits during construction. Thus, it is obvious to have each thread in a warp processing a distinct circuit and thus having the blocks be 1-dimensional, consisting of a constant amount of warps since this structure will give us high block occupancy. Now, since preliminary tests showed that a single warp in a block achieved greater efficiency than two or more warps in a block we chose to have blocks consist of 32 threads. A caveat with this is that if we wish to have ℓ not being a multiple of 32, we will need to allocate unused memory and cores and thus have SPs do useless work.

Next we notice that all gates within a single layer can be computed in arbitrary order, thus it is obvious to have one grid dimension be the amount of gates in each layer. Furthermore, as we cannot know which order the blocks will be computed in, we will need to have an iteration of kernel launches, one launch for each layer in the circuit, in order to have the output keys of the previous layer computed and ready for computing the next layer. We note that this is an effect of our specific choice of garbling scheme, since the gate output keys depend on the gate input keys. At the price of 33.3% increase in the size of a garbled circuit we could avoid this (by simply not using row reduction). In fact other authors have explored this approach [HMSG13]. Their scheme is up to a factor 3 faster than ours in computation on the same hardware. Their results were published *after* our first paper [FN13]. However, empirical evidence [KSS12] show that the bottleneck when using garbled circuits for SFE is the communication, and thus this approach does not seem viable in our setting.

Regarding memory management: We first copy the seeds onto the device, and then compute the global keys for all the circuits and the 0 keys for all the input wires in all the circuits, using a unique seed for each circuit. This is done by hashing the seed along with a unique ID in order to get a “random” key (remember we assume the ROM). Afterwards, using the generated keys, we initiate a loop of kernel launches in order to compute each layer of keys and garbled gates in each circuit. Between all these launches, all the currently computed keys, along with the global keys, remain in the global memory of the device so they can be used by the next kernels. Furthermore, we keep all the currently computed garbled gates on the device so that all the results can be copied to the host as a batch after all the kernels have finished. In order to save memory we only store the 0-key for each wire, since the 1-key can be efficient computed by simply XORing it with the appropriate global key for a given circuit.

Finally notice that the structure of the kernel for evaluation is the same as for garbling. The only difference is that before the initial launch the garbled gates for the whole circuit is copied from the host into the global memory along with the initial input keys, one key for each of the input wires, and a description of the circuit.

Memory coalescing. We memory coalesce all the data we use, both in the global memory and in the shared memory. As both keys and entries in a garbled gate consists of 160 bits (the digest size of SHA-1), i.e. five 32-bit words, we stored all data in *segments* of $32 \cdot 5 = 160$ words. The first entry is the first word of thread 1, the second entry is the first word of thread 2, and so on up to entry 33, which then contains the second word of thread 1, entry 34 contains the second word of thread 2 and so on. Thus, all data access is coalesced in a multiple of the warp size.

$\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ and $\mathbb{H}^{\mathsf{Out}}$

First notice that $\mathbb{H}^{\mathsf{Out}}$ is very similar to $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ and they can both be computed using only XOR operations on particular keys. Furthermore, each output key of both functions can be computed independently of the other output keys, since each output key only depends on the input keys to $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$, respectively $\mathbb{H}^{\mathsf{Out}}$, and the matrix M^{In}, respective M^{Out}. Thus the output of each function can clearly be computed in a SIMD manner, with threads up to the amount of outputs of $\mathbb{H}^{\mathsf{In}}$, respectively $\mathbb{H}^{\mathsf{Out}}$, i.e. s, respectively $\lceil 4.82s + 4.82 \rceil$.[13] However, the XOR operation is so light that it is in general considered overkill to have a GPU thread do nothing but a few XOR operations, because of the overhead of copying data from the RAM of the host system to the global memory and back. Thus we simply implemented the functionalities in a sequential manner using the CPU. However, using the CPU might not always be the best case as we will discuss, based on our experiments, in Section 4.6.

Commitments

For commitments to long strings ($\Omega(b)$ bits where b is the block size of the underlaying hash function modeled as a random oracle) we can use the fact that we are in the ROM and do recursive hashing: We do as in Fig. 2.2 and concatenate some random salt and an ID to the

[13] The computation of $\mathbb{H}^{\mathsf{In}}$ and $\mathbb{H}^{\mathsf{Out}}$ can be parallelized even further: Since XOR is commutative and associative we can use a classic reduction approach for computing each output key. That is, we can have several threads working on each output key by XOR'ing together a few of the input keys and storing the intermediate result. When these threads terminate new threads take over and XOR together the intermediate results, and so on, until a single thread remains which then contains the output key for a single output bit of the function. This makes it possible to parallelize up to the logarithm of the input to $\mathbb{H}^{\mathsf{In}}$, respectively $\mathbb{H}^{\mathsf{Out}}$.

string we need to commit to. It is now, say l bits long. We then simply hash all independent and continuous b bits pieces in parallel, resulting in $O(l/b)$ digests. We concatenate these digests and again hash all independent and continues b sized pieces of this string in parallel. We continue in this manner until a single digest remains which will be the commitment. The overhead of this approach is only logarithmic in the length of the string we hash.

Following this approach we can have each block in a grid compute the hash digest of a single blocksized piece of data. Now notice that in all the cases where we wish to compute a commitment to a long string we will already have the data in global memory in a coalesced manner. However, we also need to allocate a large result buffer in global memory that will be used to hold intermediate results. The recursive hashing procedure then consists of a loop of kernel calls where each iteration of the loop computes a level of the recursive hashes. The kernel itself is straight forward; shared memory is allocated for each segment of data to be hashed for each thread in a block (the block dimension will be 32 to match the way data is coalesced from the previous kernels). The data is then hashed and copied straight into the result buffer, still coalesced for a block of dimension 32. At the end of each iteration the result buffer and the data to be hashed switches pointers, so the old source data will be used as result buffer and the result buffer will act as the source. At the end the data in the current result buffer is copied back to the host memory.

The Modified OT Extension

Unlike the generation and evaluation of the garbled circuits, the modified OT extension involves many phases, several of which are depended on the previous phases and results from interacting with the other party. This means that we cannot have a single kernel, or even a single kernel function, in order to complete all the steps of the protocol for each party.

Like we did for the garbled circuits we coalesce all memory in blocks of 32 words. We also make segments, which consists of $5 \cdot 32 = 160$ words, such that each segment holds a coalesced hash value or a small 160 bit data array, for 32 threads. For this reason we again construct kernels to use blocks of 32 threads.

Using this choice, no coalescence conversion needs to be done to use the data from the modified OT extension with our implementation of garbled circuits. Furthermore, this choice will still keep an efficient and scalable organization of the memory. Also, as all the data we use for computations here is completely independent, we get the possibility of only launching a single kernel for each step of the protocol in order to avoid kernel launch overhead, resulting from the iterative launching of kernels.

The kernels needed in *Extension* and *Privacy Amplification* are almost the same so we only include a description of *Extension*.

Extension. The *Extension* step involves hashing $2 \cdot \left\lceil \frac{8}{3}\kappa \right\rceil$ seeds $\bar{n_{\mathsf{B}}}/\kappa$ times. In order to avoid redundant data copying of L_i^0 and L_i^1 to the device when we need to construct λ_i, we compute parts of all the three vectors, L_i^0, L_i^1, and λ_i in each thread. That is, we include the *Adjustment* step in the kernel. To save memory usage and bandwidth we let all the 32 threads of a single block use the same pair of seeds, thus we make each thread in a block compute 160 bits of each of the three vectors L_i^0, L_i^1, and λ_i for the same i. Next, one dimension of the grid is responsible for computing all $\bar{n_{\mathsf{B}}}$ bits of the three vectors, L_i^0, L_i^1, and λ_i, and thus contains $\left\lceil \frac{\bar{n_{\mathsf{B}}}}{32 \cdot \kappa} \right\rceil$ threads. The other dimension of the grid is responsible for doing this for each of the $\left\lceil \frac{8}{3}\kappa \right\rceil$ vectors that

needs to be computed. For the computation of the $L_i^{\Gamma[i]}$ values we proceed in the same manner, except each block only uses a single seed and each thread only computes a single digest.

Further improvements. For constructing and evaluating the garbled circuits the hash operations are clearly the main contributing computational factor. However, regarding the modified OT extension it turned out that computing the hash values on the device barely gave an improvement in the overall execution time, and that the main contributing time factor was that of transposing bits, i.e. the *Tilt-your-head* phase. In order to achieve a significant improvement in execution time we have to implement these steps efficiently in parallel. In order to do this we need to keep the overall hardware structure and memory hierarchy in mind.

First of all, we should notice that in order to construct one word of $\bar{L}_j$ or $\tilde{L}_j$ in the *Tilt-your-head* phase, we need a single bit from 32 different words in ${L'}_i^{\Gamma[i]}$ or L_i^0. In our memory organization, these are located in non-consecutive order. However, it should be noted that the remaining 31 bits of each of the 32 words are needed in the next 31 $\bar{L}$ bitstrings, $\bar{L}_{j+1}, \ldots, \bar{L}_{j+31}$. Thus, depending on the caches available, it makes sense to construct the first word of $\bar{L}_j, \bar{L}_{j+1}, \ldots, \bar{L}_{j+31}$ in a batch. That is, to load 32 words of ${L'}^{\Gamma[i]}$ or L_i^0 and use a single bit from each of these to construct the first word of $\bar{L}_j,\ \bar{L}_{j+1}, \ldots, \bar{L}_{j+31}$. For this approach to be successful we need a cache of $32 \cdot 32 = 1024$ words, or 4096 bytes in a 32 bit system. Fortunately, this is well within the amount of shared memory on a device.

Next, consider the subprotocol for parallel equality in the *Verification* phase. Notice that computing $\mathsf{H}(Z^A \| r)$ is basically the same as computing a commitment using the Π_{COM} protocol in Fig. 2.2. Thus we use the same approach as we just discussed for computing the commitments. That is, splitting the string $Z^A \| r$ into small pieces pieces larger than the digest size of H and recursively compute a hash digest. It is simple to implement on the device, by again having blocks of 32 threads and a grid of all the blocks needed to compute the individual digests. For each party we start by loading the input string into global memory and then construct a hash value of each, sufficiently large, chunk of bits, by loading the bits directly from global memory and storing the result back in global memory.

Finally, we also introduced slight multi-threading in the host code to eliminate some idle time where one of the parties might be doing, or waiting for, network communication but still is able to do computations on the data it already has. For A this includes sending the difference strings in the OT extension while computing commitments to her input keys. For B this includes receiving and verifying consistency of A's input keys while verifying the seeds of the check circuits.

Polynomial representation. In the protocol description in Section 4.3 we explained that the polynomials would have points in $\mathbb{F}_{2^{160}}$ as they need to have at least the same size as gate keys. However, our initial implementation showed, which was also expected, that doing polynomial interpolation on elements in $\mathbb{F}_{2^{160}}$ was quite slow. This was expected since multiplying two elements of $\mathbb{F}_{2^\kappa}$ requires $O\left(\kappa^2\right)$ operations using the “trivial” approach. The multiplication can of course be optimized, in particular when one of the numbers only has a few bits set. However, in our setting at least one of the numbers will always be random in $\mathbb{F}_{2^\kappa}$ and thus it does not seem possible to hope for better complexity than $\Omega(\kappa)$. However, when κ is small enough, multiplication and inversion in $\mathbb{F}_{2^\kappa}$ can be realized highly efficiently, with a good choice of reduction polynomial, in κ time, or in constant time through a lookup table. This is in particular true in the implementation of the S-box in AES for $\kappa = 8$. Now notice that in our setting we will need to use polynomials with random coefficients evaluated on points in the

range $[\ell]$. Since s is a statistical security parameter and the ciphertext size is 160[14] it will not make sense to have $s > 160$. Thus, in practice the range of points we will need to evaluate polynomials on will not be much bigger than 160.[15] Now see that $2^8 > 160$. So instead of using polynomials in $\mathbb{F}_{2^{160}}$ we use 20 polynomials in $\mathbb{F}_{2^8}$ in place of a single polynomial in $\mathbb{F}_{2^{160}}$ – i.e., we use polynomials over the field $\mathbb{F}_{2^8}^{20}$, which is still a secure secret sharing scheme, c.f. [CFIK03]. Thus we can use lookup tables to do multiplication and inversion in constant time, in a SIMD manner for each of the 20 polynomials.[16]

Polynomial construction. The $20 \cdot p$ polynomials with elements from $\mathbb{F}_{2^8}$ are generated in a SIMD manner using the GPU based on a 160 bit seed of entropy. Specifically what we do is to have $p \times (1 + \ell/2)$ threads generate coefficients consisting of 160 bits of pseudo-randomness by hashing a seed along with a unique identifier. Using these coefficients we have $p \times \ell$ threads computing ℓ points on each of the $20 \cdot p$ polynomials. More specifically the construction of $P_i(j) = (P_{i,k}(j))_{k \in [2]}$ for all $j \in [\ell]$ and $i \in [p]$. This is done by a loop of $1 + \ell/2$ iterations where an element $c_{i,j,k} \cdot j^d \in \mathbb{F}_{2^8}$ is computed in the d'th iteration and then added to the result of the previous $d - 1$ iterations. Here $c_{i,j,k}$ is a random coefficient in $\mathbb{F}_{2^8}$. The value j^d is computed by taking j^{d-1} from the previous round and multiplying it with j. This is done in order to avoid computing any exponentiations. Notice that the loop could also be parallelized in a SIMD manner by having a multiple of $1 + \ell/2$ more threads, each computing a single value of $c_{i,j,k} \cdot j^d$. Finally, these values could be added together using a reduction approach.

Polynomial interpolation. Our polynomial interpolation (part of the *Polynomial Setup*) is based on Lagrange interpolation and consists of 4 subphases, *Denominator, Numerator, Combination*, and *Reduction* in order to optimize the SIMD parallelization. Before we go through these phases remember that in Lagrange interpolation we assume knowledge of $1+\ell/2$ data point pairs $\{(x_j, y_j)\}_{j \in [1+\ell/2]}$ where all x_j are distinct. These define the interpolation polynomials in the Lagrange form:

$$L(x) := \sum_{j \in [1+\ell/2]} y_j \cdot l_j(x) \ ,$$

where the Lagrange basis polynomials are defined as

$$l_j(x) := \prod_{1 \le g \le 1+\ell/2, g \ne j} \frac{x - x_g}{x_j - x_g} \ ,$$

where $1 \le j \le 1 + \ell/2$. Using $l_j(\cdot)$, an arbitrary point x can then be computed. In our case we will know $1 + \ell/2$ points from $[\ell]$ and want to compute the remaining $\ell/2 - 1$ points in $[\ell]$.

When we verify that the ℓ points B got from A, for each of the $\lfloor 1.18 \cdot s + 2.18 \rfloor$ check polynomials, constitutes polynomials of degree at most $1 + \ell/2$, we do polynomial interpolation of the points $[2+\ell/2; \ell]$ using the points $[1; 1+\ell/2]$ and verify that the newly interpolated points are the same as A sent us. Now the phases for which we interpolate these points in a SIMD manner go as follows:

[14] The actual computational security is less because of the possibility of a birthday attack.

[15] Slightly larger because our replication factor is slightly bigger than the security parameter.

[16] In the actual implementation we do not use a look up table for multiplication as it turned out to be around two times slower than simply doing 8 iterations of bit fiddling. Furthermore, as an artifact of the implementation framework we do not do 20 polynomials in a SIMD manner, but instead sequentially.

Denominator: We have $1+\ell/2$ threads computing the denominator of each $l_j(x)$. Notice that this is enough as the choice of value of x does not come into play here, but only in the numerator. Finally, each thread computes the inverse so that each denominator can be directly multiplied onto a numerator later.

Numerator: We use $(1+\ell/2)\times(\ell/2-1)$ threads to compute the numerator of each $l_j(i)$. That is $l_j(i)$ will be computed by thread $j\cdot(\ell/2-1)+i$ so that we compute the numerator of each of the $1+\ell/2$ Lagrange polynomials using a different value for $i\in[2+\ell/2;\ell]$. Finally, each thread multiplies its result with the appropriate denominator from the previous step. That is, computing $l_j(i)$ for each $i\in[2+\ell/2;\ell]$.

Combination: We use $\lfloor 1.18\cdot s+2.18\rfloor\times(1+\ell/2)\times(\ell/2-1)$ threads to compute each of the $(1+\ell/2)$ terms of $L(i)$ for the $20\cdot\lfloor 1.18\cdot s+2.18\rfloor$ polynomials for each $i\in[2+\ell/2;\ell]$. This is simply done by having each thread multiply the appropriate value y_j for each of the $20\cdot\lfloor 1.18\cdot s+2.18\rfloor$ polynomials with the corresponding Lagrange basis polynomial (computed in the *Numerator* step).

Reduction: To continue with the maximal level of SIMD parallelization up to $\lfloor 1.18\cdot s+2.18\rfloor\times(1+\ell/2)\times(\ell/2-1)$ threads will complete a reduction approach, i.e. any given thread will XOR together two terms of $L(i)$. A new thread will take over two of the intermediate results and XOR these together. This will continue until $\lfloor 1.18\cdot s+2.18\rfloor\times(\ell/2-1)$ threads remain, containing the result $L(i)$ for all $i\in[2+\ell/2;\ell]$ and all $20\cdot\lfloor 1.18\cdot s+2.18\rfloor$ polynomials.

All the steps are done with all generated data staying in the global memory and only at the end the final results are copied back to the host's RAM. Because all the data is on the GPU it is sensible to do the reduction step instead of a sequential computation on the host system, even though the operations done will only be XORs.

Polynomial reconstruction. The polynomial reconstruction (part of the *Reconstruction* phase) is done almost in the same manner as the polynomial interpolation. The main difference being that in the reconstruction the set of $1+\ell/2$ points we know might be different for each of the $\lceil 4.82s+4.82\rceil$ sets of 20 polynomials. However, at most one of these points might be different since B will always learn the same $\ell/2$ points for all the polynomials as an effect of the cut-and-choose of garbled circuits. Thus we can implement the *Numerator* step in the following two substeps:

- First we use $(\ell/2)\times(\ell/2-1)$ threads to compute $\prod_{1\leq g\leq \ell/2, g\neq j}(x-x_g)$ for each point $j\in[\ell/2]$ where x_j is an index of a check circuit and for each of the $\ell/2-1$ points, x being an index of an evaluation circuit.
- We then use $\lceil 4.82s+4.82\rceil\times(\ell/2+1)\times(\ell/2-1)$ threads to multiply the last point, which might vary from polynomial to polynomial, onto the result of the previous step. This will give us the numerator in all the $1+\ell/2$ Lagrange basis polynomials for each of the $\ell/2-1$ points B needs to learn, i.e. the $\ell/2-1$ indices of the evaluation circuits.

The same idea applies to the computation of the *Denominator* step, but with a factor $\ell/2-1$ less, as the denominator remains the same for all possible values of x we need to learn.

With the above approach we avoid having $\lceil 4.82s+4.82\rceil\times(1+\ell/2)\times(\ell/2-1)$ threads doing a multiplication loop of $1+\ell/2$ iterations and instead only use $\ell/2\times(\ell/2-1)$ threads

with a loop of $\ell/2$ iterations and $\lceil 4.82s + 4.82 \rceil \times (1 + \ell/2) \times (\ell/2 - 1)$ threads with a constant amount of multiplications to achieve the same result.

A Note on Multi-Threading

In order to limit the time each party is idle we employ multi-threading to the protocol in a non-SIMD manner. That is, when several distinct steps of the protocol can be carried out independently of each other we fork the host process to carry out these computations in parallel. In particular we fork when some data needs to be sent/received while other operations can be carried out with the data already known.

A Note on Parallel Complexity

The parallel complexity of cut-and-choose of garbled circuits is bounded by the depth of the circuit to compute, times the complexity of garbling a single gate, along with the parallel complexity of the handling of selective failure attacks, consistency of inputs, OTs, and commitments. In our case the complexity of handling selective failure attacks is bounded by $O(\log(\max(n_{\mathsf{B}}, s)))$ as it only consists of XOR'ing (because of the free-XOR approach) keys of κ bits with each other, $O(s^2 \cdot \max(n_{\mathsf{B}}, s))$ times in parallel, using a reduction approach. The same argumentation goes for ensuring consistency of A's input, i.e. computing $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$: Using a reduction approach it is basically $O(\log(n_{\mathsf{A}}))$ SIMD XOR operations of κ bits $O(s^2 \cdot n_{\mathsf{A}})$ times in parallel. Regarding handling input recovery in case of cheating, first notice that the parallel computation of $\mathbb{H}^{\mathsf{Out}}$ is almost the same as for $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$, thus it requires $O(\log(m))$ SIMD XOR operations of κ bits using $O(s^2 \cdot m)$ SIMD processors.

Considering the polynomial generation, notice that we first generate coefficients and then evaluate points based on the coefficients. Now, evaluating points can be done using a reduction approach, that is, each term of a polynomial is evaluated independently, then two terms are added together by one thread, these results are then added together two at a time and so on, until all the terms for a given point in a given polynomial have been added together. This implies that the complexity of polynomial generation is bounded by the logarithm of the degree of the polynomial, that is $O(\log(s))$ SIMD operations using $O(s^3)$ SIMD processors operating on κ bit values. Regarding polynomial interpolation, assuming we use Lagrange interpolation, the straight forward approach would be to make an evaluation of a Lagrange basis polynomial for each point we wish to interpolate. These basis polynomials will consist of $1 + s/2$ factors. Again we can compute each factor independently and combine the factors using a reduction approach. These basis polynomials will be the same for each of the polynomials we wish to interpolate, with the exception of at most one factor (depending on which augmented output wires contains 0-values for each evaluation circuits). These basis polynomials are then used to find the actual points by multiplying each of them with a known point (learned from the links) and then adding the terms together. There will be $1 + s/2$ terms and again these can be added together using a reduction approach. Thus, the interpolation complexity is also limited to $O(\log(s))$ SIMD operations using $O(s^3)$ SIMD processors operating on κ bit values.

Notice that in the above analysis we assume that addition and multiplication is constant time, which makes sense since they can be implemented in a loop with 8 iterations or through a lookup table following the implementation idea in Section 4.6.

s		IO	Comp.	Comm.	Idle	Party total	Total	Total idle
9	A	4.073 ± 0.036	53.19 ± 0.47	45.77 ± 0.17	81.70 ± 3.5	184.7 ± 3.6	211.0 ± 3.6	160.1 ± 7.0
	B	4.040 ± 0.0012	82.81 ± 0.60	45.77 ± 0.17	78.38 ± 3.5	211.0 ± 3.6		
19	A	4.053 ± 0.020	65.81 ± 0.89	71.97 ± 0.20	86.75 ± 4.5	228.6 ± 4.3	260.1 ± 4.3	171.3 ± 8.7
	B	4.039 ± 0.00097	99.49 ± 0.71	71.97 ± 0.20	84.56 ± 4.3	260.1 ± 4.3		
30	A	4.046 ± 0.0080	119.9 ± 0.90	108.1 ± 0.26	112.3 ± 3.8	344.4 ± 3.9	398.1 ± 4.1	231.0 ± 7.6
	B	4.066 ± 0.031	167.3 ± 0.70	108.1 ± 0.26	118.7 ± 3.8	398.1 ± 4.1		
40	A	4.055 ± 0.017	137.3 ± 0.79	132.3 ± 0.30	118.5 ± 4.1	392.1 ± 4.2	455.7 ± 4.2	244.7 ± 8.1
	B	4.049 ± 0.016	193.1 ± 1.0	132.3 ± 0.30	126.2 ± 4.1	455.6 ± 4.2		
60	A	4.043 ± 0.0069	170.1 ± 1.9	178.1 ± 0.38	130.3 ± 4.5	482.6 ± 4.3	583.0 ± 4.3	263.7 ± 8.5
	B	4.093 ± 0.088	266.2 ± 0.77	178.1 ± 0.38	133.4 ± 4.3	581.8 ± 4.4		
80	A	4.055 ± 0.017	247.1 ± 1.6	231.2 ± 0.37	168.1 ± 4.1	650.5 ± 4.2	810.4 ± 3.5	340.2 ± 6.9
	B	4.069 ± 0.038	398.8 ± 1.1	231.2 ± 0.37	172.0 ± 3.0	806.1 ± 3.5		
119	A	4.055 ± 0.020	377.9 ± 2.9	343.8 ± 0.37	215.1 ± 6.9	940.8 ± 4.2	1220 ± 5.2	431.0 ± 12
	B	4.040 ± 0.0026	641.5 ± 1.1	343.8 ± 0.37	215.9 ± 5.3	1205 ± 5.2		

Table 4.1: Timing in milliseconds of the expected and 95% confidence interval of the execution of the entire second protocol computing oblivious AES-128 for both A and B under different statistical security parameters (s). Column "Comp." represents wall-clock time where at least one thread does computation. Columns "IO", "Comm.", and "Idle " represent wall-clock time where the protocol execution is single-threaded and does disk loading, network communication, or is completely idle respectively. Column "Party total" represents the total wall-clock time of protocol execution for each party. Column "Total" represents the total wall-clock execution time of the entire protocol. Column "Total idle" represents the total wall-clock time when one of the parties is idle.

Experimental Results

All of our experiments are based on the same, commonly used, circuit for oblivious 128 bit AES encryption [PSSW09].[17] This circuit is used as benchmark in [HEKM11, NNOB12, HKS+10], and more implementations of 2PC for functions expressed as a Boolean circuit. We ran experiments on the circuit with several different statistical security parameters on two consumer grade desktop computers connected to Aarhus University's gigabit local area network. At the time of purchase (2012) each of these machines had a price of less than $1600. Both machines have similar specifications: an Intel Ivy Bridge i7 3.5 GHz quad-core processor, 8 GB DDR3 RAM, an Intel series-520 180 GB SSD drive, a MSI Z77 motherboard with gigabit LAN, and a MSI GPU with an NVIDIA GTX 670 chip and 2 GB GDDR5 RAM. The machines ran up-to-date versions (at the time of testing) of Linux Mint 14 and CUDA 5.5. Each of the experiments were repeated 30 times for the first protocol and 50 times for the second, with no front-end applications running on either of the machines. The timings of our second protocol are summarized in Table 4.1 and Table 4.2. These timings include loading circuit description and randomness along with communication between the host and device and communication between the parties.[18] However, in the same manner as done in [NNOB12] the timings of the seed OTs have

[17] We thank Benny Pinkas, Thomas Schneider, Nigel P. Smart and Stephen C. Williams for supplying the circuit.

[18] In the tests the entropy used was sampled from `/dev/urandom`, which is a non-blocking source of pseudo-randomness. This source was used in order to avoid the probable high variance in the execution time caused by

not been included as it is a computation that is needed once between two parties and thus will get amortized out in a practical context. The time it takes to initialize the GPU device (driver related overhead) has not counted either, and generally would constitute between 50 and 60 milliseconds on our test systems when the GPU is set to "persistence mode".

The timings are in milliseconds and represent "wall-clock" times. However, since some aspects of the execution are multi-threaded we have chosen to count those parts as computation time, even though one thread might not be doing computation, but rather be idle or doing communication. Thus what is counted in the communication, respectively idle columns, is the time where the party is only doing communication, respectively is completely idle.

For ease of presentation we have only included the detailed timings of our second protocol.

s		Comp.	Comm.	Idle	Party total
9	A	4.836 ± 0.17	1.592 ± 0.046	< 1	6.467 ± 0.17
	B	14.04 ± 0.43	1.592 ± 0.046	4.808 ± 2.4	20.44 ± 2.5
19	A	7.373 ± 0.93	3.885 ± 0.066	< 1	11.28 ± 0.94
	B	20.10 ± 0.52	3.885 ± 0.066	10.49 ± 3.9	34.47 ± 3.9
30	A	33.24 ± 1.4	8.244 ± 0.079	< 1	41.86 ± 1.4
	B	34.56 ± 0.78	8.244 ± 0.079	12.53 ± 2.5	55.34 ± 2.7
40	A	42.60 ± 0.46	12.64 ± 0.079	< 1	55.34 ± 0.44
	B	51.63 ± 1.1	12.64 ± 0.079	18.40 ± 2.9	82.67 ± 3.1
60	A	47.35 ± 0.62	24.38 ± 0.11	< 1	71.94 ± 0.58
	B	116.1 ± 0.83	24.38 ± 0.11	23.67 ± 2.5	164.2 ± 2.7
80	A	66.48 ± 0.40	40.91 ± 0.089	< 1	107.4 ± 0.37
	B	185.4 ± 2.7	40.91 ± 0.089	39.13 ± 0.28	265.4 ± 2.7
119	A	102.4 ± 0.59	84.71 ± 0.11	< 1	187.6 ± 0.51
	B	315.1 ± 1.8	84.71 ± 0.11	71.28 ± 0.22	471.1 ± 1.7

Table 4.2: Timing in milliseconds of the expected and 95% confidence interval of the execution of the parts of the second protocol responsible for allowing forge-and-lose, when computing oblivious AES-128 for both A and B under different statistical security parameters (s). Column "Comp." represents wall-clock time where at least one thread of a given party does computation. Columns "IO", "Comm.", and "Idle " represent wall-clock time where the protocol execution is single-threaded and does disk loading, network communication, or is completely idle respectively. Column "Party total" represents the total wall-clock time of protocol execution.

Data analysis. From Table 4.1 we see that idle time takes up a significant portion of the total execution time of the second protocol, i.e. between 18% and 44% for each party and up to 76% of the wall-clock time one party sits idle. Taking Table 4.2 into account, we see that at most 30% of the overhead of recovery is idle time. The reason is probably that most messages that needs to be sent as part of our approach to input recovery, can be batched together with messages that needs to be sent as part of the generic structure of cut-and-choose of garbled circuits.

waiting for "true" randomness from `/dev/random`, which is a blocking source of randomness.

	Security	s	RO	Rounds	Time (s)	Equipment
[HEKM11]	Semi-honest	-	Yes	$O(1)$	0.2	Desktop
[FJN14]	Malicious	9	Yes	$O(1)$	0.21	Desktop w. GPU
[FN13]	Malicious	9	Yes	$O(1)$	0.29	Desktop w. GPU
[FJN14]	Malicious	40	Yes	$O(1)$	0.46	Desktop w. GPU
[FN13]	Malicious	39	Yes	$O(1)$	1.1	Desktop w. GPU
[FJN14]	Malicious	60	Yes	$O(1)$	0.58	Desktop w. GPU
[FJN14]	Malicious	80	Yes	$O(1)$	0.81	Desktop w. GPU
[NNOB12]	Malicious	58	Yes	$O(d)$	1.6	Desktop
[KSS12]	Malicious	80	No	$O(1)$	1.4	Cluster, 256 nodes
[FN13]	Malicious	79	Yes	$O(1)$	2.6	Desktop w. GPU
[AMPR14]	Malicious	40	No	1	6.4	Desktop
[SS13]	Malicious	80	No	$O(1)$	40.6	Cluster, 8 nodes

Table 4.3: Timing comparison of the secure two party SFE of oblivious 128-bit AES. d is the depth of the circuit to be computed and κ is at least 128. We note that [FN13] is our first protocol and [FJN14] is our second protocol.

It should be noted that we have spent quite a bit of time trying to limit the idle time of the implementation by using multi-threading, batching messages and restructuring steps within a given party's execution. Unfortunately, it remains unknown how much this has limited idle time compared to other implementations of cut-and-choose of garbled circuits, as other authors with comparable protocols have not included measurements of idle time.

Comparing the total times of Table 4.1 and Table 4.2 we see that input recovery constitutes from 10% up to 39% for of the execution time of B, whereas for A it goes from 3.5% up to 20%. Looking further into the different steps of input recovery it turns out that around half of the computation time B spends on *Reconstruction* is actually spent doing the "free" computation of the output keys of $\mathbb{H}^{\mathsf{Out}}$ (since it is all XOR operations on keys and we use free-XOR). He spends even more time on "free" computations before we optimized the *Reconstruction* part of our implementation to limit the amount of cache-misses.

A comparison of our protocols with the competition, which also works with garbled circuits, can be found in Table 4.3.

	s	Time (ms)
[DLT14]	128	4
[NNOB12]	83	32
[KSS13]	40	1

Table 4.4: Online time comparison of maliciously secure computation protocols evaluating oblivious 128-bit AES for two parties assuming a trusted preprocessing phase. Timing is in milliseconds and all timings are the best ones reported in the given papers assuming *large* amortization. The round complexity of all implementations is $O(d)$ where d is the depth of the circuit to be computed. All protocols are benchmarked on desktop computers.

Other Implementations of Secure Computation

We note that if one only cares about online time, i.e. that we are allowed to spend a great amount of time constructing correlated randomness before knowing which function and input we wish to do computation on, then significantly more efficient solutions exist. Protocols implementing this approach are said to be in the *preprocessing model*. We briefly discussed some of these in Chapter 1.

For completeness we include a table of only the *online* time of some of the most efficient online implementations of these protocols in Table 4.4. Again we consider the setting of two parties computing oblivious AES 128.

Bibliography

[AIK14] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. How to garble arithmetic circuits. *SIAM J. Comput.*, 43(2):905–929, 2014.

[AIKW15] Benny Applebaum, Yuval Ishai, Eyal Kushilevitz, and Brent Waters. Encoding functions with constant online rate, or how to compress garbled circuit keys. *SIAM J. Comput.*, 44(2):433–466, 2015.

[ALSZ13] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In Sadeghi et al. [SGY13], pages 535–548.

[ALSZ15] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer extensions with security for malicious adversaries. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 673–701. Springer, 2015.

[AMPR14] Arash Afshar, Payman Mohassel, Benny Pinkas, and Ben Riva. Non-interactive secure computation based on cut-and-choose. In Nguyen and Oswald [NO14], pages 387–404.

[AP14] Michel Abdalla and Roberto De Prisco, editors. *Security and Cryptography for Networks - 9th International Conference, SCN 2014, Amalfi, Italy, September 3-5, 2014. Proceedings*, volume 8642 of *Lecture Notes in Computer Science*. Springer, 2014.

[App13] Benny Applebaum. Garbling XOR gates "for free" in the standard model. In Amit Sahai, editor, *Theory of Cryptography*, volume 7785 of *Lecture Notes in Computer Science*, pages 162–181. Springer Berlin Heidelberg, 2013.

[BCD+09] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *Financial Cryptography and Data Security, 13th International Conference, FC 2009, Accra Beach, Barbados, February 23-26, 2009. Revised Selected Papers*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2009.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Paterson [Pat11], pages 169–188.

[Bea96] Donald Beaver. Correlated pseudorandomness and the complexity of private computations. In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 479–488. ACM, 1996.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10. ACM, 1988.

[BHHI10] Boaz Barak, Iftach Haitner, Dennis Hofheinz, and Yuval Ishai. Bounded key-dependent message security. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *Lecture Notes in Computer Science*, pages 423–444. Springer, 2010.

[BHKR13] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 478–492. IEEE Computer Society, 2013.

[BHR12a] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, volume 7658 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2012.

[BHR12b] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 784–796. ACM, 2012.

[Bih03] Eli Biham, editor. *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4-8, 2003, Proceedings*, volume 2656 of *Lecture Notes in Computer Science*. Springer, 2003.

[BLN+15] Sai Sheshank Burra, Enrique Larraia, Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, Emmanuela Orsini, Peter Scholl, and Nigel P. Smart. High performance multi-party computation for binary circuits based on oblivious transfer. Cryptology ePrint Archive, Report 2015/472, 2015. `http://eprint.iacr.org/`.

[Blu81] Manuel Blum. Coin flipping by telephone. In Allen Gersho, editor, *Advances in Cryptology: A Report on CRYPTO 81, CRYPTO 81, IEEE Workshop on Communications Security, Santa Barbara, California, USA, August 24-26, 1981.*, pages 11–15. U. C. Santa Barbara, Dept. of Elec. and Computer Eng., ECE Report No 82-04, 1981.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In Harriet Ortiz, editor, *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 503–513. ACM, 1990.

[BP12] Joan Boyar and René Peralta. A small depth-16 circuit for the AES s-box. In Dimitris Gritzalis, Steven Furnell, and Marianthi Theoharidou, editors, *Information Security and Privacy Research - 27th IFIP TC 11 Information Security and Privacy Conference, SEC 2012, Heraklion, Crete, Greece, June 4-6, 2012. Proceedings*, volume 376 of *IFIP Advances in Information and Communication Technology*, pages 287–298. Springer, 2012.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993.*, pages 62–73. ACM, 1993.

[Bra13] Luís T. A. N. Brandão. Secure two-party computation with reusable bit-commitments, via a cut-and-choose with forge-and-lose technique - (extended abstract). In Kazue Sako and Palash Sarkar, editors, *Advances in Cryptology - ASIACRYPT 2013 - 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part II*, volume 8270 of *Lecture Notes in Computer Science*, pages 441–463. Springer, 2013.

[Can00] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. `http://eprint.iacr.org/`.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 136–145. IEEE Computer Society, 2001.

[CDD+15] Ignacio Cascudo, Ivan Damgård, Bernardo Machado David, Irene Giacomelli, Jesper Buus Nielsen, and Roberto Trifiletti. Additively homomorphic UC commitments with optimal amortized overhead. In Jonathan Katz, editor, *Public-Key Cryptography - PKC 2015 - 18th IACR International Conference on Practice and Theory in Public-Key Cryptography, Gaithersburg, MD, USA, March 30 - April 1, 2015, Proceedings*, volume 9020 of *Lecture Notes in Computer Science*, pages 495–515. Springer, 2015.

[CFIK03] Ronald Cramer, Serge Fehr, Yuval Ishai, and Eyal Kushilevitz. Efficient multi-party computation over rings. In Biham [Bih03], pages 596–613.

[CG13] Ran Canetti and Juan A. Garay, editors. *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, volume 8043 of *Lecture Notes in Computer Science*. Springer, 2013.

[CGH04] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited. *J. ACM*, 51(4):557–594, 2004.

[CKKC13] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Carlos Cid. Multi-client non-interactive verifiable computation. In *Proceedings of the 10th Theory of Cryptography Conference on Theory of Cryptography*, TCC'13, pages 499–518, Berlin, Heidelberg, 2013. Springer.

[CKKZ12] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. On the security of the "free-XOR" technique. In Ronald Cramer, editor, *Theory of Cryptography - 9th Theory of Cryptography Conference, TCC 2012, Taormina, Sicily, Italy, March 19-21, 2012. Proceedings*, volume 7194 of *Lecture Notes in Computer Science*, pages 39–53. Springer, 2012.

[CKMZ14] Seung Geol Choi, Jonathan Katz, Alex J. Malozemoff, and Vassilis Zikas. Efficient three-party computation from cut-and-choose. In Garay and Gennaro [GG14], pages 513–530.

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In John H. Reif, editor, *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*, pages 494–503. ACM, 2002.

[Cor15] Nvidia Corporation. NVIDIA CUDA C Programming Best Practices Guide. Technical report, September 2015.

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.

[DLT14] Ivan Damgård, Rasmus Lauritsen, and Tomas Toft. An empirical study and some improvements of the MiniMac protocol for secure computation. In Abdalla and Prisco [AP14], pages 398–415.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Safavi-Naini and Canetti [SC12], pages 643–662.

[DZ13] Ivan Damgård and Sarah Zakarias. Constant-overhead secure computation of boolean circuits using preprocessing. In Amit Sahai, editor, *Theory of Cryptography*, volume 7785 of *Lecture Notes in Computer Science*, pages 621–641. Springer, 2013.

[EGL85] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6):637–647, 1985.

[Fin14] Magnus Gausdal Find. On the complexity of computing two nonlinearity measures. In Edward A. Hirsch, Sergei O. Kuznetsov, Jean-Éric Pin, and Nikolay K.

Vereshchagin, editors, *Computer Science - Theory and Applications - 9th International Computer Science Symposium in Russia, CSR 2014, Moscow, Russia, June 7-11, 2014. Proceedings*, volume 8476 of *Lecture Notes in Computer Science*, pages 167–175. Springer, 2014.

[FJN+13] Tore Kasper Frederiksen, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Peter Sebastian Nordholt, and Claudio Orlandi. MiniLEGO: Efficient secure two-party computation from general assumptions. In Johansson and Nguyen [JN13], pages 537–556.

[FJN14] Tore Kasper Frederiksen, Thomas P. Jakobsen, and Jesper Buus Nielsen. Faster maliciously secure two-party computation using the GPU. In Abdalla and Prisco [AP14], pages 358–379.

[FJNT15a] Tore Kasper Frederiksen, Thomas P. Jakobsen, Jesper Buus Nielsen, and Roberto Trifiletti. On the complexity of additively homomorphic UC commitments. In Submission and Cryptology ePrint Archive, Report 2015/694, 2015. `http://eprint.iacr.org/`.

[FJNT15b] Tore Kasper Frederiksen, Thomas P. Jakobsen, Jesper Buus Nielsen, and Roberto Trifiletti. TinyLEGO: An interactive garbling scheme for maliciously secure two-party computation. Cryptology ePrint Archive, Report 2015/309, 2015. `http://eprint.iacr.org/`.

[FKOS15] Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. A unified approach to MPC with preprocessing using OT. Under submission, 2015.

[FN13] Tore Kasper Frederiksen and Jesper Buus Nielsen. Fast and maliciously secure two-party computation using the GPU. In Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings*, volume 7954 of *Lecture Notes in Computer Science*, pages 339–356. Springer, 2013.

[FNO15] Tore Kasper Frederiksen, Jesper Buus Nielsen, and Claudio Orlandi. Privacy-free garbled circuits with applications to efficient zero-knowledge. In Oswald and Fischlin [OF15], pages 191–219.

[GG14] Juan A. Garay and Rosario Gennaro, editors. *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, volume 8617 of *Lecture Notes in Computer Science*. Springer, 2014.

[GGH+13] Craig Gentry, Sergey Gorbunov, Shai Halevi, Vinod Vaikuntanathan, and Dhinakaran Vinayagamurthy. How to compress (reusable) garbled circuits. Cryptology ePrint Archive, Report 2013/687, 2013. `http://eprint.iacr.org/`.

[GGP10] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In Tal Rabin, editor, *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, volume 6223 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2010.

[GHL+14] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In Nguyen and Oswald [NO14], pages 405–422.

[GKP+13] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 555–564. ACM, 2013.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. One-time programs. In Wagner [Wag08], pages 39–56.

[GMS08] Vipul Goyal, Payman Mohassel, and Adam Smith. Efficient two party and multi party computation against covert adversaries. In Nigel P. Smart, editor, *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, volume 4965 of *Lecture Notes in Computer Science*, pages 289–306. Springer, 2008.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229. ACM, 1987.

[Gol03] Oded Goldreich. Cryptography and cryptographic protocols. *Distributed Computing*, 16(2-3):177–199, 2003.

[Gol04] Oded Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.

[HEKM11] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011.

[HKE12] Yan Huang, Jonathan Katz, and David Evans. Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 272–284. IEEE Computer Society, 2012.

[HKE13] Yan Huang, Jonathan Katz, and David Evans. Efficient secure two-party computation using symmetric cut-and-choose. In Canetti and Garay [CG13], pages 18–35.

[HKK+14] Yan Huang, Jonathan Katz, Vladimir Kolesnikov, Ranjit Kumaresan, and Alex J. Malozemoff. Amortizing garbled circuits. In Garay and Gennaro [GG14], pages 458–475.

[HKS+10] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: tool for automating secure two-party computations. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 451–462. ACM, 2010.

[HMSG13] Nathaniel Husted, Steven Myers, Abhi Shelat, and Paul Grubbs. GPU and CPU parallelization of honest-but-curious secure two-party computation. In Charles N. Payne Jr., editor, *Annual Computer Security Applications Conference, ACSAC '13, New Orleans, LA, USA, December 9-13, 2013*, pages 169–178. ACM, 2013.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2003.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*, pages 21–30. ACM, 2007.

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In Wagner [Wag08], pages 572–591.

[IW14] Yuval Ishai and Hoeteck Wee. Partial garbling schemes and their applications. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*, volume 8572 of *Lecture Notes in Computer Science*, pages 650–662. Springer, 2014.

[JKO13] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In Sadeghi et al. [SGY13], pages 955–966.

[JKSS10] Kimmo Järvinen, Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Garbled circuits for leakage-resilience: Hardware implementation and evaluation of one-time programs - (full version). In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 383–397. Springer, 2010.

[JN13] Thomas Johansson and Phong Q. Nguyen, editors. *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*. Springer, 2013.

[KH10] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[KM11] Jonathan Katz and Lior Malka. Constant-round private function evaluation with linear complexity. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, volume 7073 of *Lecture Notes in Computer Science*, pages 556–571. Springer, 2011.

[KMR14] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. FleXOR: Flexible garbling for XOR gates that beats free-XOR. In Garay and Gennaro [GG14], pages 440–457.

[Kol05] Vladimir Kolesnikov. Gate evaluation secret sharing and secure one-round two-party computation. In Bimal K. Roy, editor, *Advances in Cryptology - ASIACRYPT 2005, 11th International Conference on the Theory and Application of Cryptology and Information Security, Chennai, India, December 4-8, 2005, Proceedings*, volume 3788 of *Lecture Notes in Computer Science*, pages 136–155. Springer, 2005.

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 724–741. Springer, 2015.

[KS06] Mehmet S. Kiraz and Berry Schoenmakers. A protocol issue for the malicious case of Yao's garbled circuit construction. In *In Proceedings of 27th Symposium on Information Theory in the Benelux*, pages 283–290, 2006.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498. Springer, 2008.

[KSS12] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. Billion-gate secure computation with malicious adversaries. In Tadayoshi Kohno, editor, *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 285–300. USENIX Association, 2012.

[KSS13] Marcel Keller, Peter Scholl, and Nigel P. Smart. An architecture for practical actively secure MPC with dishonest majority. In Sadeghi et al. [SGY13], pages 549–560.

[KW13] Seny Kamara and Lei Wei. Garbled circuits via structured encryption. In Andrew A. Adams, Michael Brenner, and Matthew Smith, editors, *Financial Cryptography and Data Security - FC 2013 Workshops, USEC and WAHC 2013, Okinawa, Japan, April 1, 2013, Revised Selected Papers*, volume 7862 of *Lecture Notes in Computer Science*, pages 177–188. Springer, 2013.

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, volume 6477 of *Lecture Notes in Computer Science*, pages 177–194. Springer, 2010.

[Lin13] Yehuda Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In Canetti and Garay [CG13], pages 1–17.

[LO13] Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In Johansson and Nguyen [JN13], pages 719–734.

[LOS14] Enrique Larraia, Emmanuela Orsini, and Nigel P. Smart. Dishonest majority multiparty computation for binary circuits. In Garay and Gennaro [GG14], pages 495–512.

[LP09] Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *J. Cryptology*, 22(2):161–188, 2009.

[LP12] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. *J. Cryptology*, 25(4):680–722, 2012.

[LP15] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. *J. Cryptology*, 28(2):312–350, 2015.

[LPSY15] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. Efficient constant round multi-party computation combining BMR and SPDZ. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 319–338. Springer, 2015.

[LR14] Yehuda Lindell and Ben Riva. Cut-and-choose Yao-based secure computation in the online/offline and batch settings. In Garay and Gennaro [GG14], pages 476–494.

[MF06] Payman Mohassel and Matthew K. Franklin. Efficiency tradeoffs for malicious two-party computation. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*, volume 3958 of *Lecture Notes in Computer Science*, pages 458–473. Springer, 2006.

[MNT93] Yishay Mansour, Noam Nisan, and Prasoon Tiwari. The computational complexity of universal hashing. *Theor. Comput. Sci.*, 107(1):121–133, 1993.

[MR13] Payman Mohassel and Ben Riva. Garbled circuits checking garbled circuits: More efficient and secure two-party computation. In Canetti and Garay [CG13], pages 36–53.

[MS13] Payman Mohassel and Seyed Saeed Sadeghian. How to hide circuits in MPC an efficient framework for private function evaluation. In Johansson and Nguyen [JN13], pages 557–574.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Safavi-Naini and Canetti [SC12], pages 681–700.

[NO09] Jesper Buus Nielsen and Claudio Orlandi. LEGO for two-party secure computation. In Omer Reingold, editor, *Theory of Cryptography, 6th Theory of Cryptography Conference, TCC 2009, San Francisco, CA, USA, March 15-17, 2009. Proceedings*, volume 5444 of *Lecture Notes in Computer Science*, pages 368–386. Springer, 2009.

[NO14] Phong Q. Nguyen and Elisabeth Oswald, editors. *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, volume 8441 of *Lecture Notes in Computer Science*. Springer, 2014.

[NPS99] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM Conference on Electronic Commerce*, EC '99, pages 129–139, New York, NY, USA, 1999. ACM.

[OF15] Elisabeth Oswald and Marc Fischlin, editors. *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, volume 9057 of *Lecture Notes in Computer Science*. Springer, 2015.

[Pat11] Kenneth G. Paterson, editor. *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, volume 6632 of *Lecture Notes in Computer Science*. Springer, 2011.

[Pin03] Benny Pinkas. Fair secure two-party computation. In Biham [Bih03], pages 87–105.

[PSSW09] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, volume 5912 of *Lecture Notes in Computer Science*, pages 250–267. Springer, 2009.

[PSSZ15] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 515–530. USENIX Association, 2015.

[PVW08] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In Wagner [Wag08], pages 554–571.

[Rab81] Michael O. Rabin. How to exchange secrets with oblivious transfer. Technical Report TR-81, Aiken Computation Lab, Harvard University, 1981.

[Rog91] Phillip Rogaway. *The Round Complexity of Secure Protocols*. PhD thesis, Massachusetts Institute of Technology, 1991.

[SC12] Reihaneh Safavi-Naini and Ran Canetti, editors. *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*. Springer, 2012.

[SGY13] Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors. *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. ACM, 2013.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[SP11] Jyh-Charn Liu Shi Pu, Pu Duan. Fastplay-a parallelization model and implementation of SMC on CUDA based GPU cluster architecture. Cryptology ePrint Archive, Report 2011/097, 2011. `http://eprint.iacr.org/`.

[SS11] Abhi Shelat and Chih-Hao Shen. Two-output secure computation with malicious adversaries. In Paterson [Pat11], pages 386–405.

[SS13] Abhi Shelat and Chih-Hao Shen. Fast two-party secure computation with minimal assumptions. In Sadeghi et al. [SGY13], pages 523–534.

[ST12] Nigel Smart and Stefan Tillich. Circuits of basic functions suitable for MPC and FHE, 2012. `http://www.cs.bris.ac.uk/Research/CryptographySecurity/MPC/`.

[Wag08] David Wagner, editor. *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, volume 5157 of *Lecture Notes in Computer Science*. Springer, 2008.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 160–164. IEEE Computer Society, 1982.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 162–167. IEEE Computer Society, 1986.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Oswald and Fischlin [OF15], pages 220–250.

Appendix

Overview of Variables

A list of variable names and their meaning is given in Table 1 and Table 2.

Symbol	Meaning
s	Statistical security parameter.
κ	Computational security parameter.
ℓ	The circuit replication factor, $\Theta(s)$.
p	The polynomial replication factor, $\Theta(s)$.
Δ_j	The global key for the j'th garbled circuit.
$P_i(j)$	The j'th values of the i'th polynomial in the field $\mathbb{F}_{2^\kappa}$.

Table 1: Overview of the free parameters of the protocol along with their meaning.

Symbol	Meaning
$\mathcal{G}$	A tuple containing the algorithms of a garbling scheme.
Gb	A randomized algorithm generating a garbled circuit.
Ev	A deterministic algorithm evaluating a garbled circuit.
ev	A deterministic algorithm evaluating a plain circuit.
En	A deterministic algorithm encoding a plain input into a garbled input.
De	A deterministic algorithm decoding a garbled output into a plain output.
Ve	A deterministic algorithm verifying the construction of a garbled circuit.
f	The plain description of a Boolean circuit.
F	A garbled circuit.
X	The garbled input to a garbled circuit.
Y	The garbled output of a garbled circuit.
e	The data used to construct a specific garbled input to a garbled circuit.
d	The data used to decode a specific garbled output of a garbled circuit.
x	A bit string representing the input to a Boolean circuit.
y	A bit string representing the output of a Boolean circuit.
x_{A}	A bit string representing A's input to a Boolean circuit.
x_{B}	A bit string representing B's input to a Boolean circuit.
y_{A}	A bit string representing A's output of a Boolean circuit.
y_{B}	A bit string representing B's output of a Boolean circuit.
n_{A}	The length of A's input to a Boolean circuit.
n_{B}	The length of B's input to a Boolean circuit.
w	The amount of wires in f.
q	The amount of gates in f.
t	The fan-in size (the amount of input wires) to a gate.
d	The "type" of XOR gate in a FleXOR scheme. I.e. the amount of ciphertexts needed to transmit of a FleXOR gate.
L	A function mapping the left input wire of a gate to the gate where this wire comes from.
R	A function mapping the right input wire of a gate to the gate where this wire comes from.
W	A function mapping the input wire of a gate to the gate where this wire comes from.
G	A function describing the function (e.g. AND) of a given gate.
I	A function describing the amount of input wires to a given gate.

Table 2: Overview of the algorithms are variables of a garbling scheme.

Acronyms

2PC Secure Two-Party Computation

DAG Directed Acyclic Graph

DDH Decisional Diffie-Hellman

DKC Dual-Key Cipher

GC Garbled Circuit

GPU Graphics Processing Unit

iff if and only if

KDF Key Derivation Function

KDM Key Dependent Message

LEGO Large Efficient Garbled-circuit Optimization

LSB Least Significant Bit

LPN Learning Parity with Noise

LWE Learning With Errors

MAC Message Authentication Code

MIMD Multiple Instruction, Multiple Data

MPC Secure Multi-Party Computation

MSB Most Significant Bit

NIZK Non-Interactive Zero Knowledge

OT Oblivious Transfer

PFE Private Function Evaluation

PPT Probabilistic Polynomial Time

PRF Pseudorandom Function

PRG Pseudorandom Generator

RAM Random Access Memory

ROM Random Oracle Model

SFE Secure Function Evaluation

SIMD Same Instruction, Multiple Data

UC Universal Composition

VSS Verifiable Secret Sharing

wlog without loss of generality

ZK Zero Knowledge

ZKAoK Zero Knowledge Argument of Knowledge

Index

Adaptive security, 23
  Garbled circuits, 40–41
Arithmetic garbled circuits, 41

Coin-tossing, 27–28
Commitments, 24–27
CUDA, 113–114
Cut-and-choose, 6

Dishonest majority, 8
DKC, *see* Dual-Key Cipher
Dual execution, 7
Dual-Key Cipher, 34–36

External value, 4

FleXOR, 11, 66–69
Forge-and-lose, 7, 14–15
Free-XOR, 5, 38, 42

Garbling, 3
Gate garbling, 37–38
GPGPU, 113

Hybrid model, 23

Information theoretic garbled circuits, 41
Input consistency, 7, 13

KDF, *see* Key Derivation Function
Key Derivation Function, 48–55
Key size preserving, 38

Leakage function, *see* Side-information function
LEGO, 7, 18
Link, 15, 83

Malicious security, 23
MiniMAC, 9

Negligible function, 21

Oblivious Transfer, 28
  Batch, 84–87
  Extension, 80–83
OT, *see* Oblivious Transfer

Pairwise independent, 93
Permutation bit, 4
Polynomial
  Consistency, 16–17
Preprocessing model, 8–9, 124

Random Oracle Model, 24
Replication factor, 7
Reusable garbled circuits, 42
ROM, *see* Random Oracle Model
Row reduction, 5

Secret sharing, 8
Security
  Computational, 21
  Statistical, 13, 21
Selective failure, 7, 13–14
Semi-honest security, 23
Side-information function, 32, 38
SIMD, 113
SPDZ, 9
Static security, 23
Structure free garbling, 38–39

Threshold security, 24
TinyOT, 8

UC, *see* Universal Composability
Uniform difference property, 93
Universal Composability, 22–23
Universal hash function, 93–95

Zero-Knowledge, 70–71
ZK, *see* Zero-Knowledge