Faster Maliciously Secure Two-Party Computation Using the GPU
Full version

Tore Kasper Frederiksen, Thomas P. Jakobsen, and Jesper Buus Nielsen *
April 17, 2014

Department of Computer Science, Aarhus University
{jot2re|tpj|jbn}@cs.au.dk

Abstract We present a new protocol for maliciously secure two-party computation based on cut-and-choose of garbled circuits using the recent idea of "forge-and-loose" which eliminates around a factor 3 of garbled circuits that needs to be constructed and evaluated. Our protocol introduces a new way to realize the "forge-and-loose" approach which avoids an auxiliary secure two-party computation protocol, does not rely on any number theoretic assumptions and parallelizes well in a same instruction, multiple data (SIMD) framework.
With this approach we prove our protocol universally composable-secure against a malicious adversary assuming access to oblivious transfer, commitment and coin-tossing functionalities in the random oracle model.
Finally, we construct, and benchmark, a SIMD implementation of this protocol using a GPU as a massive SIMD device. The findings compare favorably with all previous implementations of maliciously secure, two-party computation.

* Partially supported by the European Research Commission Starting Grant 279447 and the Danish National Research Foundation and The National Science Foundation of China (grant 61061130540) for the Sino-Danish Center for the Theory of Interactive Computation. Tore and Thomas are supported by Danish Council for Independent Research Starting Grant 10-081612.

1 Introduction

Background. Secure two-party computation (2PC) is the area of cryptography concerned with two mutually distrusting parties who wish to securely compute an arbitrary function with private output based on their independent and private input. A bit more formally Alice has the input x, Bob the input y, and they wish to compute the function $f(x, y) = z$ without Bob learning anything about y and without Alice learning anything about x.[1]

This area was introduced in 1982 by Andrew Yao [Yao82, Yao86], specifically for the *semi-honest* case where both parties are assumed to follow the prescribed protocol and only try to compromise security by extracting information from their own views of the protocol execution. Yao showed how to construct a protocol preventing this using a technique referred to as the *garbled circuit approach*. This approach involves having one party (*the constructor*), say, Alice, encrypt, or "garble", a Boolean circuit computing the desired functionality. This is done by choosing two random keys for each wire in the circuit, one representing a value of 0 and another representing a value of 1. Each gate in the garbled circuit is then constructed such that Bob (*the evaluator*), given only one key for each input wire, can compute only one key for the output wire, namely the key corresponding to the bit that the gate is supposed to output (for example, the logical AND of the two input bits). Alice sends the garbled circuit to Bob and makes sure that for each input wire, Bob learns only one of that wire's keys. For Alice's own input, she simply sends these keys to Bob. Using an oblivious transfer (OT) protocol, Bob also learns one key for each input wire corresponding to his own input, without Alice learning Bob's input. Now, given one key for each input wire, Bob can then evaluate the whole garbled circuit, gate by gate, while at the same time, he cannot learn which bits flow on the wires. Only when he reaches the output wires, he uses auxiliary information to learn which bits the keys encode. See [LP09] for a thorough description of Yao's scheme.

A major reason why Yao's original protocol is only secure against a semi-honest adversary is that Bob cannot be sure that the garbled circuit he receives from Alice has been garbled correctly. One way to cope with this, and achieve *malicious* security where a corrupt party might deviate from the prescribed protocol in an arbitrary manner, is with a *cut-and-choose* approach: Instead of sending one circuit, Alice sends several independently garbled versions of the circuit to Bob. Bob then randomly selects some of these, called the *check circuits*, which are then degarbled, allowing Bob to verify that they do indeed correspond to the correct function f. If this is the case, he knows that the remaining circuits, called *evaluation circuits*, contain a majority of correct circuits except with negligible probability in the security parameter.

Doing cut-and-choose on several garbled circuits introduces some other issues that has 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, some mechanisms must ensure *input consistency* amongst the circuits: Alice and Bob must give the same input to all the circuits. Information about Alice's input will leak to Bob if he gets to evaluate the circuit on different inputs. But information might also leak to Alice, depending on the function the circuit is computing, if she gives inconsistent input. Another, more subtle, issue we must handle when wishing malicious security for the garbled circuit approach is known as the *selective failure attack*: Since Alice will supply Bob with the keys in correspondance with his input bits through an OT Alice can simply input garbage for one of the keys, e.g, the 0-key for the first bit of his input. If Bob aborts the protocol, because he cannot evaluate a garbled circuit when one of the keys is garbage, Alice 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!

Cut-and-choose of Garbled Circuits. In recent years several protocols have been presented which obtain malicious security by combining Yao's original protocol with cut-and-choose on the circuits [LP07, PSSW09, LP11, sS11, KSS12, Bra13, FN13, HKE13, Lin13, MR13, SS13]. They mainly differ in the way they handle the above issues and, as a consequence, in how many circuits are needed to send from Alice to Bob. We call the amount of circuits needed to be sent from Alice to Bob for the *replication factor*. Still, most of these approaches use the fact that a few of the evaluation circuits may be incorrect by letting Bob output the *majority* of the results output by the evaluation circuits.

[1] Who should learn the output z can differ and in the most general case there might be a specific output for Alice and one for Bob, i.e., $f(x, y) = z = (z_A, z_B) = (f_A(x, y), f_B(x, y))$ where Alice should only learn z_A and Bob only z_B. In this paper we only consider the case where Bob learns the entire output z. If output for Alice is also needed then this can be achieved using general approaches. We consider this in more detail in Appendix H.

Recent results [Bra13, HKE13, Lin13] avoid the need to take majority of the evaluation circuits. This means that Alice only succeeds in cheating if *all* evaluation circuits are maliciously constructed while *all* check circuits are honestly constructed, which happens with noticeably less probability than only the majority of the evaluations circuits are honestly constructed, thus making it possible to run a protocol with noticeably less garbled circuits. The main idea of [Lin13] and [Bra13] (called "forge-and-loose") is to make sure that if Alice cheats, that is, if two evaluation circuits yield different results, then this enables Bob to learn Alice's private input x. He can then use this to compute locally, unencrypted, the correct input $f(x, y)$. In [Lin13] Bob learns Alice's input through an auxiliary secure protocol execution, using as input part of the transcript of the "original" protocol execution, as proof that Alice is corrupt. In [Bra13] a special kind of commitments are used which leaks some auxiliary information if the output of two garbled circuits differs. This auxiliary information can then be used to learn Alice's input. The idea of [HKE13] is to have both parties play the role of the circuit constructor *and* circuit evaluator respectively in two simultaneous executions of a cut-and-choose of garbled circuits protocol. The output of the protocol is decided to be the output at least one of the garbled circuits, *each* of the parties constructed, agree on.

Other Approaches to Malicious Security. Several other approaches to maliciously secure two party computation exist, based on Yao's garbled circuits as well as other novel constructions.

In [NO09] Nielsen and Orlandi introduced the notion of cut-and-choose at the gate level in order to achieve malicious security for Yao's garbled circuits, popularly referred to as the LEGO approach. This gave an asymptotic increase in efficiency of $O(\log(|C|))$ where $|C|$ is the amount of gates in the circuit to compute. However, their approach relied on heavy group based computations (as the protocol required homomorphic commitments) for each gate in the circuit.

In the later work [FJN$^+$13] the group operations per gate were removed, while retaining the same asymptotic efficiency of the original LEGO approach. The authors achieved this by introducing a new approach for constructing xor-homomorphic commitments based on error correcting codes.

Another approach, *not* using garbled gates, was introduced in [NNOB12] where heavy usage of oblivious transfers was used to construct verified and authenticated Boolean gates. To achieve efficiency the authors constructed a highly efficient maliciously secure "OT extension", which took a few random OTs as "seeds" and then used a random oracle to extend these to polynomially many random OTs.

Yet another approach, not necessarily using Yao's garbled circuits, is the idea of "MPC-in-the-head" from [IKOS07, IPS08] where the parties run a semi-honestly secure two-party protocol, but within this they emulate $n > 2$ parties that run a virtual second protocol secure against a malicious minority. This results in a final protocol maliciously secure against a dishonest majority. Another idea is to base the protocol on preprocessed constrained randomness as in [DZ13].

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 *both* parties. [MF06, HKE12, MR13].

Besides all of the above approaches a whole other subfield of secure multi-party computation exists where the function to be evaluated is arithmetic (which can clearly also be used to evaluate Boolean circuits when the field for the arithmetic operations is $\mathbb{F}_2$). Some of the more recent works in this area include [BDOZ11] (semi-honest security) and [DPSZ12, DKL$^+$13] (malicious security).

Motivation. The area of 2PC and multi-party computation, MPC, (when more than two parties supply input) is very interesting as efficient solutions yield several practical applications. The first case of this was described in [BCD$^+$09] where MPC was used for deciding the market clearing price of suger beet quotas in Denmark. Still, several other applications of 2PC and MPC exists such as voting, anonymous identification, privacy preserving database queries etc. For this reason we believe that it is highly relevant to find practically efficient protocols for 2PC and MPC.

Our Results. We present a new maliciously secure two-party protocol based on cut-and-choose of garbled circuits. The protocol combines previous ideas, mainly from [Bra13, Lin13, HKE13] and [FN13, SS13] with new optimizations to a protocol with the following benefits:

1. A small replication factor for the same reasons as [Bra13, Lin13].[2] But unlike [Lin13] our protocol eliminates the need for running an auxiliary protocol and unlike [HKE13] we eliminate the need of both parties both constructing *and* evaluating $O(s)$ garbled circuits.
2. A very lightweight approach to ensure that Alice gives consistent input to all garbled circuits used for evaluation.
3. Reliance only on lightweight primitives (in practice a hash function) and few OTs.
4. A large degree of same instruction, multiple data (SIMD) parallelization, and therefore direct benefits from access to a large amount of computation cores.
5. A security proof in the Universally Composable (UC) model [Can01].[3] We then show how to realize a highly efficient version of the protocol in the Random Oracle Model (ROM) without number theoretic assumptions, only assuming access to an OT functionality.

Finally, we also provide an implementation of the protocol written in C and CUDA, using a consumer grade GPU for exploiting the inherent parallelism in the protocol. Based on benchmarks of this we show that our approach is more efficient than any other existing solutions based on garbled gates.

Outline. We start by going through the overall ideas of our scheme, along with the primitives we need in Section 2. We then continue with a high level description of our protocol in Section 3. A discussion of our implementation along with experimental results are given in Section 4 and we finally conclude the paper in Section 5 by giving a discussion of the findings from our implementation and directions for future work. In Appendix A we give two tables describing the variable and symbols used throughout the paper. Detailed description of the primitives we need are given in Appendix B and Appendix C, then in Appendix D a fully detailed description of our protocol is given and in Appendix E we give a full proof of security with supplementing details in Appendix F, Appendix G and Appendix H.

Notation. We assume Alice is the constructor and Bob the evaluator and that the functionality they wish to compute is $f(x, y) = z$, where Alice gives input x, Bob gives input y and only one party, Bob, receives the output z. We denote the amount of bits of x as $|x| = m$, the amount of bits of y as $|y| = n$ and the amount of bits of z as $|z| = o$. We call the Boolean circuit computing the functionality $f(\cdot, \cdot)$ for C and assume it consist of gates with fan-in 2 and fan-out 1 such as AND, XOR, OR etc. Furthermore, we say the size of C, $|C|$, is the amount of non-XOR Boolean gates. When considering a garbled version of this circuit we add a tilde, i.e., $\tilde{C}$. In a garbled circuit we define the *semantic* value of a key to be the bit it represents. We furthermore define the *plain* values or functions to be the unencrypted, unhidden values or functionality, i.e., the actual function $f(x, y) = z$ with inputs x and y and output z. We use s to represent a statistical security parameter, and we let κ be a computational security parameter. Technically, this means that for any fixed s and any adversary, the advantage of the adversary is $2^{-s} + \mathrm{negl}(\kappa)$ for a negligible function negl. I.e., the advantage of any adversary goes to 2^{-s} faster than any inverse polynomial in the computational security parameter. If $s = \Omega(\kappa)$, this means that the advantage is negligible. If $s = O(\log(\kappa))$ or $s = O(1)$ we only have covert security (see [HL10] for a detailed description of this concept). Let $[n]$ denote the set of integers $\{1, 2, \ldots, n\}$ and let $\mathrm{H}(\cdot)$ denote a hash function modeled as a random oracle giving κ bits of output. We use subscript to denote index, i.e. x_i denotes the i'th bits of a vector x and $m_{i,j}$ denotes the i'th row and j'th column of the matrix m. Finally we let $x \in_R S$ mean that x is sampled uniformly at random from the set S. An overview of the various variables and parameters along with their meaning is given in Appendix A.

2 The Big Picture

We assume the reader is familiar with free-xor Yao circuits [KS08], otherwise see Appendix B. The overall strategy is as follows:

1. *Garbling* Alice garbles a bunch of circuits and sends them to Bob.

[2] We do not strictly obtain the same replication factor as [Bra13,Lin13]. Where the replication factor is s, when s is the statistical security parameter, ours will be slightly larger as in [HKE13]. See Appendix F for details.

[3] Specifically we provide a UC-proof of our protocol in a hybrid/random oracle model with OT, commitments and coin tossing, and a secure garbling scheme with free-xor [KS08].

2. *Oblivious Transfer* The parties engage in an OT protocol, secure against malicious adversaries, such that Bob learns the keys for each garbled circuit in correspondance with his plain input to the functionality they wish to compute.
3. *Cut-and-Choose* The parties use a coin tossing protocol to select around half of the garbled circuits as *check* circuits and the remaining as *evaluation* circuits. Bob then verifies that the check circuits have been constructed correctly and receives from Alice the keys corresponding to her input to the evaluation circuits.
4. *Evaluation* Bob will then use his own input keys, which he learned from the OT, along with Alice's input keys to evaluate all the evaluation circuits. Using the results from the evaluated circuits he finds the correct output.

This generic scheme leaves open how to handle the problems that arise due to the fact that the parties might act maliciously and due to multiple circuits being garbled and evaluated instead of just one circuit. These problems are (1) selective failure attacks, (2) consistency of the parties' input and (3) how to know which output is the correct one in the evaluation phase. Many different approaches exist to solve these problems with different levels of efficiency and security assumptions. In the following sections we shortly discuss various solutions and in particular, how we solve these issues.

2.1 Avoiding Selective Failure Attacks

Besides the obvious demand that we want the output of a 2PC protocol to be correct in accordance with the functionality and inputs, i.e., $z = f(x, y)$, we also want to ensure that a party cannot make its input depend on the other party's input. Furthermore, we want privacy, that is, it should not be possible for one party to deduce anything about the other party's private input, except for what is learned from the output $z = f(x, y)$. In particular this means that if Alice is malicious, the protocol must ensure that the part of the behavior of Bob that is observable by Alice, does not in any way depend on Bob's input. It is because of this we need to handle the potential selective failure attack Alice can do on the keys given as input to the OT. This attack was first pointed out by [KS06, MF06] and is essentially due to the fact that the OTs are separated from the garbling: Alice can use different inputs to the OT than she uses elsewhere in the protocol. Having Alice commit to her input keys does not help since she can still use different keys in the OTs than those used in the commitments.

Mohassel and Franklin [MF06] suggested solving this by means of *committing OT*'s which makes it possible for Bob to verify Alice's input to the OT against the keys in the check-circuits. Lindell and Pinkas [LP07] solved this by increasing the input wires of Bob, letting Bob replace his original input with constrained randomness, where certain input wires xor'ed together would give his real input. This eliminates the attack since it is hard for Alice to "guess" this randomness, which is needed in order for her to successfully complete her attack. Unfortunately, this also increases the size of Bob's input from n bits to $\max(4n, 8s)$ bits. This technique was later refined by shelat and Shen to give a smaller increase in the amount of input bits needed by Bob [SS13].

Another approach works by introducing a tighter coupling between the OTs and the rest of the protocol. Lindell and Pinkas [LP11] introduced the *single-choice batch cut-and-choose oblivious transfer* based on Diffe-Hellman pseudo-random synthesizers, which were used to "glue" together the OTs and the garbled circuits. This was also used to ensure consistency of the parties' inputs to the different garbled circuits.

In our solution we handle the selective failure problem in the same manner as in [LP07]. The approach involves increasing the amount of input bits of the evaluator from a string y of n bits to a string $\bar{y}$ of $\bar{n} = \max(4n, 8s)$ bits. More specifically, what we do is to have Bob choose a random binary matrix $M^{\mathsf{Sec}} \in_R \{0,1\}^{\bar{n}} \times \{0,1\}^n$ and his input randomly, i.e., $\bar{y} \in_R \{0,1\}^{\bar{n}}$, but under the constraint that $M^{\mathsf{Sec}} \cdot \bar{y} = y$ where y is the "true" input. Thus the augmented functionality computes exactly the same as the original. Still, the idea of this approach is that if a selective failure attack is done, using the augmented function will not leak any useful information: as the entire vector $\bar{y}$ is random, learning a few bits of this vector will only give the adversary a negligible advantage in learning one of the constructor's true input bits. This follows from the fact that the other bits of $\bar{y}$ will be used to hide each of the actual bits of y. The details and a full proof of security of this approach can be found in [LP07].

2.2 Ensuring Consistency of Bob's Input

Making just multiple calls to a basic protocol for OT, Bob could input different choice bits for each of the circuits. This would allow him to evaluate the function on different inputs. That is, he would learn both $f(x, y)$ and $f(x, y')$ for two different inputs y and y' of his own choice. This obviously leaks too much information to Bob.

To avoid this we must make sure that for each of the bits y_i in Bob's input, he gives the same choice bit for all of the garbled circuits in the OTs. This can easily be achieved by doing fewer OTs, but of longer bit strings: For each OT Alice inputs a concatenation of the 0-, respectively 1-keys for a given input wire for all circuits, Bob inputs only one choice bit and receives the keys for all the circuits corresponding to this choice bit on the given input wire. This approach is used in most previous results [LP07, LP11, sS11, FN13, Lin13, SS13]. In the ROM this can be very efficiently realized using black box access to a one-out-of-two OT of strings of length s [FN13, SS13]. For more details and its realization in the ROM, see Appendix C.3.

2.3 Ensuring Consistency of Alice's Input

To see why there is also a problem if Alice is inconsistent in her inputs, assume the functionality to be computed is the inner product, that Alice inputs the strings $\langle 100\ldots0\rangle$, $\langle 010\ldots0\rangle, \ldots, \langle 000\ldots1\rangle$ and that she is the party who is supposed to learn the output of the computation. In this case what she will learn is the majority of bits in Bob's input string, which is clearly too much information.

One could perhaps think that this problem can be solved by having Bob abort if the results of the circuit evaluations diverge. But aborting based on this may leak information about Bob's secret input to Alice. To see this, note that the function f might give the same outcome for several different inputs from Alice, but *only* if Bob's input comes from some particular subset of his possible inputs. Alice would learn whether Bob's input belongs to this subset based on whether Bob aborts or not. See [LP07] for a concrete example of such a function.

A similar attack is possible if Bob instead of aborting, randomly picks one of the different outputs from the circuits: For certain functions f, Alice would be able to control what the majority of outputs would be and to let this depend on Bob's secret input.[4]

Since our aim is to allow secure evaluation of *any* function (and since it is not obvious which functions allow this kind of leakage) we therefore must ensure that all circuit evaluations are based on the same inputs from Alice.

Lindell and Pinkas [LP07] embedded a consistency check of Alice's input into the cut-and-choose step itself, resulting in $O(ms^2)$ commitments. The authors later improved on this [LP11] by introducing a protocol for *single-choice cut-and-choose batch oblivious transfer* based on Diffie-Hellmann pseudo-random synthesizers which ensures both input consistency and protections against selective failure attacks. This avoids the quadratic amount of commitments, and instead requires only $O(ms)$ operations, though these operations include heavy modular exponentiations. Shelat and Shen [sS11] ensures input consistency using another approach based on claw-free functions.

We follow a different approach, introduced independently by Frederiksen and Nielsen [FN13] and shelat and Shen [SS13]. The idea is to augment the functionality such that Alice's input, apart from being used in the original computation of f is also fed into a universal hash function decided by Bob. This means that Bob learns the hashed value of Alice's input for each circuit. We can now safely let Bob abort if any of these hash values diverge since having Bob abort in this case does not reveal anything to Alice about his input. Intuitively, this approach is secure if the output of the augmented functionality does not leak any information about Alice's input and Alice commits to her input choices before learning the specific hash function to be used. This will ensure that Bob does not learn anything about Alice's input and that even an unbounded Alice only has negligible probability in s of finding two inputs that collide. We achieve the

[4] If Bob aborts this is an example of a *selective failure* attack. If he instead takes majority or picks a random result in this case it is sometimes called a *selective input* attack. There seems to be some confusion about these terms. Selective input attacks are used to describe the attacks where Alice can deduce information about Bob's private input by submitting different inputs to different circuits. So, enforcing consistent input from Alice is the same as preventing all selective input attacks, and – if Bob aborts when results diverge – not enforcing consistent input from Alice would make the protocol vulnerable to a specific selective failure attack.

first property by having Alice 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.

A specific, but simple way to do this is as follows [FN13]: Assume the functionality we wish to compute is defined by f as $f(x, y) = z$. We then define a new function f' as $f'(x', y') = f'(x\|a,\ y\|b) = z\|c$ where $a \in_R \{0,1\}^s$, $b \in_R \{0,1\}^{m+s-1}$ and $c \in \{0,1\}^s$. To compute c we define a matrix $M^{\mathsf{In}} \in \{0,1\}^s \times \{0,1\}^{|x|}$ where the i'th row is the first $m = |x|$ bits of $b << i$ where $<<$ denotes the bitwise left shift. Specifically the j'th bit of the i'th row is the $i + j$'th bit of the binary vector b, i.e., $M^{\mathsf{In}}_{i,j} = b_{i+j-1}$. Using this matrix the computation of c is defined as $c = (M^{\mathsf{In}} \cdot x) \oplus a$, assuming all binary vectors are in column form. With this modification the new function computes the same as the original, but requires s extra random bits of input from Alice and $m + s - 1$ extra random bits from Bob. The s extra output bits will work as digest bits and can be used to check that Alice is consistent with her inputs to the circuits by verifying that they are the same in all the garbled circuits which Bob evaluates.

Next, notice that it is in fact not needed to have Bob specify the specific hash function as part of his input. Instead he can send Alice the vector b defining the matrix M^{In} and have her garble the circuits to compute the hash function this vector specifies [SS13]. 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.

This means that the binary string $b \in \{0,1\}^{m+s-1}$ defines a family of universal hash functions mapping m bits to s bits. In turn a sampling of such a function is simply a random choice of the string b. We will call a specific function from this family for $\mathbb{H}^{\mathsf{In}}$ and call the actual circuit augmentation for $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ where a will be Alice's auxiliary input sampled from $\{0,1\}^s$.

A problem with this approach is that Alice now can try to find a collision for the hash function before she gives her input keys to Bob. This is a significant problem as collisions for universal hash functions can be easy to find. Fortunately, this is only a problem if Alice is not committed to her input before she is informed of the hash function. If we let her commit to her input and *then* let Bob reveal his choice of hash function to her we will not have this problem.

It should be noted that in [SS13] another universal hash function was used, which required $2s + \log(s)$ auxiliary random bits of input from Alice, unlike only s in [FN13]. However, the authors of [FN13] did not observe that it was enough to send M^{In} in plain, nor did they actually *prove* their construction secure. In Appendix E we prove that the universal hash function used in [FN13] is secure in our setting when M^{In} is sent in plain after Alice commits to her inputs.

We follow the approach of circuit augmentation outlined above. Concretely we implement the consistency check by having Alice commit to her input before the cut-and-choose phase. Then we have Bob giving her the specification of the hash function. Alice then garbles the circuit with the hash function augmentation. When Bob later has evaluated the evaluation circuits, he aborts if the hash values from any two of the augmentations of these circuits differ.

2.4 Computing the Correct Output

Despite the input consistency of Alice being enforced as discussed above, Bob may still end up with more than one output when he evaluates the evaluation circuits. This is due to the fact that the cut-and-choose technique does not ensure that all the remaining evaluation circuits are good. With non-negligible probability some of them may not evaluate at all and some may evaluate, but result in diverging output.

Given several different outputs, Bob cannot just abort the computation or choose a random result. This would make him vulnerable to the same kind of selective failure attack as we discussed in connection with the input consistency. However, Lindell and Pinkas [LP07] showed that for a statistical security parameter s, if the number of garbled circuits, i.e., the replication factor, is greater than cs for some constant c and if half are chosen as check circuits, then, except with negligible probability, the majority of the outputs are well-defined and are in fact the correct output.

For realistic circuit sizes, the replication factor has proven to be very important for the performance of the overall protocol [PSSW09, KSS12, FN13]. For this reason, recent effort has been made to reduce the replication factor. As a start, [PSSW09] claimed that [LP07] only requires a replication factor $\ell \geq 4s$ (to achieve failure probability at most 2^{-s}). The analysis was later improved by [LP11] showing that one only needed a replication factor $\ell \geq 3.22s$. Finally, [sS11] refined this further by showing that taking 60% of the circuits to be check circuits, a replication factor $\ell \geq 3.13s$ is sufficient.

All of these results are based on taking majority of the outputs from the evaluation circuits. As mentioned in the introduction, recently Brandão [Bra13] and Lindell [Lin13] independently showed how to avoid taking majority by instead enabling Bob to learn Alice's private input if the results diverge, and thereby letting Bob compute the correct result in "clear". Doing so allows for a replication factor of only s to achieve an error probability of 2^{-s}, which is a considerable improvement. Independently Huang *et al.* [HKE13] showed the same, but with a slightly higher replication factor. In particular, doing the protocol with a replication factor ℓ gives an error probability of at most $2^{-\ell+O(\log \ell)}$. For comparison, note that for a typical application with $s = 40$, [LP11] and [sS11] need 129 and 125 circuits respectively, while [Bra13, Lin13] requires only 40 circuits to be garbled.

Our protocol is based on the same idea as [Bra13, Lin13], namely to make it possible for Bob to recover Alice's plain input x if any inconsistency arrives in the evaluation circuits and, in turn, let Bob evaluate the functionality in plain. To do this, [Lin13]relies on a secondary secure computation "inside" the protocol in order for Bob to learn x. While being independent of the size of the original circuit, this still introduces a significant overhead. Furthermore, [Lin13] requires a number of modular exponentiations linear in the input size and reliance on the DDH assumption in order to ensure Alice inputs the correct x to this secondary computation. In [Bra13] Bob gets to recover Alice's input by having her commit to her input keys and the keys on the circuit output wires using trapdoor commitments where a the trapdoor can be efficiently computed if two different output keys are achieved for the same wire, but in different garbled circuits.

Our main theoretical contribution arrives in the way we allow Bob 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 strictly require any specific computational assumption. [5]

Let $t = \ell/2$, that is, t is the amount of check circuits. Also, let $k'^0_{i,j}$ and $k'^1_{i,j}$ be the zero and one keys for the i'th output wire on the j'th circuit. We use the free-xor technique [KS08] and let Alice garble each circuit with a different Δ. That is, for circuit j and all wires k Alice chooses the wire keys such that $k^0_{k,j} \oplus k^1_{k,j} = \Delta_j$.

We select the replication factor ℓ such that, except with probability 2^{-s} if Alice 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 Bob will be able to efficiently compute the Δ values for one of those circuits. Knowing the Δ value used to garble a circuit allows Bob to learn the input x that Alice submitted for that circuit.

Using Polynomials to Compute Δ's. We now explain how we let Bob compute Δs for those circuits whose outputs diverge.

As part of the garbling phase, Alice associates a polynomial of degree at most t, P_i with each output wire in the circuit, where $t = \ell/2$. Now for each garbled circuit $j \in [\ell]$ and each output wire $i \in [o]$ Alice associates a point on the corresponding polynomial, i.e., a value $P_i(j)$. 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 $\left\{\left\{k'^0_{i,j}\right\}_{i=1}^{o}\right\}_{j=1}^{\ell}$ and point $P_i(j)$ we associate the "link" $L_{i,j}$ (see Appendix B for details). For simplicity assume that the link is realized as follows $L_{i,j} = (r_{i,j}, s_{i,j}, h_{i,j}, g_{i,j})$ where $r_{i,j}, s_{i,j} \in_R \{0,1\}^\kappa$ are uniformly random sampled elements, $h_{i,j} = \mathrm{H}\left(P_i(j), r_{i,j}\right) \oplus k'^0_{i,j}$ and $g_{i,j} = \mathrm{H}\left(k'^0_{i,j}, s_{i,j}\right) \oplus P_i(j)$. Alice sends all these links to Bob.

Next notice, that after the cut-and-choose step, Bob will learn all the 0-keys on the output wires for t garbled circuits and in turn be able to learn t 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, Bob will be able to learn one more point on one of the polynomials. This will give him a total of $t + 1$ points on a polynomial of degree at

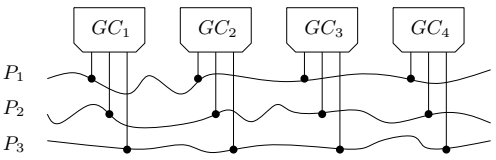

Figure 1. Four garbled circuits with three output wires. Each 0-key on these output wires is linked to a particular point on a polynomial of degree at most 2.

[5] We do require access to OT, but any one-out-of-two OT can be used, including OT extensions.

most t and he will thus be able to do polynomial interpolation and learn all the points on this polynomial. This again will make it possible for Bob to learn the 0-key on the 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 specifically, suppose that Bob 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, w.l.o.g. that this is the case for output wire 1 in circuit 1 and circuit 2, such that Bob learns both $k'^0_{1,1}$ and $k'^1_{1,2} = k'^0_{1,2} \oplus \Delta_2$. Using the link $L_{1,1} = (r_{1,1}, s_{1,1}, h_{1,1}, g_{1,1})$ Bob can compute the point $P_1(1)$ as $P_1(1) = g_{1,1} \oplus \mathrm{H}\left(k'^0_{1,1}, s_{1,1}\right)$. This gives him a total of $t+1$ points on the polynomial P_1 (remember that he already knew t 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 $P_1(2)$. Finally, he can then compute the following:

$$h_{1,2} \oplus \mathrm{H}\left(P_1(2), r_{1,2}\right) \oplus k'^1_{1,2} = k'^0_{1,2} \oplus k'^1_{1,2} = \Delta_2 \,.$$

Using Δ_2 he can completely degarble the second garbled circuit, and in turn, also learn Alice's plain input to this circuit. This is so, as knowing the Δ value and one key for all wires allows to compute both keys for all wires and thus completely open up a garbled circuit.

This approach generalizes to each output wire in two circuits where one is the 0-key and other the 1-key. Thus, assuming we have enough output wires with different values, this makes it possible to find Δ for all garbled circuit, and in turn Alice's plain input to all garbled circuits.

We want to emphasize, however, that Bob does not use the Δ values 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 Bob only opens up the input layer to learn the semantic value of the input keys of Alice to the circuit. He then evaluates the universal hash function on Alice's input in plain and compare it with the semantic meaning of the garbled evaluation of the universal hash function. Then Bob 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 this part of the circuit is known by Alice already. Now, for all the remaining circuits, the inputs of Alice 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 Bob after Alice committed to her inputs. Hence Bob 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 Alice.

Ensuring Consistency of Polynomials. We made a few assumptions in the strategy described above. That is, it only works if the polynomials are consistent, that is, if they are at most degree t *and* we have enough output wires that diverge in value to make sure Bob can learn the Δ values for each of the evaluation circuits. Since Alice 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 t. In particular she might construct these to be degree $2t$ and thus make it impossible for Bob to use polynomial interpolation to find the Δ values. We could use techniques like Kate *et al.* [KZG10] to ensure consistency 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 Alice prepares more polynomials than actually needed, and then Bob randomly chooses half of the polynomials as *check* polynomials, which Alice then reveals and Bob verifies to have degree at most t. We say that a polynomial of degree greater than t is *inconsistent* and if it has degree at most t we say it is *consistent*. Thus for Bob to accept the check polynomials they must all be consistent. After this phase, a large part of the 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 [LP07].

Ensuring consistency of the polynomials with cut-and-choose in this way 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 Bob can learn the Δ values for each of the evaluation circuits.

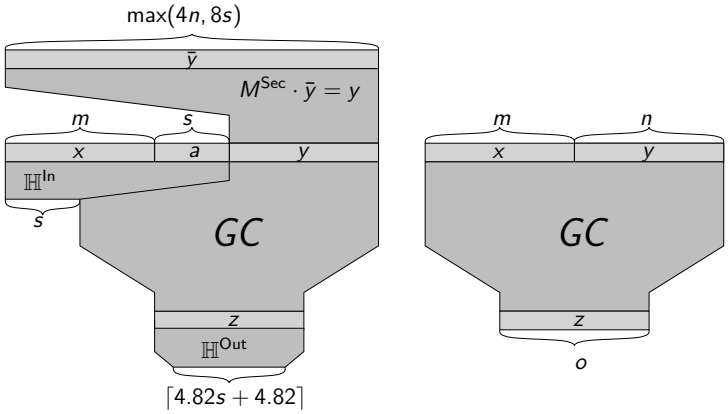

Figure 2. Illustration comparing the original circuit (right) and the augmented circuit (left). Inputs and output are shown in light grey, whereas actual computation is shown with a darker grey.

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. Intuitively, this ensures that any divergence on one bit in the original output will result in the divergence of many bits in the output of the hash function. Thus the hash function ensures that we have enough output wires which will differ in their keys, 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 keys to learn Δ for all the evaluation circuits.

Later in this paper 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 Bob will be able to learn all the Δ values of the evaluation circuits if he ends up with two or more evaluation circuits where at least one output wire diverge.

This augmentation on the output wires is done in almost the same efficient way as the augmentation to ensure input consistency of Alice. That is, we exploit the free-xor technique and let Bob send the specification of the hash function to Alice in clear. Specifically, let z be the binary output of the computation. Then we have Bob choose two random binary strings $b^{\mathsf{Out1}} \in_R \{0,1\}^{\lceil o+4.82s+3.82 \rceil}$ and $b^{\mathsf{Out2}} \in_R \{0,1\}^{\lceil 4.82s+4.82 \rceil}$. Using the first string we define a $\lceil 4.82s + 4.82 \rceil \times o$ binary matrix M^{Out} such that the j'th bit of the i'th row is the $i + j - 1$'th bit of b^{Out1}. That is, $M^{\mathsf{Out}}_{i,j} = b^{\mathsf{Out1}}_{i+j-1}$. The augmentation should then compute $\left(M^{\mathsf{Out}} \cdot z\right) \oplus b^{\mathsf{Out2}}$ assuming z is a binary vector in column form.

This means that the binary strings $b^{\mathsf{Out1}} \in \{0,1\}^{\lceil o+4.82s+4.82 \rceil}$ and $b^{\mathsf{Out2}} \in \{0,1\}^{\lceil 4.82s+4.82 \rceil}$ defines a family of universal hash functions mapping o 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 b^{Out1} and b^{Out2}. We will call a specific function from this family for $\mathbb{H}^{\mathsf{Out}}$.

An illustration of the final circuit with both augmentations is given in Fig. 2 and an informal version of the protocol itself is given in Section 3.

3 The Protocol

Assume we have access to a free-xor garbling scheme, an OT functionality, a coin-tossing functionality along with a commitment scheme supporting both computationally hiding/binding commitments and commitments where the opening to a commitment is *exactly* the message committed to. We call the last type of commitment scheme a *verifiable commitment* scheme as it allows a party to verify if a message is the opening of a commitment.

We let Alice garble ℓ versions of a Boolean circuit C using the free-xor technique [KS08], we call these garbled circuits $\tilde{C}_j$ 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 $\tilde{C}_j$, we have that $k^1_{i,j} = k^0_{i,j} \oplus \Delta_j$ in correspondance with a free-xor garbling scheme.

At an informal level our protocol can then be summarized with the phases *Setup*, *Polynomial Setup*, *Oblivious Transfer*, *Garbling*, *Commitment*, *Augmentation*, *Cut-and-Choose*, *Evaluation*, and *Reconstruction*. Of these, *Setup*, *Oblivious Transfer*, *Garbling*, and *Commitment*, are very similar to other protocol based on cut-and-choose of Yao garbled circuit, e.g., [LP07, LP11, sS11, FN13, SS13]. In Fig. 3 we sketch what happens in each of these steps and leave the specific details to Appendix D and the proof of security against a static, malicious, computationally bounded adversary to Appendix E.

1. *Setup*
   (a) Assume the function to compute is $f(x, y) = z$ where Alice inputs x, Bob inputs y and Bob learns the results z, and $x \in \{0,1\}^m$, $y \in \{0,1\}^n$ and $z \in \{0,1\}^o$.
   (b) Bob then samples a random binary matrix $M^{\mathsf{Sec}} \in_R \{0,1\}^{\bar{n}} \times \{0,1\}^n$ and a random input $\bar{y}$ of $\bar{n} = \max(4n, 8s)$ bits such that $M^{\mathsf{Sec}} \cdot \bar{y} = y$. He sends this matrix to Alice and they both agree on a function, f' with this M^{Sec} embedded such that $f'(x, \bar{y}) = f(x, y)$ and call the Boolean circuit computing this function C.
2. *Polynomial Setup*
   (a) Alice choses $p = 6s + 7$ random polynomials, P_i for $i \in [p]$, of degree $\leq t = \ell/2$, over a finite field with 2^κ elements.
   (b) Next Alice computes ℓ points on each of the polynomials, achieving a total of $\ell \cdot (6s + 7)$ points. Alice commits to each point j on each polynomial i using a verifiable commitment scheme and sends these commitments to Bob.
   (c) Alice and Bob 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 Alice sends the openings to the points committed to in the previous step. Bob then uses the points to interpolate the polynomials and verifies that they are in fact all polynomials of degree at most t. If not, he aborts.
3. *Oblivious Transfer* Alice chooses ℓ global differences for the garbling, we call these $\Delta_1, \Delta_2, ..., \Delta_\ell$. Using these values she constructs random keys for each of Bob's $\bar{n}$ input bits in correspondence with the free-xor garbling scheme for ℓ distinct garbled circuits. Alice and Bob then engage in batch OT such that Bob learns the ℓ keys in correspondence to each of his input bits y.
4. *Garbling* Alice produces ℓ garbled versions $\tilde{C}_j$ of the circuit C using the global differences and keys constructed in the previous step in the free-xor manner and sends these circuits to Bob.
5. *Commitment* Alice commits to her own choice of input keys. She also makes verifiable commitments to the concatenation of the 0- and 1-keys on each input wire in each of the garbled circuits.
6. *Augmentation*
   (a) Bob randomly samples a universal hash function $\mathbb{H}^{\mathsf{In}}$ from a family of universal hash functions mapping $|x| = m$ bits to s bits and a universal hash function $\mathbb{H}^{\mathsf{Out}}$ mapping $|z| = o$ bits to $\lceil 4.82s + 4.82 \rceil$ bits from another family of hash functions as described in the previous section.
   (b) Bob sends a description of these hash functions to Alice 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.
   (c) Alice then sends Bob the output decryption tables of the augmentations $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ and $\mathbb{H}^{\mathsf{Out}}$. Alice also sends a "link" of each of the output keys of $\mathbb{H}^{\mathsf{Out}}$ to the corresponding polynomial points. That is, for each output wire i of $\mathbb{H}^{\mathsf{Out}}$ in each garbled circuit j she sends to Bob the link $L_{i,j}$.
7. *Cut-and-Choose* Alice and Bob uses a coin-tossing protocol to agree on a random subset $\mathcal{C}$ of size $t = \ell/2$ of the garbled circuits, called the *check circuits*. Alice then sends both the 0- and 1-key for each input wire in the set of check circuits Bob then verifies the keys against the commitments to the input keys and then verifies that the check circuits were correctly constructed. Using the output keys of $\mathbb{H}^{\mathsf{Out}}$ along with the "links" from the augmentation phase Bob computes the t points on each of the $\lceil 4.82s + 4.82 \rceil$ polynomials remaining from the set-up phase.
8. *Evaluation* Alice opens the input keys in correspondence with her input x for all the evaluation circuits to Bob. Bob 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 divergent or if the output decryption table is incorrectly constructed he aborts. Otherwise he evaluates the garbled circuits. If any of the garbled circuits fails to evaluate he discards it. 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.
9. *Reconstruction*
   (a) Bob 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 expect to find in $\mathbb{H}^{\mathsf{Out}}$'s output decryption table. 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 t polynomial points from the cut-and-choose phase along with the discrepant keys of an augmented output wire in two circuits to learn $t + 1$ 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 Bob 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.
   (b) Using these Δ's along with Alice's input keys and the verifiable commitments to the keys of the input wires Bob extracts the plain values of Alice's input in all the evaluation circuits. For each input, Bob evaluates $\mathbb{H}^{\mathsf{In}}$ on the plain input and discharges 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 Alice's plain input for one of the remaining circuits and computes the function f in plain to learn the correct output. If there are no remaining circuits, Bob terminates.

Figure 3. Protocol Overview

4 Implementation

As our protocol is designed to work well in the SIMD model we did our implementation in CUDA which is a platform that supports both explicit programming of SIMD execution, along with cheap hardware, GPUs,

which supports SIMD execution. More specifically we implemented our protocol in ANSI C using the CUDA extension by NVIDIA. Some parts of our implementation is based on the code from [FN13] which gave very efficient construction and evaluation of garbled circuits (including handeling of the possible selective failure attack on Bob's input), along with an "OT extension", using a consumer GPU. We direct the reader to this paper for details on these aspects of the implementation.

4.1 Trading Assumptions for Efficiency

Since the description of our 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. In particular this means that our implementation assumes the existence of random oracles and SHA-1 is used to implement random oracle queries with a 160 bit output, thus we set $\kappa = 160$. With this assumption we use the ROM garbling scheme of [PSSW09], which only requires 4 calls to the oracle to construct a garbled gate, and only a single call to the oracle in order to evaluate a garbled gate, see Appendix B for details.

We realize a coin-tossing functionality along with a commitment scheme (that fits our needs) highly efficiently in the ROM. We also base the OTs we need on an efficient OT extension, which only assume access to $O(s)$ "seed" OTs in the ROM. See Appendix C for details on these realizations. Finally, we also implement the verification of "circuit seeds" [GMS08] to eliminate the need of sending the garbled computation tables of the check circuits.[6]

4.2 SIMD Implementation Optimizations

$\mathbb{H}^{\mathsf{In}}$ 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. This follows since each output key only depend 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 done in a SIMD manner, up to the amount of inputs of $\mathbb{H}^{\mathsf{In}} + 1$, respectively $\mathbb{H}^{\mathsf{Out}}$, i.e. $m + 1$, respectively o.[7] However, the xor operation is so light that it is in general considered overkill to do this on the GPU because of the overhead of copying data from the RAM of the host system to the RAM of the GPU and back. Thus we simply implemented the functionalities in a sequential manner using the CPU. However, this might not always be the best case as we will discuss, based on our experiments, in Section 4.5.

Commitments. The commitments, both the computationally hiding/binding and verifiable types are also done in the same manner as in [FN13] i.e. by using SHA-1 to hash the string we wish to commit to (concatenated with an ID and some pseudorandomness if it is supposed to be computationally hiding). For commitments of long strings a reduction approach is used: Concatenate some random salt to the string we need to commit to and assume it is now l bits long, then we can simply hash all independent and continues ck bits pieces in parallel, for some constant c. This results in $\lceil l/ck \rceil$ digests. We concatenate these digests and again hash all independent and continues ck sized pieces of this string in parallel. We continue in this manner until a single digest remains, which will be the commitment. The overhead, and sequential complexity, of this approach is only logarithmic in the size of the string we hash.

[6] In this approach Alice constructs each specific gabled circuit using a small seed of entropy and a pseudorandom generator (in our case SHA-1). Then, instead of sending the garbled computation tables of all the garbled circuits she sends a hash digest of each of the garbled circuits, which is much smaller than the actual garbled computation tables). If a circuit is selected as a check circuit Alice simply sends the seed for that circuit and Bob can construct the garbled computation tables himself and verify them against the digest. For the evaluation circuits Alice sends the actual garbled computation tables, which Bob verifies against the commitments to make sure Alice did not try to cheat.

[7] This can be parallelized even further: Since xor is commutative we can use a classic reduction approach for computing each output key. That is, 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 $\mathbb{H}^{\mathsf{Out}}$.

Polynomial Representation. In the protocol description we explained that the polynomials would have points in GF $\left(2^{160}\right)$ as they need to have the same size as the wire keys. However, our initial implementation showed, which was also expected, that doing polynomial interpolation on elements in GF $\left(2^{160}\right)$ was quite slow. This is expected since multiplying two elements of GF (2^κ) requires $O(\kappa^2)$ operations using the "trivial" approach. This 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 GF (2^κ) and thus it does not seem possible to hope for better complexity than $O(\kappa)$. However, when κ is small enough, multiplication and inversion in GF (2^κ) 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 $[2t] = [\ell] = [O(s)]$. Since s is a statistical security parameter and the garbled table are based on a computation security parameter of $\kappa = 160$ it will not make sense to have $s > 160$. This means that in practice the range of points we will need to evaluate polynomials on will always be strictly less than 200. Now see that $2^8 > 200$. This means that instead of using polynomials in GF $\left(2^{160}\right)$ we use 20 polynomials in GF $\left(2^8\right)$ in place of a single polynomial in GF $\left(2^{160}\right)$ – i.e., we use polynomials over the ring GF $\left(2^8\right)^{20}$, which is still a secure secret sharing scheme, c.f. [CFIK03]. This again means that we can use lookup tables to do multiplication and inversion in constant time, in a SIMD manner for each of the 20 polynomials.[8]

Polynomial Construction. The $20p$ polynomials with elements from GF $\left(2^8\right)$ are generated in a SIMD manner using the GPU based on a 160 bit seed of entropy. Specifically what we do is to have $p \times (t+1)$ threads generate coefficients consisting of 160 bits of pseudorandomness by hashing a seed along with a unique identifier. Using these coefficients we have $p \times 2t$ threads computing $2t$ points on each of the $20p$ polynomials. More specifically $P_i(j)$ for all $j \in [2t]$ and $i \in [20p]$. This is done by a loop of $t+1$ iterations where an element $c_{i,j}j^d \in$ GF $\left(2^8\right)$ is computed in the d'th iteration and then added to the result of the previous $d-1$ iterations. 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 $t+1$ more threads each computing a single value of $c_{i,j}j^d$. Finally, these values could be added together using a reduction approach. However, since benchmarks showed that the construction of the polynomials only accounted for a very small part of the execution time, we choose to optimize other aspects of the protocols contributing more to the overall execution time.

Polynomial Interpolation. Our polynomial interpolation (part of the *Polynomial Setup*) is based on Lagrange interpolation and consists of 4 sub phases, *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 $t+1$ data point pairs $\{(x_j, y_j)\}_{j=1}^{t+1}$ where all x_j are distinct. These define the interpolation polynomials in Lagrange form:

$$L(x) := \sum_{j=1}^{t+1} y_j l_j(x) \ ,$$

where the Lagrange basis polynomials are defined as

$$l_j(x) := \prod_{1 \le g \le t+1, g \ne j} \frac{x - x_g}{x_j - x_g} \ ,$$

where $1 \le j \le t+1$. With this construction an arbitrary point x can then be computed. In our case we will know $t+1$ points from $[2t]$ and want to compute the remaining $t-1$ points in $[2t]$.

When we verify the $2t$ points Bob got from Alice for each of the $\lfloor 1.18s + 2.18 \rfloor$ check polynomials constitutes polynomials with degree at most $t+1$ we do polynomial interpolation of the points $]t+1; 2t]$

[8] 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 shifts and if statements. Furthermore, as an artifact of the implementation framework we do not do 20 polynomials in a SIMD manner, but instead sequentially.

using the points $[1; t+1]$ and verify that the newly interpolated points are the same Alice sent us. Now the phases for which we interpolate these points in a SIMD manner goes as follows:

Denominator: We have $t+1$ 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 $(t+1) \times (t-1)$ threads to compute the numerator of each $l_j(i)$. That is $l_j(i)$ will be computed by thread $j \cdot (t-1) + i$ so that we compute the numerator of each of the $t+1$ Lagrange polynomials using a different value for $i \in]t+1; 2t]$. Finally, each thread multiply its result with the appropriate denominator from the previous step. That is, computing $l_j(i)$ for each $i \in]t+1; 2t]$.

Combination: We use $\lfloor 1.18s + 2.18 \rfloor \times (t+1) \times (t-1)$ threads to compute each term of the $t+1$ terms of $L(i)$ for the $20\lfloor 1.18s + 2.18 \rfloor$ polynomials for each $i \in]t+1; 2t]$. This is simply done by having each thread multiply the appropriate values y_j for each for the $20\lfloor 1.18s + 2.18 \rfloor$ polynomials with the corresponding Lagrange basis polynomial (computed in the "Numerator" step).

Reduction: To continue with the maximal level of SIMD parallelization up to $\lfloor 1.18s + 2.18 \rfloor \times (t+1) \times (t-1)$ 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 $\lfloor 1.18s + 2.18 \rfloor \times (t-1)$ threads remain, containing the result $L(i)$ for all $i \in]t+1; 2t]$ and all $20\lfloor 1.18s + 2.18 \rfloor$ polynomials

All the steps are done with all generated data staying in the GPU's RAM and only at the end the final results are copied back to the host's RAM. Because of this it is sensible to do the reduction step instead 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 $t+1$ 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. This follows since Bob will always learn the same t points for all the polynomials as an effect of the cut-and-choose of garbled circuits. This means that we can implement the *numerator* step in the following two substeps:

- First we use $t \times (t-1)$ threads to compute $\prod_{1 \leq g \leq t, g \neq j} (x - x_g)$ for each point $j \in [t]$ where x_j is an index of a check circuit and for each of the $t-1$ points, x being an index of an evaluation circuit.
- We then use $\lceil 4.82s + 4.82 \rceil \times (t+1) \times (t-1)$ 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 $t+1$ Lagrange basis polynomials for each of the $t-1$ points Bob needs to learn, i.e. the $t-1$ indices of the evaluation circuits.

The same idea applies to the computation of the *denominator* step, but with a factor $t-1$ less, as the denominator remains the same for all possible values of x we need to learn.

With the above approach we avoid having $\lceil 4.82s + 4.82 \rceil \times (t+1) \times (t-1)$ threads doing a multiplication loop of $t+1$ iterations and instead only use $t \times (t-1)$ threads with a loop of t iterations and $\lceil 4.82s + 4.82 \rceil \times (t+1) \times (t-1)$ threads with a constant amount of multiplications to achieve the same result.

4.3 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. This is in particular true when some data needs to be sent/received while other operations can be carried out with the data already known.

4.4 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 garbling of a single gate along with the parallel complexity of the handling of selective failure attacks, consistency of inputs, OTs and commitments. In our case, as well as in [FN13], the

complexity of handeling selective failure attacks is bounded by $O(\log(\max(n, s)))$ as it only consists of xor'ing (because of the free-xor approach) at most $\max(4n, 8s)$ keys of κ bits with each other, $O(sn)$ times in parallel. This can be done using a reduction approach, assuming access to $O(sn\kappa \max(n, s))$ SIMD processors. The same argumentation goes for ensuring consistency of Alice's input, i.e. computing $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$: Using a reduction approach it is basically $O(\log(m))$ SIMD xor operations using $O(s^2\kappa m)$ SIMD processors. Regarding handeling 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 $O(\log(o))$ SIMD xor operations using $O(s^2 o\kappa)$ 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 has 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 $O(s^3\kappa)$ SIMD processors. Regarding polynomial interpolation, assuming we do this by 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. This means that the interpolation complexity is also limited to $O(\log(s))$ SIMD operations using $O(s^3\kappa)$ SIMD processors.

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 4.2.

4.5 Experimental Results

All of our experiments are based on the same, commonly used, circuit for oblivious 128 bit AES encryption.[9] This circuit is used as benchmark in [HEKM11, LP11, NNOB12, HKS+10, FN13], and many more implementations of 2PC for functions expressed as a Boolean circuit. We ran experiments on this 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 of Linux Mint 14 and CUDA 5.5. Each of the experiments was repeated 50 times and with no front end applications running on either of the machines. The timings are summarized in Table 1 and Table 2. These timings include loading circuit description and randomness along with communication between the host and device and communication between the parties.[10] However, in the same manner as done in [NNOB12] and [FN13] the timings of the seed OTs have not been included as this 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 this 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

[9] We thank Benny Pinkas, Thomas Schneider, Nigel P. Smart and Stephen C. Williams for supplying the circuit.

[10] In the tests the entropy used was sampled from `/dev/urandom`, which is a non-blocking source of pseudorandomness. This source was used in order to avoid high variance in the execution time caused by waiting for "true" randomness from `/dev/random`, which is a blocking source of randomness.

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 1. Timing of our protocol computing oblivious AES-128 for both Alice (A.) and Bob (B.) under different statistical security parameters (s). Communication is on a gigabit LAN and all times are in milliseconds. Column "IO" represent wall-clock time spent on disk IO. Column "Comp." represents wall-clock time where at least one thread of a given party does computation. Column "Comm." represents wall-clock time where the protocol execution is single-threaded and it does actual sending and receiving of data. Column "Idle" represents wall-clock time where the protocol execution is single-threaded and waiting for the other party to start sending/receiving data. 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.

communication and idle columns is time where the party is completely idle or only doing communication.

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 2. Timing of the parts of our protocol responsible for cheating recovery, i.e. part of the execution responsible for making the forge-and-loose approach work. The timings are taken from the computation of oblivious AES-128 for both Alice (A.) and Bob (B.) under different statistical security parameters (s). Communication is on a gigabit LAN and all times are in milliseconds. Column "Comp." represents wall-clock time where at least one thread of a given party does computation. Column "Comm." represents wall-clock time where the protocol execution is single-threaded and does actual sending and receiving of data. Column "Idle" represents wall-clock time where the protocol execution is single-threaded and waiting for the other party to start sending/receiving data. Column "Party total" represents the total wall-clock time of protocol execution.

Data Analysis. From Table 1 we see that idle time takes up a significant portion of the total execution time of the protocol, i.e. between 23% and 44% for each party and up to 76% of the wall-clock time one party sits idle. Taking Table 2 into account, we see at most 24% of the overhead of recovery is idle time. The reason for this is probably the fact 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.

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 this.

Comparing the total times of Table 1 and Table 2 we see that input recovery constitutes from 10% up to 39% for of the execution time for Bob, whereas for Alice 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 Bob 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). This was even worse before we optimized this part of our implementation to limit the amount of cache-misses.

Table 3. Timing comparison of secure two party computation protocols evaluating oblivious 128-bit AES. d is the depth of the circuit to be computed. Notice that our implementation is run on the same hardware and using the same garbling scheme as [FN13].

	Security	s	Model	Rounds	Time (s)	Equipment
[HEKM11]	Semi honest	-	ROM	$O(1)$	0.20	Desktop
This work	Malicious	2^{-9}	ROM	$O(1)$	0.21	Desktop w. GPU
This work	Malicious	2^{-40}	ROM	$O(1)$	0.46	Desktop w. GPU
[FN13]	Malicious	2^{-39}	ROM	$O(1)$	1.1	Desktop w. GPU
This work	Malicious	2^{-60}	ROM	$O(1)$	0.58	Desktop w. GPU
This work	Malicious	2^{-80}	ROM	$O(1)$	0.81	Desktop w. GPU
[NNOB12]	Malicious	2^{-58}	ROM	$O(d)$	1.6	Desktop
[KSS12]	Malicious	2^{-80}	SM	$O(1)$	1.4	Cluster, 512 nodes
[FN13]	Malicious	2^{-79}	ROM	$O(1)$	2.6	Desktop w. GPU
[SS13]	Malicious	2^{-80}	SM	$O(1)$	40.6	Cluster, 8 nodes

5 Conclusion and Future Work

We have presented a new approach to the recent idea of "forge-and-loose" which does not need any number theoretic assumptions, avoids an auxiliary secure computation and parallelizes well in a SIMD manner. We have implemented a random oracle realizations of this protocol in CUDA to take advantage of its SIMD friendly structure. Our implementation shows that the protocol becomes up to a factor 3 faster than the most comparable implementation. However, this is only true for large statistical security parameters. The benchmark shows that this is not because of the extra computations associate with our forge-and-loose approach, but rather because of the asymmetry of the overall cut-and-choose of garbled circuits approach, which takes up a significantly larger percentage of the total execution time for the lower statistical security parameters.

As future work it would be interesting to look into ways of making the overhead, needed for recovering Alice's input, smaller. In particular for larger statistical security parameters. In relation to this, making implementations (based on the same garbling, OT, etc.) of the other protocols using the forge-and-loose approach would be interesting, in particular to see which forge-and-loose technique introduces the least amount of overhead in different contexts. Finally, as we see that a significant portion of the wall-clock execution time of the protocol, even when using multi-threading, is idle time it would be interesting to look into ways of making the general structure of cut-and-choose of garbled circuits more symmetric.[11]

5.1 Acknowledgement

The authors would like to thank Rasmus Lauritsen for useful discussions.

References

[App13] Benny Applebaum. Bootstrapping obfuscators via fast pseudorandom functions. *IACR Cryptology ePrint Archive*, 2013:699, 2013.

[11] This as already been done in [MR13, HKE13]. However, their protocols require both parties to execute the protocol both as circuit constructor *and* as circuit evaluator. Thus, even though their protocols will probably have less idle time, it does not seem to be helpful in making the wall-clock execution time smaller.

[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*, 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 Kenneth G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2011.

[Blu82] Manuel Blum. Coin flipping by telephone - a protocol for solving impossible problems. In *COMPCON*, pages 133–137. IEEE Computer Society, 1982.

[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, *ASIACRYPT (2)*, volume 8270 of *Lecture Notes in Computer Science*, pages 441–463. Springer, 2013.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptology*, 13(1):143–202, 2000.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE Computer Society, 2001.

[CFIK03] Ronald Cramer, Serge Fehr, Yuval Ishai, and Eyal Kushilevitz. Efficient multi-party computation over rings. In Eli Biham, editor, *EUROCRYPT*, volume 2656 of *Lecture Notes in Computer Science*, pages 596–613. Springer, 2003.

[CKKZ12] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. On the security of the "free-XOR" technique. In *TCC*, pages 39–53, 2012.

[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, *ESORICS*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.

[DZ13] Ivan Damgård and Sarah Zakarias. Constant-overhead secure computation of boolean circuits using preprocessing. In *TCC*, pages 621–641, 2013.

[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 Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 537–556. Springer, 2013.

[FN13] Tore Kasper Frederiksen and Jesper Buus Nielsen. Fast and maliciously secure two-party computation using the GPU. *IACR Cryptology ePrint Archive*, 2013:46, 2013.

[GMS08] Vipul Goyal, Payman Mohassel, and Adam Smith. Efficient two party and multi party computation against covert adversaries. In Nigel P. Smart, editor, *EUROCRYPT*, volume 4965 of *Lecture Notes in Computer Science*, pages 289–306. Springer, 2008.

[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 *USENIX Security Symposium*. 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*, pages 272–284. IEEE Computer Society, 2012.

[HKE13] Yan Huang, Jonathan Katz, and Dave Evans. Efficient secure two-party computation using symmetric cut-and-choose. *IACR Cryptology ePrint Archive*, 2013:81, 2013.

[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, *ACM Conference on Computer and Communications Security*, pages 451–462. ACM, 2010.

[HL10] C. Hazay and Y. Lindell. *Efficient Secure Two-party Protocols: Techniques and Constructions*. Information security and cryptography. Springer Berlin Heidelberg, 2010.

[HMQ04] Dennis Hofheinz and Jörn Müller-Quade. Universally composable commitments using random oracles. In Moni Naor, editor, *TCC*, volume 2951 of *Lecture Notes in Computer Science*, pages 58–76. Springer, 2004.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *STOC*, 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.

[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 *ICALP (2)*, pages 486–498, 2008.

[KSS12] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. Towards billion-gate secure computation with malicious adversaries. *IACR Cryptology ePrint Archive*, 2012:179, 2012.

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *ASIACRYPT*, 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. *IACR Cryptology ePrint Archive*, 2013:79, 2013.

[LP07] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In Moni Naor, editor, *EUROCRYPT*, volume 4515 of *Lecture Notes in Computer Science*, pages 52–78. Springer, 2007.

[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.

[LP11] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In Yuval Ishai, editor, *TCC*, volume 6597 of *Lecture Notes in Computer Science*, pages 329–346. Springer, 2011.

[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*, volume 3958 of *Lecture Notes in Computer Science*, pages 458–473. Springer, 2006.

[MNT90] Yishay Mansour, Noam Nisan, and Prasoon Tiwari. The computational complexity of universal hashing. In *Structure in Complexity Theory Conference*, page 90, 1990.

[MR13] Payman Mohassel and Ben Riva. Garbled circuits checking garbled circuits: More efficient and secure two-party computation. In Ran Canetti and Juan A. Garay, editors, *CRYPTO (2)*, volume 8043 of *Lecture Notes in Computer Science*, pages 36–53. Springer, 2013.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *CRYPTO*, pages 681–700, 2012.

[NO09] Jesper Buus Nielsen and Claudio Orlandi. LEGO for two-party secure computation. In Omer Reingold, editor, *TCC*, volume 5444 of *Lecture Notes in Computer Science*, pages 368–386. Springer, 2009.

[NPS99] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *ACM Conference on Electronic Commerce*, pages 129–139, 1999.

[PSSW09] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *ASIACRYPT*, volume 5912 of *Lecture Notes in Computer Science*, pages 250–267. Springer, 2009.

[PVW08] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In Wagner [Wag08], pages 554–571.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[sS11] shelat abhi and Chih-Hao Shen. Two-output secure computation with malicious adversaries. In *EUROCRYPT*, pages 386–405, 2011.

[SS13] Abhi Shelat and Chih-Hao Shen. Fast two-party secure computation with minimal assumptions. *IACR Cryptology ePrint Archive*, 2013:196, 2013.

[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 *FOCS*, pages 160–164. IEEE Computer Society, 1982.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167. IEEE Computer Society, 1986.

A Overview of Variables and Parameters

A list of variable names and their meaning is given in Table 4 and Table 5.

Symbol	Meaning
s	Statistical security parameter.
κ	Computational security parameter.
C	The plain description of the Boolean circuit to compute.
f	The function computed by C.
x	A bit string representing the constructor's (Alice's) input to the circuit.
y	A bit string representing the evaluator's (Bob's) input to the circuit.
z	The circuit output destined for the evaluator.

Table 4. Overview of the free parameters of the protocol along with their meaning.

Symbol	Meaning
ℓ	Circuit replication factor; defined as the smallest even integer satisfying $\ell - \frac{1}{2}\log(\ell) + \log\left(2\frac{\sqrt{2\pi}}{e^2}\right) \geq s$.
t	The amount of check circuits; $t = \ell/2$.
p	The number of polynomials; defined as $p \geq 6s + 7$.
$\|C\|$	Amount of non-XOR gates in C.
m	Amount of input bits to the circuit from the constructor, $m = \|x\|$.
n	Amount of input bits to the circuit from the evaluator, $n = \|y\|$.
o	Total amount of output bits from the non-augmented circuit, $o = \|z\|$.
$\bar{m}$	The length of the constructor's extended input; $\bar{m} = n + s$.
$\bar{n}$	The length of the evaluator's extended input; $\bar{n} = \max\{4n, 8s\}$.
ψ	The check set for cut-and-choose on polynomials.
π, ϕ	The check and evaluation sets, respectively, for cut-and-choose on circuits.
Δ_j	The global key for the j'th garbled circuit.
$P_i(j)$	The j'th values of the i'the polynomial in the field $\mathbb{F}_{2^\kappa}$.

Table 5. Overview of "internal" variables and parameters derived from the free parameters.

B Preliminaries

B.1 Garbled Circuits

Our protocol requires a free-xor garbling scheme. We now describe a garbling scheme that combines the state of the art optimizations for Yao garbled gates i.e., free-xor [KS08], permutation bits [NPS99] and garbled row-reduction [NPS99]. The scheme is secure in the ROM and is due to [PSSW09].

In this scheme one can garble a gate by doing 4 evaluations of SHA-1, and a garbled gate consists of only 3 ciphertexts (therefore saving on communication complexity). The evaluation of the gate can be done using a single SHA-1 evaluation.

When considering keys in the following we do so only for a single garbled circuit and thus eliminate the circuit index, i.e., the 0-key on wire i is called k_i^0 and the 1-key k_i^1. When it is unknown if a key represents a 0 or 1 bit we discard the superscript, i.e., we simply call the key k_i. We will sometimes also abuse notation and call the index of the respectively left, right and output key of a given gate for k_L, k_R and k_O respectively, adding the superscript 0 or 1 as appropriate.

- We have a (possibly randomized) algorithm $\mathsf{Yao}(g, k_L^0, k_R^0, \Delta, id)$ with g a description of a Boolean function, a left input *0-key* $k_L^0 \in \{0,1\}^\kappa$, a right input 0-key $k_R^0 \in \{0,1\}^\kappa$, a global difference $\Delta \in \{0,1\}^\kappa$ and a unique gate identifier id, outputs a garbled gate $\tilde{g}$ and a output 0-key $k_O^0 \in \{0,1\}^\kappa$.
- We have an algorithm $\mathsf{Eval}(\tilde{g}, k'_L, k'_R)$ that on input a garbled gate $\tilde{g}$, a left key $k'_L \in \{0,1\}^\kappa$ and a right key $k'_R \in \{0,1\}^\kappa$ outputs an output key $k'_O \in \{0,1\}^\kappa \cup \{\perp\}$.
- We define the *1-keys* k_L^1,k_R^1,k_O^1 s.t. $k_L^0 \oplus k_L^1 = k_R^0 \oplus k_R^1 = k_O^0 \oplus k_O^1 = \Delta$.

The idea is that a garbled gate $\tilde{g}$ has a 0- and a 1-key associated with each of its wires (left input, right input and output wire), and that these keys represent the bit values on those wires. E.g., if $\tilde{g}$ is a garbled AND gate generated as $(\tilde{g}, k_O^0) \leftarrow \mathsf{Yao}(\wedge, k_L^0, k_R^0, \Delta, id)$ then $\mathsf{Eval}(\tilde{g}, k_L^a, k_R^b)$ for any $a, b \in \{0,1\}$ should output $k_O^{a \wedge b}$.

Note that if Alice samples Δ and a 0-key, say k_L^0, at random and gives the key k_L^a to Bob then there is no way for Bob to infer the bit a from k_L^a. Furthermore, even if Bob learns a he cannot guess the key k_L^{1-a} with better probability than guessing Δ. For a garbling scheme to be secure we intuitively want that even if Bob learns $\tilde{g}$ and keys k_L^a and k_R^b for $a, b \in \{0,1\}$, and is able to evaluate $k_O^{g(a,b)} \leftarrow \mathsf{Eval}(\tilde{g}, k_L^a, k_R^b)$, then he cannot learn anything about k_L^{1-a}, k_R^{1-b} or $k_O^{1-g(a,b)}$, even of he knows a and/or b.

Thus Bob can evaluate the garbled gate $\tilde{g}$ without knowing anymore about the output than he can infer from his knowledge of a and b. Furthermore, Bob cannot evaluate the gate on any other inputs. Thus if Bob sends back $k_O^{g(a,b)}$ to Alice, Alice can learn $g(a,b)$ (as she knows k_O^0 and Δ) and be confident that this is the correct result.

Definition 1. *We say that* $(\mathsf{Yao}, \mathsf{Eval})$ *is a* Yao free-xor garbling scheme *if the following holds:*

Correctness: *Let* $(\tilde{g}, k_O^0) \leftarrow \mathsf{Yao}(g, k_L^0, k_R^0, \Delta, id)$ *where* k_L^0, k_R^0 *and* Δ *are random bit strings,* id *is a uniques gate identifier and* g *is a Boolean function such that* $g : \{0,1\} \times \{0,1\} \to \{0,1\}$ *then for all* $b_L, b_R \in \{0,1\}$ *we demand that* $Pr\left[\mathsf{Eval}(\tilde{g}, k_L^{b_L}, k_R^{b_R}) = k_O^{g(b_L, b_R)}\right] = 1$.

From a garbled gate scheme we can construct a garbled circuit scheme. Let C be a Boolean circuit with n input wires, m output wires and some binary XOR gates and c non-XOR binary gates. Associate a unique identifier W to each wire. First sample a uniformly random $\Delta \in \{0,1\}^\kappa$. Then for each input wire W, sample a uniformly random key $k_W^0 \in \{0,1\}^\kappa$. Then inductively, for all gates which has both input keys defined, proceed as follows: For an XOR-gate with input wires L and R and output wire O, let $k_O^0 = k_L^0 \oplus k_R^0$. To later evaluate the gate on k_L^a and k_R^b, compute $k_O^{a \oplus b} = k_L^a \oplus k_R^b$. For a non-XOR-gate with identifier id, input wires L and R and output wire O and truth table g, sample $(\tilde{g}_{id}, k_O^0) \leftarrow \mathsf{Yao}(g, k_L^0, k_R^0, \Delta, id)$. This will eventually associate a key k_W^0 to each wire. Denote the input keys as $k_{I,1}^0, \ldots, k_{I,n}^0$. Denote the output keys as $k_{O,1}^0, \ldots, k_{O,m}^0$. Let ID be the set of identifiers of non-XOR-gates. The output of the garbling is $\left(\left(\Delta, \left\{k_{I,i}^0\right\}_{i=0}^n, \left\{k_{O,i}^0\right\}_{i=0}^m\right), \{\tilde{g}_{id}\}_{id \in \mathrm{ID}}\right)$. We use $\tilde{C}$ to denote $\{\tilde{g}_{id}\}_{id \in \mathrm{ID}}$, and we call this the garbled circuit. We use $\left(\left(\Delta, \left\{k_{I,i}^0\right\}_{i=0}^n, \left\{k_{O,i}^0\right\}_{i=0}^m\right), \tilde{C}\right) \leftarrow \mathsf{Yao}(C)$ to denote a random garbling. Let $x \in \{0,1\}^n$. To evaluate $\tilde{C}$ on $\left\{k_{I,i}^{x_i}\right\}_{i=0}^n$, inductively compute $k_W^{x_W}$ for each wire, where x_W is the value that wire W would have in $C(x)$. This is possible because of the correctness of the gargled gates and the fact that the XOR evaluation described above is correct too. Let $y = C(x)$. The output of the blind evaluation is $\left\{k_{O,i}^{y_i}\right\}_{i=0}^m = \mathsf{Eval}\left(\tilde{C}, \left\{k_{I,i}^{x_i}\right\}_{i=0}^n\right)$. The evaluation is deterministic if the evaluation of each garbled gate is deterministic, which we will assume for simplicity.

Definition 2. *We say that* $(\mathsf{Yao}, \mathsf{Eval})$ *is a* Yao free-xor gate garbling scheme *if the following holds:*

Correctness: *Let* $\left(\left(\Delta, \left\{k_{I,i}^0\right\}_{i=0}^n, \left\{k_{O,i}^0\right\}_{i=0}^m\right), \tilde{C}\right) \leftarrow \mathsf{Yao}(C)$. *Then for all* $x \in \{0,1\}^n$ *and* $y = C(x)$ *it holds that* $\mathsf{Eval}\left(\tilde{C}, \left\{k_{I,i}^{x_i}\right\}_{i=0}^n\right) = \left\{k_{O,i}^{y_i}\right\}_{i=0}^m$.

Secrecy: *For a value* $\Delta \in \{0,1\}^\kappa$*, let* $\mathcal{O}_\Delta(\cdot)$ *be an oracle which on input* Q *outputs a bit* b *where* $b = 1$ *if and only if (iff)* $Q = \Delta$*. Consider the following indistinguishability under chosen input attack game for a stateful adversary* $\mathcal{A}$*.*

Notation, Convention:

Let KDF be a secure key derivation function, e.g., as in [PSSW09]:

$$\mathsf{KDF}(k_L, k_R, id) = \mathrm{H}(k_L \| k_R \| id) \ .$$

Furthermore, call the least significant bit of every key the permutation bit and define a function $\mathrm{lsb} : \{0,1\}^\kappa \to \{0,1\}$ which extracts exactly this bit. We write this as $p_K = \mathrm{lsb}(K)$ and we assume that $\mathrm{lsb}(\Delta) = 1$ – see discussion at the end of this section.

Garbling, $\mathsf{Yao}\left(g, k_L^0, k_R^0, \Delta, id\right)$**:**

With $k_L^0, k_R^0, \Delta \in \{0,1\}^\kappa$ under the constraint that $\mathrm{lsb}(\Delta) = 1$ and id a unique identifier do the following:

1. Define

$$\begin{aligned} \tau^0 &= g(p_L, p_R) \\ \tau^1 &= g(p_L, 1-p_R) \\ \tau^2 &= g(1-p_L, p_R) \\ \tau^3 &= g(1-p_L, 1-p_R) \end{aligned}$$

2. Compute $k_O^0 = \mathsf{KDF}\left(k_L^{p_L}, k_R^{p_R}, id\right) \oplus (\tau^0 \cdot \Delta)$;
3. Compute:

$$\begin{aligned} \alpha_1 &= k_O^{\tau^1} \oplus \mathsf{KDF}\left(k_L^{p_L}, k_R^{\overline{p_R}}, id\right) \\ \alpha_2 &= k_O^{\tau^2} \oplus \mathsf{KDF}\left(k_L^{\overline{p_L}}, k_R^{p_R}, id\right) \\ \alpha_3 &= k_O^{\tau^3} \oplus \mathsf{KDF}\left(k_L^{\overline{p_L}}, k_R^{\overline{p_R}}, id\right) \end{aligned}$$

4. Output a garbled gate $\tilde{g} = (id, \alpha_1, \alpha_2, \alpha_3)$ and the zero output key k_O^0.

Evaluation, $\mathsf{Eval}\left(\tilde{g}, k_L', k_R'\right)$**:**

With $k_L', k_R' \in \{0,1\}^\kappa$ do the following:

1. Parse $\tilde{g} = (id, \alpha_1, \alpha_2, \alpha_3)$. Define $\alpha_0 = 0^\kappa$.
2. Compute $p_{L'} = \mathrm{lsb}(L')$ and $p_{R'} = \mathrm{lsb}(R')$ and let $j = 2p_{L'} + p_{R'}$.
3. Return $k_O' = \alpha_j \oplus \mathsf{KDF}(k_L', k_R', id)$.

Figure 4. Free-xor Garbled Gates Construction

$\mathsf{indcia}^{\mathcal{A}}_{(\mathsf{Yao},\mathsf{Eval})}(\kappa)$

$C \leftarrow \mathcal{A}(1^\kappa)$, *where* $C : \{0,1\}^n \to \{0,1\}^m$ *is a circuit as above.*

$(x^0, x^1) \leftarrow \mathcal{A}(1^\kappa)$, *where* $x^0, x^1 \in \{0,1\}^n$.

$\left(\left(\Delta, \left\{k_{I,i}^0\right\}_{i=0}^n, \left\{k_{O,i}^0\right\}_{i=0}^m\right), \tilde{C}\right) \leftarrow \mathsf{Yao}(C)$.

$c \leftarrow \{0,1\}$.

$b \leftarrow \mathcal{A}^{\mathcal{O}_\Delta(\cdot)}\left(\tilde{C}, \left\{k_{I,i}^{x_i^c}\right\}_{i=0}^n\right)$.

If $\mathcal{O}_\Delta(\cdot)$ *ever returned* 1*, then output* 0.

Otherwise, output $b \oplus c$.

We say that the scheme is IND-CIA *if for all PPT* $\mathcal{A}$*s, it holds that* $\left|\mathsf{Pr}\left(\mathsf{indcia}^{\mathcal{A}}_{(\mathsf{Yao},\mathsf{Eval})}(\kappa) = 0\right) - \frac{1}{2}\right| = \mathrm{negl}(\kappa)$.

The definition of secrecy captures two requirements at the same time: that one cannot distinguish keys representing different inputs, even given a garbled circuit, and that one cannot compute Δ even given a garbled circuit and one set of input keys. Assume namely that $\mathcal{A}$ could distinguish with advantage ε. Then he could output $c = b$ with probability $\frac{1}{2} + \varepsilon$, and then $b \oplus c = 0$ with probability $\frac{1}{2} + \varepsilon$, and hence $\left|\mathsf{Pr}\left(\mathsf{indcia}^{\mathcal{A}}_{(\mathsf{Yao},\mathsf{Eval})}(\kappa) = 0\right) - \frac{1}{2}\right| \geq \varepsilon$. Assume on the other hand that $\mathcal{A}$ could compute Δ with probability ε. Then he can input Δ to the oracle when he has it. Furthermore, he always outputs a uniformly random b. Then $\left|\mathsf{Pr}\left(\mathsf{indcia}^{\mathcal{A}}_{(\mathsf{Yao},\mathsf{Eval})}(\kappa) = 0\right) - \frac{1}{2}\right| = \varepsilon$.

In Fig. 4, a free-xor garbled gates construction, with optimizations, is presented. We briefly sketch its properties: for correctness, remember that the requirement for correctness is: Let $(\tilde{g}, k_O^0) \leftarrow \mathsf{Yao}(g, k_L^0,$

$k_R^0, \Delta, id)$, then for all $a, b \in \{0,1\}$ we have

$$\mathsf{Pr}\left[\mathsf{Eval}\left(\tilde{g}, k_L^a, k_R^b\right) = k_O^{g(a,b)}\right] = 1\ .$$

This is the case because, if we let $k'_L = k_L^a, k'_R = k_R^b$, then $\mathrm{lsb}(k'_L) = a \oplus p_L$ and $\mathrm{lsb}(k'_R) = b \oplus p_R$. Then by construction $\tau^j = g(a,b)$ for $j = 2p_L + p_R$ and

$$\alpha_j = k_O^{\tau^j} \oplus \mathsf{KDF}(k_L^a, k_R^b, id)$$

and therefore

$$k'_O = k_O^{\tau^j} = k_O^{g((p_L \oplus a),(p_R \oplus b))}\ .$$

This garbling scheme can be proven secure under the assumption that $\mathsf{KDF}(\cdot)$ behaves like a random function. Note that, as with any other free-xor based construction, we need to assume that the hash function we use is circular 2-correlation robust [CKKZ12].

An output decryption table for a pair of keys, (k_w^0, k_w^1) on wire w can for example, assuming the ROM, be implemented as the pair $\left(\mathrm{H}(k_w^0), \mathrm{H}(k_w^1)\right)$. In particular one can view it as a verifiable commitment to the 0-, respectively 1-key, in that particular order.

B.2 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 e bits (we use $e = 8$ in the implementation). With this in hand we describe the simple polynomial interpolation methods we will use in our protocol in Fig. 5.

Initialization

Initialized with the same parameters (t, ℓ, κ, e) from both Alice and Bob, with $2^e > \ell$ and e dividing κ, define the following polynomial manipulation methods. In the description we assume for notational convenience that $e = \kappa$. If $\kappa < e$, the values will be secret shared, interpolated and checked in κ/e blocks.

Construct

1. Alice construct a random polynomial in GF (2^e) of degree t by choosing $t+1$ random values in $\{0,1\}^e$ which will be the coefficients. Call this polynomial P.
2. Using P compute and return the following ℓ pairs $(i, P(i))_{i=1}^{\ell}$.

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

1. If ψ is not a subset of $[\ell]$ with $|\psi| \geq t+1$ then return `abort`. Otherwise take the $t+1$ 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$(\psi, (i, P(i))_{i\in\psi})$

1. If ψ is not a subset of $[\ell]$ of at least $t+1$ entries then output `abort`. Otherwise take the $t+1$ 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 5. The Methods for Polynomial Manipulation

B.3 Links

We use a technical tool called *links*. A *link scheme* is a tuple (G, F) of PPT algorithms. The *link generator* G takes as input two values $x, y \in \{0,1\}^\kappa$, 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 $x, y \in \{0,1\}^\kappa$ and $L \leftarrow G(x,y)$ it holds with probability 1 that $F(\texttt{forwards}, L, x) = y$ and $F(\texttt{backwards}, L, y) = x$.

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 x nor y, the link should not reveal any information on x not y. We will model this by requiring that an adversary who cannot compute x nor y cannot distinguish a link of x and y from a link of any two other values.

Definition 3. *We call a distribution* $(x_1, y_1, x_2, y_2, \ldots, x_\ell, y_\ell, z) \leftarrow D$ *on* $(\{0,1\}^\kappa)^{2\ell} \times \{0,1\}^*$ *hard if it holds for all PPT* B *that it wins the following game with negligible probability: First sample* $(x_1, y_1, x_2, y_2, \ldots, x_\ell, y_\ell, z) \leftarrow D$ *and give* z *to* B*. Run* B *to produce* $X \subset \{0,1\}^*$*, where the set is represented by writing down the elements of the set in a list. Then* B *wins iff* $x_i \in X$ *or* $y_i \in X$ *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* A *and all distributions* $D \in \mathcal{D}$ *that* A *wins the following game with probability negligibly close to* $\frac{1}{2}$*: Sample* $(x_1, y_1, x_2, y_2, \ldots, x_\ell, y_\ell, z) \leftarrow D$ *and give* z *to* A*. Sample uniformly random* $(x'_1, y'_1, x'_2, y'_2, \ldots, x'_\ell, y'_\ell, z) \in (\{0,1\}^\kappa)^{2\ell}$*. Sample a uniformly random bit* $b \in \{0,1\}$*. If* $b = 0$*, let* $(x''_1, y''_1, x''_2, y''_2, \ldots, x''_\ell, y''_\ell) = (x_1, y_1, x_2, y_2, \ldots, x_\ell, y_\ell)$*. Otherwise, let* $(x''_1, y''_1, x''_2, y''_2, \ldots, x''_\ell, y''_\ell) = (x'_1, y'_1, x'_2, y'_2, \ldots, x'_\ell, y'_\ell)$*. For* $i = 1, \ldots, \ell$*, sample* $L_i \leftarrow G(1^\kappa, x''_i, y''_i)$*. Input* $(L_1, \ldots, L_\ell)$ *to* A*. Run* A *to produce a guess* g*. The adversary wins iff* $g = b$*.*

We can build a link scheme from a hash function $\mathrm{H} : (\{0,1\}^\kappa)^2 \to \{0,1\}^\kappa$ as follows: On input $G(x, y)$, sample uniformly random salts $r, s \in \{0,1\}^\kappa$. Then output $L = (r, s, h, g) = (r, s, \mathrm{H}(x, r) \oplus y, \mathrm{H}(y, s) \oplus x)$. On input $F(\texttt{forwards}, (r, s, h, g), x)$, output $\mathrm{H}(x, r) \oplus h = y$. Similarly for the other direction, i.e. $F(\texttt{backwards}, (r, s, h, g), y) = \mathrm{H}(y, s) \oplus h = y$.

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 A queries some $\mathrm{H}(\cdot, r_i)$ or $\mathrm{H}(\cdot, s_i)$, the value $(r_i, s_i, \mathrm{H}(x_i, r_i) \oplus y_i, \mathrm{H}(y_i, s_i) \oplus x_i)$ is uniformly random in its view. So, turn it into an adversary B by giving it a uniformly random values (r_i, s_i, g_i, h_i) as input, and then add each x_i queried to some $\mathrm{H}(\cdot, r_i)$ or $\mathrm{H}(\cdot, s_i)$ to the set X, and then at the end output X.

It appears, however, to be a very strong assumption to assume that the scheme is secure for all hard distributions in the plain model. Assume namely that one could black box obfuscate a function $f_{x,y}$ which takes as input a function F and a value L and then outputs 1 if $F(\texttt{forwards}, L, x) = y$ and $F(\texttt{backwards}, L, y) = x$. Let the obfuscation of $f_{x,y}$ be called $O_{x,y}$. Then sample (x, y) uniformly at random and output $(x, y, O_{x,y})$. This is a hard distribution, as it is hard to find x or y from just $O_{x,y}$. However, given $L = G(x, y)$ for a link scheme (G, F) one can run $O_{x,y}$ on F and L and will get back 1. Running it on a random link will return 0 except with negligible probability. Hence the link scheme is not secure when the side information $O_{x,y}$ is given. So, assuming that the link scheme is secure is also an assumption that $f = f_{x,y}$ cannot be black-box obfuscated. This is certainly a strong assumption. Note, however, that this might not be too unreasonable an assumption: According to the recent result [App13], black-box obfuscating any complexity class which allows to implement a pseudo-random function would allow to black-box obfuscate all poly-sized circuits, which is impossible. It appears hard to black-box obfuscate a sufficiently strong hash function and not being able to obfuscate a pseudo-random function. It also seems hard to imagine that one can obfuscate f but not obfuscate the function H on which it is based. Hence it might just be impossible to obfuscate f. However, this is a rather fragile argument, so we would like to not rely on this faint hope.

Another way around the problem of an obfuscator as side information is to require that there is high min-entropy about either x or y given z. In that case z cannot contain an obfuscation containing x and y, like in the argument above. Indeed, it seems a reasonable assumption that if each x_i or y_i hash high min-entropy in the view of the adversary, then the above construction is a secure link scheme in the standard model for a sufficiently strong hash function H. Assume, e.g., that x_i has high min-entropy in the view of the adversary. Then it is reasonable to assume that $r_i, \mathrm{H}(x_i, r_i)$ acts as a strong extractor and that $r_i, \mathrm{H}(x_i, r_i,) \oplus y_i$ hiddes all partial information about y_i to a computationally bounded adversary. In that case y_i is still hard to compute given z and $r_i, \mathrm{H}(x_i, r_i) \oplus y_i$, and then it is reasonable to assume that $s_i, \mathrm{H}(y_i, s_i)$ acts as an extractor of the computational entropy of y_i and produces a pseudo-random value $s_i, \mathrm{H}(y_i, s_i)$, and that $s_i, \mathrm{H}(y_i, s_i) \oplus x_i$ hides all partial information about x_i to a computationally bounded adversary.

We will call a distribution super-hard if for each pair (x_i, y_i) one of the two values x_i or y_i has min-entropy $\Theta(k)$ bits in the view of the adversary. We will assume that our link scheme is secure for all super-hard distributions.

C Some Sub-Functionalities and Their Implementations

In this appendix we describe the ideal functionality for secure two-party function evaluation that we realize with our protocol. 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.

C.1 Secure Two-Party Function Evaluation

In Fig. 6 the ideal functionality for secure function evaluation is presented.

Initialization
Initialized by a circuit $\mathcal{C}$ and a statistical security parameter s.

Input Alice
On input $(\texttt{input}, x)$ from Alice, where x is a possible first input to f and where no input was given by Alice before, store x, and output `input` to Bob.

Input Bob
On input $(\texttt{input}, y)$ from Bob, where y is a possible second input to f and where no input was given by Bob before, and at a point after `input` was output to Bob, compute $z = f(x, y)$ and returns this value to Bob.

Corruption
The adversary may input $(\texttt{corrupt}, \text{Alice})$ or $(\texttt{corrupt}, \text{Bob})$ to corrupt Alice or Bob respectively. Any corrupted party may then at any time input `abort` in which case `abort` is output to both Alice and Bob and the ideal functionality halts. If Alice is corrupted and does not abort, then Alice may input $(\texttt{cheat}, z')$. In that case the ideal functionality samples a bit $d \in \{0, 1\}$ with $d = 0$ with probability 2^{-s}. If $d = 0$, then output y to Alice and output z' to Bob. If $d = 1$, then output `cheating` to Alice and Bob.

Figure 6. The ideal functionality, $\mathcal{F}_{\mathsf{SFE}}$, for secure function evaluation for two parties. The box reflects the structure of garbling and that only Bob gets output.

C.2 Commitments

We also need a commitment scheme as defined in Fig. 7. In the ROM this can simply be initialized using a hash function. That is, one commits by sending a hash of the value concatenated with a random string (random string can even be omitted if the value is chosen randomly from a big domain). For details on universally composable commitments in the ROM see [HMQ04].

Initialization
Both Alice and Bob input $(\texttt{init})$.

Input
- Upon receiving the command $(\texttt{commit}, id, x)$ from either Alice or Bob, if id is a new unique identifier, then the functionality stores $(id, 0, x, \text{name})$ where $\text{name} =$ Alice if Alice is the party sending the command and $\text{name} =$ Bob otherwise. The functionality then outputs $(\texttt{commit}, id)$ to both parties.
- Upon receiving the command $(\texttt{vc}, id, x)$ from either Alice or Bob, if id is a new unique identifier, then the functionality stores $(id, 1, x, \text{name})$ where $\text{name} =$ Alice if Alice is the party sending the command and $\text{name} =$ Bob otherwise. The functionality then outputs $(\texttt{vc}, id)$ to both parties.

Output
Upon receiving $(\texttt{open}, id)$ from Alice if a commitment $(id, \cdot, \cdot, \text{Alice})$ is stored then the functionality outputs $(\texttt{open}, id, x)$ to Bob. Similarly upon receiving $(\texttt{open}, id)$ from Bob if a commitment $(id, \cdot, \cdot, \text{Bob})$ is stored then the functionality outputs $(\texttt{open}, id, x)$ to Alice.

Query
Upon receiving $(\texttt{query}, id, x)$ from any party the box outputs `accept` if a commitment $(id, 1, x, \cdot)$ is stored. Otherwise it returns `reject`.[a]

[a] This command is used to verify wether or not a commitment is of the value expected. That is, `vc` is used to make commitments without randomness.

Figure 7. The ideal functionality $\mathcal{F}_{\mathsf{COM}}$ for maliciously secure commitments

C.3 Single Choice Batch Oblivious Transfer

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

Like in the rest of the paper we will use ℓ to denote the number of circuits to garble (that is, the replication factor) and n to denote the number of bits in Bob's input while κ will denote the amount of bits in each key.

We here consider a functionality that allows $n\ell$ OTs to be carried out in one "batch". In addition, while the functionality does not make any restrictions on the keys Alice inputs except that they are of κ bits, it does force Bob 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 $\mathcal{F}_{\text{BOTCC}}^{\bar{n},\ell,\kappa}$ and it is outlined in Fig. 8. Note that $\mathcal{F}_{\text{BOTCC}}^{\bar{n},\ell,\kappa}$ should not be confused with the batch single choice cut-and-choose functionality used in [LP11] that also performs the cut-and-choose on the circuits.

Initialization
Initialized by (n, ℓ, κ)
Transfer
On input $\left(\texttt{transfer}, \{\{a_{i,j}^0, a_{i,j}^1\}_{i=1}^{\bar{n}}\}_{j=1}^{\ell}\right)$ from Alice where $a_{i,j}^0, a_{i,j}^1 \in \{0,1\}^\kappa$ for some non-negative integer constant κ and $\left(\texttt{transfer}, \{b_i\}_{i=1}^{\bar{n}}\right)$ from Bob where $b_i \in \{0,1\}$, the functionality outputs $\left(\texttt{transfer}, \{\{a_{i,j}^{b_i}\}_{i=1}^{\bar{n}}\}_{j=1}^{\ell}\right)$ to Bob.
Corruption
A corrupted player can input `abort` at any time, in which case $\perp$ is output to the honest player and the functionality no longer responds to input.

Figure 8. The ideal functionality $\mathcal{F}_{\text{BOTCC}}^{\bar{n},\ell,\kappa}$ (with ℓ circuits, $\bar{n}$ input bits from Bob and keys of κ bits).

Given access to a random oracle, Fig. 10 shows how to realize $\mathcal{F}_{\text{BOTCC}}^{\bar{n},\ell,\kappa}$ efficiently in the $\mathcal{F}_{\text{OT}}$-hybrid model in the presence of malicious and static adversaries. The ideal functionality of $\mathcal{F}_{\text{OT}}$ is shown in Fig. 9 and can for example be realized as described in [PVW08].[12]

Initialization
Both Alice and Bob input $(\texttt{init})$.
Transfer
- Upon receiving $(\mathsf{sid}, \mathsf{sender}, a^0, a^1)$ where $a^0, a^1 \in \{0,1\}^\kappa$ from the first party, if no triplet $(\mathsf{sid}, \mathsf{sender}, \cdot, \cdot)$ is stored, then store the triplet $(\mathsf{sid}, \mathsf{sender}, a^0, a^1)$.
- Upon receiving $(\mathsf{sid}, \mathsf{receiver}, b)$ where $b \in \{0,1\}$ from the other party, if a triplet $(\mathsf{sid}, \mathsf{sender}, \cdot, \cdot)$ is stored where none of the elements is the symbol $\perp$ then output (sid, a^b) to the second party and (sid) to the adversary, otherwise output nothing. Finally, update the triplet such that what is now stored is $(\mathsf{sid}, \mathsf{sender}, \perp, \perp)$.

Corruption
A corrupted player can input `abort` at any time, in which case $\perp$ is output to the honest player and the functionality no longer responds to input.

Figure 9. The ideal functionality $\mathcal{F}_{\text{OT}}$

Theorem 1. *Assuming the hash function $H(\cdot)$ has κ bits output and is a random oracle then the protocol in Fig. 10 UC-securely realizes the $\mathcal{F}_{\text{BOTCC}}^{\bar{n},\ell,\kappa}$ functionality in the $\mathcal{F}_{\text{OT}}$-hybrid model in the presence of static, malicious adversaries.*

If Bob has many bits of input to the functionality he wish to compute with Alice, i.e. if $\bar{n}$ is larger than cs for some constant c, then we can use an "OT-extension" to limit the amount of heavy cryptographic operations needed to realize the $\mathcal{F}_{\text{BOTCC}}^{\bar{n},\ell,\kappa}$ functionality. A very efficient way of doing this is described in [NNOB12].

[12] We could aways realize the functionality by $\bar{n}$ 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.

Input

Alice holds ℓ sets of $\bar{n}$ key pairs, $\{\{(a^0_{i,j}, a^1_{i,j})\}^{\bar{n}}_{i=1}\}^{\ell}_{j=1}$, with $a^0_{i,j}, a^1_{i,j} \in \{0,1\}^\kappa$. Bob holds a set of $\bar{n}$ choice bits $\{b_i\}^{\bar{n}}_{i=1}$, i.e., $b_i \in \{0,1\}$.

The Protocol

1. Alice and Bob initialize $\mathcal{F}_{\text{OT}}$ by calling $(\texttt{init})$.
2. For each $i \in [\bar{n}]$ Alice calls $(i, \mathsf{sender}, r^0_i, r^1_i)$ where $r^0_i, r^1_i \in \{0,1\}^\kappa$.
3. For each $i \in [\bar{n}]$ Bob calls $(i, \mathsf{receiver}, b_i)$ where $\{b_i\}^{\bar{n}}_{i=1}$ is the set of his choice bits.
4. Thus, for each $i \in [\bar{n}]$ Bob learns $(i, r^{b_i}_i)$ with $r^{b_i}_i \in \{0,1\}^\kappa$ from $\mathcal{F}_{\text{OT}}$.
5. Bob now uses a hash function with κ bits output (modeled as a random oracle) to extend each of the n κ-bit strings to ℓ strings of κ bits. He computes:
$$r'^{b_i}_{i,j} = \text{H}\left(r^{b_i}_i \| id_{i,j}\right) \in \{0,1\}^\kappa$$
for all $i \in [\bar{n}]$ and all $j \in [\ell]$ where $id_{i,j}$ is a unique identifier. Thus he ends up with a total of $\bar{n} \cdot \ell$ strings each of κ bits.
6. In a similar manner Alice uses the same hash function and IDs to extend each of her $2\bar{n}$ κ-bit inputs to ℓ strings of κ bits. She computes the following pairs for all $i \in [\bar{n}]$ and all $j \in [\ell]$:
$$\left(r'^0_{i,j}, r'^1_{i,j}\right) = \left(\text{H}\left(r^0_i \| id_{i,j}\right), \text{H}\left(r^1_i \| id_{i,j}\right)\right) \in \{0,1\}^\kappa \times \{0,1\}^\kappa \ .$$
Thus she ends up with a total of $2\bar{n} \cdot \ell$ strings each of κ bits; half of these are equal to the ones Bob ended up with in the previous step.
7. Alice then sends the differences needed to change these values to her original input set. That is, she computes and sends the following pairs to Bob:
$$\left(\delta^0_{i,j}, \delta^1_{i,j}\right) = \left(r'^0_{i,j} \oplus a^0_{i,j}, r'^1_{i,j} \oplus a^1_{i,j}\right) \ .$$
8. Bob uses these pairs to reconstruct the input keys in correspondence with his input set $\{b_i\}^n_{i=1}$. Thus he simply computes and outputs:
$$a^{b_i}_{i,j} = \delta^{b_i}_{i,j} \oplus r'^{b_i}_{i,j} \ .$$

Figure 10. A protocol efficiently realizing $\mathcal{F}^{\bar{n},\ell,\kappa}_{\text{BOTCC}}$ in the $\mathcal{F}_{\text{OT}}$-hybrid model with access to a random oracle.

C.4 Coin-tossing

In order to prove simulation security we need to have some of the random choices made in the protocol be done using a coin-tossing functionality. We define an ideal functionality for this in Fig. 11.

Initialization

Upon receiving inputs $(\texttt{init}, \kappa)$ from both Alice and Bob the functionality leaks κ to the adversary.

Output

The functionality outputs the same uniformly random bit string of κ bits to both Alice and Bob, i.e., $\rho \in \{0,1\}^\kappa$. If Alice is corrupted, then Alice gets to see ρ first, and if she then inputs `abort`, then output `abort` to Bob instead of ρ.

Figure 11. The ideal functionality $\mathcal{F}_{\text{CT}}$ for maliciously secure coin tossing of κ bits

The Protocol

1. Bob selects a random string $\rho_B \in_R \{0,1\}^\kappa$ and sends a hash digest of it to Alice.
2. Alice also chooses a random string, $\rho_A \in_R \{0,1\}^\kappa$ and sends this to Bob.
3. Bob then sends the string ρ_B to Alice, who computes a hash digest of it and verifies that it equal to the digest she received in the first step. Furthermore, if $|\rho_B| \neq \kappa$ Alice outputs `abort`.
4. Alice and Bob now defines $\rho = \rho_A \oplus \rho_B$ and both output ρ.

Figure 12. A protocol realizing $\mathcal{F}_{\text{CT}}$ in the ROM

UC-secure coin-tossing can be based UC-secure commitment using the protocol in [Blu82] or in the ROM as described in Fig. 12.

D The Full Protocol

The full protocol is described in the figures on the following pages and the proof is given afterwards in Appendix E.

We let Alice garble ℓ versions of a Boolean circuit C using the free-xor technique [KS08], we call these garbled circuits $\tilde{C}_j$ 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 $\tilde{C}_j$, we have that $k^1_{i,j} = k^0_{i,j} \oplus \Delta_j$ in correspondance with a free-xor garbling scheme.

Now, remember that Alice is supposed to input $x \| a \in \{0,1\}^{m+s}$ in the augmented circuit, we then say that the wires from 1 to m represent her input x and the wires from $m+1$ to $m+s$ represent her auxiliary random input. Next, the wires from $m+s+1 = \bar{m}+1$ to $\bar{m}+\bar{n}$ will represent Bob's input. The wires from $\bar{m}+\bar{n}+1$ to $\bar{m}+\bar{n}+|C|$ will represent the internal wires of the circuit, i.e., the output wires of the $|C|$ non-XOR gates and in turn the wires from $\bar{m}+\bar{n}+|C|-o+1$ to $\bar{m}+\bar{n}+|C|+o$ will represent the output z.

In the following we enumerate the output keys resulting from the augmentations from 1 to $\lceil 5.82s + 4.82 \rceil$. The first s will correspond to the digest on Alice's input and the last $\lceil 4.82s + 4.82 \rceil$ will correspond to the hash function applied to the output z. To distinguish these keys from keys used in the actual circuit we call these keys $k'_{i,j}$ for $i \in [\lceil 5.82s + 4.82 \rceil]$ and $j \in [\ell]$.

Setup

1. Let $\bar{n} = \max\{4n, 8s\}$ and let ℓ be the smallest even integer satisfying $\ell - \frac{1}{2}\log(\ell) + \log\left(\frac{2\sqrt{2\pi}}{e^2}\right) \geq s$. Let p be the smallest integer satisfying $p \geq 6s + 7$.
2. Alice and Bob initialize $\mathcal{F}_{\mathsf{COM}}$, $\mathcal{F}_{\text{BOTCC}}^{\bar{n},\ell,\kappa}$, and $\mathcal{F}_{\text{CT}}$.
3. Bob chooses $M^{\mathsf{Sec}} \in \{0,1\}^{n \times \bar{n}}$ and a random bit vector $\bar{y} \in \{0,1\}^{\bar{n}}$ under the constraint that $M^{\mathsf{Sec}} \cdot \bar{y} = y$ and sends M^{Sec} to Alice.
4. Based on M^{Sec} the parties agree on a Boolean circuit C computing the function $f'(x, \bar{y}) = f(x, y) = z$ where Alice gives $x \in \{0,1\}^m$ as input and Bob gives $y \in \{0,1\}^{\bar{n}}$ as input and Bob receives $z \in \{0,1\}^o$ as output.

Polynomial Setup

1. Alice randomly chooses p polynomials of degree at most $t = \ell/2$. 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 $(P_i(j))_{j=1}^{\ell}$ thus uniquely define the polynomial P_i of degree at most t.
2. Using $\mathcal{F}_{\mathsf{COM}}$ Alice commits to $P_i(j)$ for all $i \in [p]$ and $j \in [\ell]$. Formally, she calls $\mathcal{F}_{\mathsf{COM}}\left(\mathtt{vc}, \mathsf{ID}_{i,j}^{\mathsf{poly}}, P_i(j)\right)$ and Bob receives $\left(\mathtt{vc}, \mathsf{ID}_{i,j}^{\mathsf{poly}}\right)$ from $\mathcal{F}_{\mathsf{COM}}$ for all $i \in [p]$ and $j \in [\ell]$.
3. Bob 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 this challenge to Alice.
4. Alice decommits to all the points corresponding to the received challenge. That is, for each $i \in \psi$ and $j \in [\ell]$ she calls $\mathcal{F}_{\mathsf{COM}}\left(\mathtt{open}, \mathsf{ID}_{i,j}^{\mathsf{poly}}\right)$ and in turn Bob learns $P_i(j)$.
5. Bob now verifies that for each $i \in \psi$ the points $(P_i(j))_{j=1}^{\ell}$ define a polynomial of degree at most $t = \ell/2$ by computing **Verify**$(\psi, (i, P_i)_{i \in \psi})$. If any of these checks fail Bob outputs `abort` and halts. Otherwise define the set $\chi = [p] \backslash \psi$, which is then the indices of the remaining polynomials.

Oblivious Transfer

1. Alice chooses the ℓ random global differences for the garbling phase. Call these elements $\Delta_j \in_R \{0,1\}^{\kappa}$ for $j \in [\ell]$.
2. Alice chooses $\ell \cdot \bar{n}$ random κ-bit key pairs which will be the keys used for Bob's input in ℓ garbled circuits to be constructed later. These keys are constructed under the constraint that any pair of 0- and 1-keys xor'ed together from the j'th circuit will give Δ_j, i.e., $k_{i,j}^0 \oplus k_{i,j}^1 = \Delta_j$. We call these key pairs $\left(k_{m+s+i,j}^0, k_{m+s+i,j}^1\right)$ for $i \in [\bar{n}]$ and $j \in [\ell]$.
3. Alice then inputs to $\mathcal{F}_{\text{BOTCC}}^{\bar{n},\ell,\kappa}$ the set of pairs $\left\{\{k_{m+s+i,j}^0, k_{m+s+i,j}^1\}_{i=1}^{\bar{n}}\right\}_{j=1}^{\ell}$ and Bob gives $\{\bar{y}_i\}_{i=1}^{\bar{n}}$. Thus Bob learns the set $\left\{k_{m+s+i,j}^{\bar{y}_i}\right\}_{i \in [\bar{n}], j \in [\ell]}$.

Garbling

1. Alice then uses a free-xor garbling scheme to garble ℓ copies of the circuit C, denoted $\tilde{C}_1, \tilde{C}_2, \ldots, \tilde{C}_\ell$. For the global difference in the j'th circuit she uses the values Δ_j already chosen in the *Setup* phase. For Bob's augmented input $\bar{y}$ she uses the keys just constructed for the oblivious transfer. She also constructs m 0-keys for her own input bits. She furthermore constructs s "extra" input 0-keys (which will be used as her auxiliary) for each garbled circuit. Define these extra bits to be the vector a such that Alice's augmented input is $\bar{x} = x \| a \in \{0,1\}^{m+s}$ and similarly define $\bar{m} = |x\|a| = m + s$. Each garbled circuit includes for each gate of C the corresponding garbled computation table and for each output wire the corresponding output decryption table.
2. Alice now sends each of the garbled circuits to Bob.

Commitment

1. Now Alice uses $\mathcal{F}_{\mathsf{COM}}$ to verifiably commit to the concatenation of the 0- and 1-key on each of her input wires along with the s auxiliary input wires for each garbled circuit. She does so by calling $\mathcal{F}_{\mathsf{COM}}\left(\mathtt{vc}, \mathsf{ID}_{i,j}^{\mathsf{A-order}}, k_{i,j}^0 \| k_{i,j}^1\right)$ for all $i \in [\bar{m}]$ and $j \in [\ell]$.
2. She also commits to her augmented input by calling $\mathcal{F}_{\mathsf{COM}}\left(\mathtt{commit}, \mathsf{ID}_{i,j}^{\mathsf{A-in}}, k_{i,j}^{\bar{x}_i}\right)$ for all $i \in [\bar{m}]$ and $j \in [\ell]$.
3. Finally, she commits to the keys for Bob's input wires, that is, for all $i \in [\bar{n}]$ and $j \in [\ell]$ she calls $\mathcal{F}_{\mathsf{COM}}\left(\mathtt{commit}, \mathsf{ID}_{\bar{m}+i,j}^{\mathsf{B-order}}, k_{\bar{m}+i,j}^0 \| k_{\bar{m}+i,j}^1\right)$.

Figure 13. The Protocol – Part 1

Augmentation

1. Now Bob samples three random bit vectors $b^{\mathsf{In}} \in_R \{0,1\}^{\bar{m}-1}$, $b^{\mathsf{Out1}} \in_R \{0,1\}^{\lceil o+4.82s+3.82 \rceil}$ and $b^{\mathsf{Out2}} \in_R \{0,1\}^{\lceil 4.82s+4.82 \rceil}$. The first two vectors defines the matrices $M^{\mathsf{In}} \in \{0,1\}^{m \times s}$ and $M^{\mathsf{Out}} \in \{0,1\}^{\lceil 4.82s+4.82 \rceil}$ respectively as follows
$$M^{\mathsf{In}}_{i,j} = b^{\mathsf{In}}_{i+j} \ , \quad M^{\mathsf{Out}}_{i,j} = b^{\mathsf{Out1}}_{i+j} \ .$$
2. Bob sends the three vectors b^{In}, b^{Out1} and b^{Out2} to Alice who then defines the matrices M^{In} and M^{Out} similarly.
3. Alice augments the garbled circuits to compute $\left(M^{\mathsf{In}} \cdot x\right) \oplus a$ where x is Alice's binary inputs (in column form) and a is her auxiliary binary inputs (in column form). That is, she computes the keys
$$k'^0_{i,j} = \left(\bigoplus_{l=1}^{m} M^{\mathsf{In}}_{i,l} \cdot k^0_{l,j} \right) \oplus k^0_{m+i,j} \ ,$$
for all $i \in [s]$ and all circuits $j \in [\ell]$.
4. She uses these keys to construct output decryption tables for the augmentation based on M^{In}. She then sends these tables to Bob.
5. Next, Alice augments the garbled circuits further such that they compute $(M^{\mathsf{Out}} \cdot z) \oplus b^{\mathsf{Out2}}$ where z is the binary output (in column form). That is, she computes the keys
$$k'^0_{s+i,j} = \left(\bigoplus_{l=1}^{o} M^{\mathsf{Out}}_{i,l} \cdot k^0_{l+\bar{m}+\bar{n}+|C|-o,j} \right) \oplus \left(b^{\mathsf{Out2}}_i \cdot \Delta_j \right) \ ,$$
for all $i \in [\lceil 4.82s + 4.82 \rceil]$ and all circuits $j \in [\ell]$.
6. Alice then computes the links $L_{i,j} = G\left(P_{\chi[i]}(j), k'^0_{s+i,j}\right)$ for $i \in [\lceil 4.82s + 4.82 \rceil]$ and $j \in [\ell]$. Alice then sends these links to Bob.

Cut-and-choose

1. Alice and Bob invoke $\mathcal{F}_{\mathrm{CT}}$ to randomly select $t = \ell/2$ elements from $[\ell]$. Call these t 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 Alice then opens the commitments to all the input keys along with the polynomial points associated with the augmented output wires. That is, she executes the following commands:
$$\mathcal{F}_{\mathsf{COM}}\left(\mathtt{open}, \mathsf{ID}^{\mathsf{A-order}}_{i,j}\right), \mathcal{F}_{\mathsf{COM}}\left(\mathtt{open}, \mathsf{ID}^{\mathsf{B-order}}_{i,j}\right), \mathcal{F}_{\mathsf{COM}}\left(\mathtt{open}, \mathsf{ID}^{\mathsf{poly}}_{i,j}\right)$$
and thus Bob receives from $\mathcal{F}_{\mathsf{COM}}$ the values $k^0_{i,j} \| k^1_{i,j}$ for all $j \in \pi$ and $i \in [\bar{m} + \bar{n}]$ along with $P_{\chi[i]}(j)$ for all $j \in \pi$ and $i \in [\lceil 4.82s + 4.82 \rceil]$.
3. Bob uses the pairs of input keys to degarble the circuits $\tilde{C}_j$ and checks that they do in fact compute the function agreed upon in the *Setup* phase.
4. Bob also checks that the augmentation computing a digest on Alice's input has been constructed correctly by verifying its output decryption table in all the check circuits, i.e. the circuits which have indices in π. That is, he computes $\Delta_j = k'^0_{1,j} \oplus k'^1_{1,j}$ for all $j \in \pi$. He then computes
$$k'^0_{i,j} = \left(\bigoplus_{l=1}^{m} M^{\mathsf{In}}_{i,l} \cdot k^0_{l,j} \right) \oplus k^0_{m+i,j} \ , \quad k'^1_{i,j} = k'^0_{i,j} \oplus \Delta_j$$
and then uses these keys to check that the output decryption tables has been correctly constructed. If any checks fail Bob outputs `abort`.
5. In the same manner Bob then uses all the output keys from the garbled circuits to compute the augmented output. That is, he computes
$$k'^0_{s+i,j} = \left(\bigoplus_{l=1}^{o} M^{\mathsf{Out}}_{i,l} \cdot k^0_{l+\bar{m}+\bar{n}+|C|-o,j} \right) \oplus \left(b^{\mathsf{Out2}}_i \cdot \Delta_j \right) \ , \quad k'^1_{s+i,j} = k'^0_{s+i,j} \oplus \Delta_j$$
and uses these values to compute $P_{\chi[i]}(j)$ via the link $L_{i,j}$.That is he computes $F(\mathtt{forwards}, L_{i,j}, k'^0_{s+i,j}) = P_{\chi[i]}(j)$. He then verifies these values by calling $\mathcal{F}_{\mathsf{COM}}\left(\mathtt{check}, \mathsf{ID}^{\mathsf{poly}}_{\chi[i],j}, P_{\chi[i]}(j)\right)$ for all $i \in [\lceil 4.82s + 4.82 \rceil]$ and $j \in \pi$.
6. If any of the above checks fail or if $\mathcal{F}_{\mathsf{COM}}$ returns `reject` on any of the calls Bob outputs `abort` and terminates the protocol.

Figure 14. The Protocol – Part 2

Evaluation

1. Alice now opens her input keys by calling $\mathcal{F}_{\mathsf{COM}}\left(\texttt{open}, \mathsf{ID}_{i,j}^{\mathsf{A-order}}\right)$ for all $i \in [\bar{m}]$ and $j \in \phi$ from which Bob in turn learns $k_{i,j}^{\bar{x}_i}$.
2. Using Alice's input keys Bob evaluates the augmentation for consistency checking of Alice's input for all $j \in \phi$. Thus he computes
$$k'_{i,j} = \left(\bigoplus_{l=1}^{m} M_{i,l}^{\mathsf{In}} \cdot k_{l,j}^{0}\right) \oplus k_{m+i,j}^{0}$$
for all $i \in [s]$ and $j \in \phi$. He then uses the output decryption tables to find the bit each key $k'_{i,j}$ represent, he calls this bit $z'_{i,j}$. If there is a problem with any output decryption table he output `abort`. He then verifies that $z'_{i,j} = z'_{i,j'}$ for all $i \in [s]$ and $j, j' \in \phi$ and if any check fails he outputs `abort`.
3. Bob now uses Alice'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 and in turn learns the semantic value on each output wire. Each circuit where the evaluation fails (in practice by the output decryption table not decrypting to 0 or 1) Bob discards.
4. If all, non-discarded, garbled circuits evaluates to the same semantic value, then Bob accepts this value as output and outputs `ok` . If there are different outputs then Bob proceeds with the *Reconstruction* phase.[a]

Reconstruction

1. Define $z_{l,j} \in \{0,1\}$ to be the l'th output bit of the j'th garbled circuit, which Bob learns for all $l \in [o]$ and $j \in \phi$ when he evaluates the circuits (if Bob cannot evaluate or find the semantic value of the l'th output wire in the j'th circuit we say $z_{l,j} = \perp$). As he knows the matrix M^{Out} and b^{Out2} he can now compute the bits $z'_{s+i,j} = \left(\bigoplus_{l=1}^{o} M_{i,l}^{\mathsf{Out}} \cdot z_{l,j}\right) \oplus b^{\mathsf{Out2}}$ for all $i \in [\lceil 4.82s + 4.82 \rceil]$ and $j \in \phi$.
2. Similarly he computes the augmented output keys as $k_{s+i,j}^{\prime z'_{s+i,j}} = \left(\bigoplus_{l=1}^{o} M_{i,l}^{\mathsf{Out}} \cdot k_{l,j}^{z_{l,j}}\right)$ 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, Bob computes the values $P_{\chi[i]}(j)$ for all $i \in [\lceil 4.82s + 4.82 \rceil]$ and $j \in \phi$. That is, he uses link $L_{i,j}$ to compute $P_{\chi[i]}(j) = F(\texttt{forwards}, L_{i,j}, k_{s+i,j}^{\prime z'_{s+i,j}})$. for all $i \in [\lceil 4.82s + 4.82 \rceil]$ and $j \in \phi$ where $z'_{s+i,j} = 0$. For each of these values, he calls $\mathcal{F}_{\mathsf{COM}}\left(\texttt{check}, \mathsf{ID}_{\chi[i],j}^{\mathsf{poly}}, P_{\chi[i]}(j)\right)$. If $\mathcal{F}_{\mathsf{COM}}$ returns `reject` on any of these instances he discards the associated circuit j.
4. Bob now takes the remaining garbled circuits that evaluate to different values, and look at all the augmented output wires that evaluates 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 exist a pair of indicies $j, j' \in U$ where $z'_{s+i,j} \neq z'_{s+i,j'}$ and let $V = W \times U$ be the cartesian product of U and W. Now for each $i \in W$ do the following:[b]
   (a) Find the first element $j \in U$ for which it is true that $z'_{s+i,j} = 0$. Now using link $L_{i,j}$ he calls $\mathcal{F}_{\mathsf{COM}}\left(\texttt{check}, \mathsf{ID}_{i,j}^{\mathsf{poly}}, \mathrm{H}\left(k'_{s+i,j}, s_{i,j}\right) \oplus g_{i,j}\right)$. If $\mathcal{F}_{\mathsf{COM}}$ returns `reject` then take the next element (if it exist) $j \in U$ where $z'_{s+i,j} = 0$ and call $\mathcal{F}_{\mathsf{COM}}\left(\texttt{check}, \mathsf{ID}_{i,j}^{\mathsf{poly}}, \mathrm{H}\left(k'_{s+i,j}, s_{i,j}\right) \oplus g_{i,j}\right)$ and continue taking elements (if they exist) until $\mathcal{F}_{\mathsf{COM}}$ returns `accept`.
   (b) When this happens do polynomial interpolation using the set of pairs $S = \{\pi \cup j\} \times \left\{P_i(l)_{l \in \pi} \cup \mathrm{H}\left(k'_{s+i,j}, s_{i,j}\right) \oplus g_{i,j}\right\}$ 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 $\mathcal{F}_{\mathsf{COM}}\left(\texttt{check}, \mathsf{ID}_{i,l}^{\mathsf{poly}}, P_i(l)\right)$.
   (c) Next, for each element $j' \in U$ for which it is true that $z'_{s+i,j} = 1$ compute the values $\Delta_{j'} = \mathrm{H}\left(P_i(j'), r_{i,j}\right) \oplus h_{i,j} \oplus k'_{s+i,j'}$.
5. Now Bob should know Δ_j for all $j \in \phi$ where the garbled circuit has not bee discarded. If he does not, then he output `abort` and terminates.
6. Using Δ_j and Alice's input keys Bob can find Alice's input for all circuits calling $\mathcal{F}_{\mathsf{COM}}\left(\texttt{check}, \mathsf{ID}_{i,j}^{\mathsf{A-order}}, k_{i,j}^{x_i} \| k_{i,j}^{x_i} \oplus \Delta_j\right)$ and $\mathcal{F}_{\mathsf{COM}}\left(\texttt{check}, \mathsf{ID}_{i,j}^{\mathsf{A-order}}, k_{i,j}^{x_i} \oplus \Delta_j \| k_{i,j}^{x_i}\right)$, if the first call returns `accept`, Alice's bit $x_i = 0$, if the second call returns `accept` then $x_i = 1$, otherwise he outputs `abort` and terminates the protocol.
7. If Alice's input bits are the same in all the remaining garbled circuits Bob evaluates the function $f(x, y) = z$ unencrypted and finally outputs `ok` and uses z as his output of the computation. 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 Alice's input of any remaining circuit, to evaluate the function f in plain.[c]

[a] In order to avoid timing attacks Bob should always do the following computations for simulated values.

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

[c] Alice's input will be unique on each of the remaining garbled circuit unless there is a collision in $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$, which only happens with probability at most 2^{-s}.

Figure 15. The Protocol – Part 3

E Proof of Security

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

E.1 Universal Hash Functions

Definition 4 ((Binary) universal hash function). *Given 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 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 5 ($\mathbb{H}^{\mathsf{ln}}$ (formal)). *Let* $b \in \{0,1\}^{m+s-1}$ *and* $M^{\mathsf{ln}} \in \{0,1\}^s \times \{0,1\}^m$ *be a matrix with each entry defined as* $M^{\mathsf{ln}}_{i,j} = b_{i+j-1}$*. Now define the function* $h_b : x \to z$ *as* $h_b(x) = M^{\mathsf{ln}}x = z$ *when queried on* b*. Finally define a family of functions* $\mathbb{H}^{\mathsf{ln}} = h_b(\cdot)_{b \in \{0,1\}^{m+s-1}}$.

Lemma 1. $\mathbb{H}^{\mathsf{ln}}$ *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:

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

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

Now what we need to show is $\mathsf{Pr}[\left(M^{\mathsf{ln}}x\right) \oplus \left(M^{\mathsf{ln}}x'\right) = z] = 2^{-s}$ for any $z \in \{0,1\}^s$ when each entry of M^{ln} is defined as $M^{\mathsf{ln}}_{i,j} = b_{i+j-1}$, $b \in_R z^{m+s-1}$ and $x \neq x'$. First notice that matrix multiplication is linear, so these are equivalent:

$$\begin{aligned}\left(M^{\mathsf{ln}}x'\right) \oplus \left(M^{\mathsf{ln}}x\right) &= z \\ M^{\mathsf{ln}}(x \oplus x') &= z\end{aligned}$$

This means that it is enough to show that $\mathsf{Pr}[M^{\mathsf{ln}}(x \oplus x') = z] = 2^{-s}$ 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^{m-1} choices of b for each of the 2^s possible elements in the range of the function. Thus when b is chosen at random (from $\{0,1\}^{m+s-1}$) 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 b, $M^{\mathsf{ln}}\tilde{x}$ can be computed as follows:

$$M^{\mathsf{ln}}\tilde{x} = \begin{pmatrix} b_1 & b_2 & \dots & b_m \\ b_2 & b_3 & \dots & b_{m+1} \\ \vdots & \vdots & \ddots & \vdots \\ b_s & b_{s+1} & \dots & b_{m+s-1} \end{pmatrix} \cdot \begin{pmatrix} \tilde{x}_1 \\ \tilde{x}_2 \\ \vdots \\ \tilde{x}_m \end{pmatrix} \ .$$

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

$$\begin{pmatrix} b_1 & b_2 & \cdots & b_m \\ b_2 & b_3 & \cdots & b_{m+1} \\ \vdots & \vdots & \ddots & \vdots \\ b_s & b_{s+1} & \cdots & b_{m+s-1} \end{pmatrix} \cdot \begin{pmatrix} \tilde{x}_1 \\ \tilde{x}_2 \\ \vdots \\ \tilde{x}_m \end{pmatrix} = \begin{pmatrix} \tilde{x}_1 & \tilde{x}_2 & \tilde{x}_3 & \cdots & \tilde{x}_m & 0 & \cdots & 0 \\ 0 & \tilde{x}_1 & \tilde{x}_2 & \cdots & \tilde{x}_m & 0 & \cdots & 0 \\ 0 & 0 & \tilde{x}_1 & \cdots & \tilde{x}_m & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & \tilde{x}_1 & \cdots & \tilde{x}_m \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{m+s-1} \end{pmatrix} .$$

A bit more specific we view b as a column vector of $m + s - 1$ bits and define a binary matrix $X \in \{0,1\}^s \times \{0,1\}^{m+s-1}$ where

$$X_{i,j} = \begin{cases} \tilde{x}_{j-i+1} & \text{if} j - i + 1 \leq m \\ 0 & \text{otherwise} \end{cases}$$

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

$$\begin{pmatrix} \tilde{x}_1 & \cdots & 1 & \cdots & \tilde{x}_m & 0 & 0 & \cdots & 0 \\ 0 & \tilde{x}_1 & \cdots & 1 & \cdots & \tilde{x}_m & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \\ 0 & 0 & \cdots & 0 & \tilde{x}_1 & \cdots & 1 & \cdots & \tilde{x}_m \end{pmatrix} .$$

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

Next consider row i and column j. No matter what the value is of all the other $m + s - 2$ entries in row i are set to, the i'th bit in the output vector can be uniquely decided by setting b_i to either 0 or 1. That is, if $b_i = 0$ the $m + s - 2$ other entries in row i and the vector b computes the i'th output bit, if it is set to 1, the negation of this will be the i'th output bit. The point is that b_i can be used to hit either 0 or 1 no matter what the other bits in the matrix and the vector b 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 b for each row to determine each output bit we see that for a given output vector (of s bits) we need s bits of b to uniquely decide its value, whereas the rest of the $m + s - 1 - s = m - 1$ bits of b remain free no matter what output we desire and what the input is. Thus there will exist at least 2^{m-1} choices of b to give a specific output, in particular z on any non-zero choice of x. But since we have already established, 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^{m-1} choices of b for each output, and in particular z, as $2^s \cdot 2^{m-1} = 2^{m+s-1}$ covers all the possible choices of b. □

Definition 6 ($\mathbb{H}^{\mathsf{Out}}$ (formal)). *Let $b^{\mathsf{Out1}} \in \{0,1\}^{\lceil o+4.82s+3.82 \rceil}$, $b^{\mathsf{Out2}} \in \{0,1\}^{\lceil 4.82s+4.82 \rceil}$ and $M^{\mathsf{Out}} \in \{0,1\}^{\lceil 4.82s+4.82 \rceil} \times \{0,1\}^o$ be a matrix with each entry defined as $M^{\mathsf{Out}}_{i,j} = b^{\mathsf{Out1}}_{i+j-1}$. Now define the function*

$$h'_{b^{\mathsf{Out1}}, b^{\mathsf{Out2}}} : x \to z \text{ as } h'_{b^{\mathsf{Out1}}, b^{\mathsf{Out2}}}(x) = (M^{\mathsf{Out}} x) \oplus b^{\mathsf{Out2}} = z \ ,$$

when queried on $b^{\mathsf{Out1}}, b^{\mathsf{Out2}}$. Finally define a family of functions

$$\mathbb{H}^{\mathsf{Out}} = h'_{b^{\mathsf{Out1}}, b^{\mathsf{Out2}}}(\cdot)_{b^{\mathsf{Out1}} \in \{0,1\}^{\lceil 4.82s+4.82 \rceil}, b^{\mathsf{Out2}} \in \{0,1\}^{\lceil o+4.82s+3.82 \rceil}} \ .$$

Lemma 2. *$\mathbb{H}^{\mathsf{Out}}$ is a pairwise independent universal hash function.*

Proof. A proof can be found in [MNT90]. □

E.2 Proof of Protocol Security

We prove the protocol secure according to the standard real/ideal simulation paradigm with sequential composition [Can00,Gol04,HL10] under the assumption that our garbling scheme is IND-CIA and hybrid access to the $\mathcal{F}_{\text{BOTCC}}$, $\mathcal{F}_{\text{CT}}$ and $\mathcal{F}_{\mathsf{COM}}$ functionalities. In the end we argue how the proof can become secure in the UC model [Can01].

Theorem 2. *Let π be the protocol in Fig. 13 to Fig. 15 and assume that the garbling scheme $\mathcal{G} = (\mathsf{Yao}, \mathsf{Eval})$ used in π is IND-CIA secure according to Def. 1 and calls to $H(\cdot)$ are realized as calls to the random oracle. Then π securely realizes $\mathcal{F}_{\mathsf{SFE}}$ in Fig. 6 in the ($\mathcal{F}_{\text{BOTCC}}$, $\mathcal{F}_{\text{CT}}$, $\mathcal{F}_{\mathsf{COM}}$)-hybrid model with abort in the presence of static, malicious adversaries.*

Proof. We split the proof up in two lemmas, one considering a corrupted Alice and one where Bob is corrupted. The two lemmas are treated in Appendix E.3 and Appendix E.4, respectively. Theorem 2 follows directly from these two lemmas. □

E.3 Corrupt Alice

Lemma 3. *Let π be the protocol in Fig. 13 to Fig. 15. Assume that the garbling scheme $\mathcal{G} = (\mathsf{Yao}, \mathsf{Eval})$ used in π is IND-CIA secure according to Def. 1 and calls to $H(\cdot)$ in the are replaced with calls to a random oracle. Then π securely realizes $\mathcal{F}_{\mathsf{SFE}}$ in Fig. 6 in the ($\mathcal{F}_{\text{BOTCC}}$, $\mathcal{F}_{\text{CT}}$, $\mathcal{F}_{\mathsf{COM}}$)-hybrid model with abort in the presence of static, malicious adversary corrupting* Alice*.*

Proof. We do the proof in the hybrid model where a trusted party is used to compute $\mathcal{F}_{\text{BOTCC}}$, $\mathcal{F}_{\text{CT}}$ and $\mathcal{F}_{\mathsf{COM}}$, and where Yao and Eval are assumed to be IND-CIA secure according to Def. 1.

Before proceeding with the formal proof we notice that if Alice tries to cheat she can only do so before the cut-and-choose phase as there is no interaction (except opening of commitments) with her after this phase.

A common strategy [LP07, LP11, FN13, sS11] is to take the result to be the *majority* of the results of the evaluation circuits. It can be shown that [LP11] selecting half of the circuits as evaluation circuits, the majority of the evaluation circuits are correct except with probability at most 2^{-s} if $\ell = 3.22s$ [LP11]. It was later shown [sS11] that with the majority approach, selecting 3/5 of the circuits as check circuits allows for a replication factor of $\ell = 3.13s$.

However, in our approach we make sure that two evaluation circuits evaluating to different results enables Bob to recover Alice's plain input for these circuits. Hence, intuitively, a cheating Alice must guess *exactly* the cut-and-choose challenge. In this case, where only a single evaluation circuit needs to be correct, the optimal ratio between check and evaluation circuits is 1/2. This is seen as a cheating Alice's success probability is $\binom{s}{cs}^{-1}$ when $c \leq 1$. This number is clearly minimized when $c = 1/2$ since $\binom{s}{cs}$ 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 Alice guesses exactly the cut-and-choose challenge with probability at most 2^{-s}.

Lemma 4. *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. See Appendix F. □

Choosing the replication factor according to Lemma 4, the only problem is therefore that Bob might not know which circuit is the correct one. Thus, if more than one circuit "seemingly" evaluate correctly Bob must be able to discover which of these evaluations are correct. He does that by restoring Alice'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 Alice in the real world execution of protocol π. We construct a simulator $\mathcal{S}$ running in the ideal world with external access to $\mathcal{F}_{\mathsf{SFE}}$. Internally $\mathcal{S}$ runs $\mathcal{A}$, controlling Alice, along with a simulation of the interaction between $\mathcal{A}$, an honest Bob and the ideal functionalities ($\mathcal{F}_{\text{BOTCC}}$, $\mathcal{F}_{\text{CT}}$, $\mathcal{F}_{\mathsf{COM}}$). This is illustrated in Fig. 16. The simulator $\mathcal{S}$ works as follows:

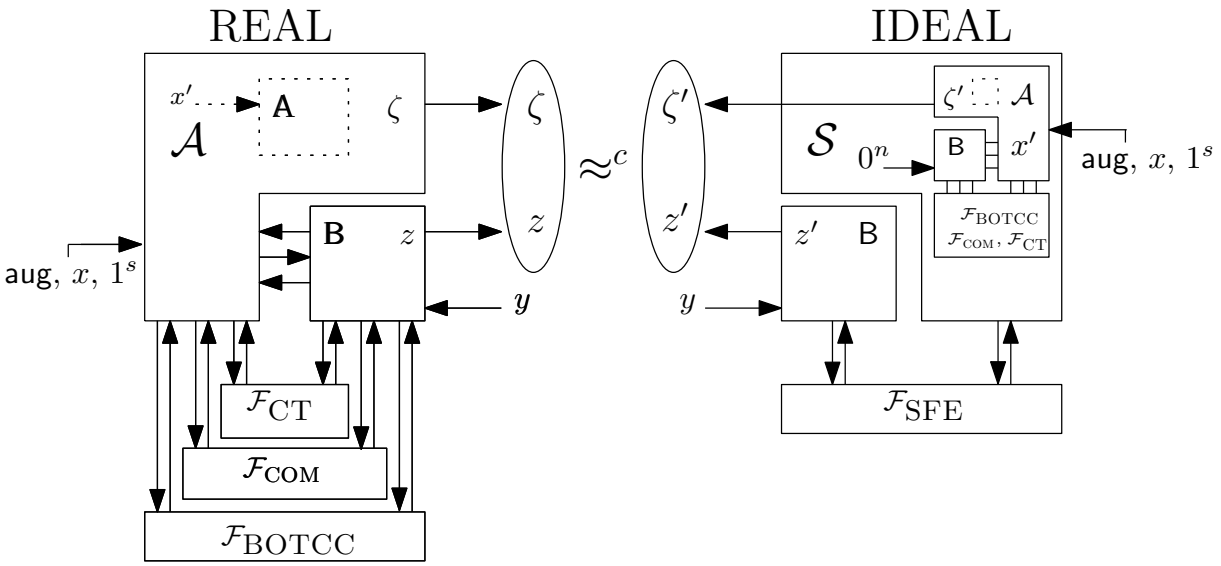

Figure 16. The ideal and the real world execution with corrupt Alice ($\mathcal{A}$). We must prove that for all adversaries $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that for all input x, y and all auxiliary input aug to $\mathcal{A}$ and all values of s it holds that the (joint) distributions $\text{IDEAL}_{\mathcal{S}}(x, y, z, \kappa)$ and $\text{REAL}_{\mathcal{A}}(x, y, z, \kappa)$ are computationally indistinguishable in κ.

- $\mathcal{S}$ starts simulating internally the real world execution of protocol π consisting of $\mathcal{A}$, Bob and the trusted parties $\mathcal{F}_{\text{BOTCC}}$, $\mathcal{F}_{\text{CT}}$ and $\mathcal{F}_{\mathsf{COM}}$. If an honest Bob would abort at any time during the simulation, then $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\mathsf{SFE}}$.
- The simulator then executes the *Setup*, *Polynomial Setup*, *Oblivious Transfer* and *Garbling* phases as an honest Bob would, but simulating his "real" input as the all-0 string, i.e., it sets $y = 0^n$.
- When reaching the *Commitment* phase the simulator extracts the ℓ (possibly different) inputs of $\mathcal{A}$ to the ℓ garbled circuits it has constructed in the *Garbling* phase. It does so directly since $\mathcal{A}$ calls $\mathcal{F}_{\mathsf{COM}}$ 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 call the i'th input bit in the j'th circuit given by $\mathcal{A}$ for x'^j_i. When we consider the entire vector of input bits in the j'th circuit we remove the subscript, i.e., x'^j, and when we consider the input to the ideal functionality we simply call it x'. In either case if we consider the augmented input we add a bar, i.e., $\bar{x}'^j_i$, $\bar{x}'^j$ and $\bar{x}'$.
- The simulator continues as an honest Bob would and chooses two random universal hash functions in the *Augmentation* phase and then runs the *Cut-and-Choose* and *Evaluation* phases.
- At the end of the *Evaluation* phase an honest Bob (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 Alice's input vectors extracted during the *Oblivious Transfer* phase to one of the evaluatable garbled circuits and input this to $\mathcal{F}_{\mathsf{SFE}}$. Otherwise, it will continue simulating an honest Bob in the *Reconstruction* phase and in turn find Alice's input to each of the garbled circuits. $\mathcal{S}$ will, like honest Bob, 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 the an input of Alice given to the correctly constructed circuits. $\mathcal{S}$ then uses this as input to $\mathcal{F}_{\mathsf{SFE}}$. Notice that there can only be one suitable set of inputs (except with probability 2^{-s}) since having more would imply that either the output decryption tables for $\mathbb{H}^{\mathsf{In}}$ were incorrectly constructed, in which case the simulator would have discarded the circuit, or Alice 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 Bob's input $\bar{y}$ is only used as input to the trusted party computing $\mathcal{F}^{\bar{n},\ell,\kappa}_{\text{BOTCC}}$. In particular, the messages that Bob sends during the protocol and hence the view of $\mathcal{A}$ do not depend on $\bar{y}$. Therefore, the output of $\mathcal{A}$ in the real world execution and $\mathcal{S}$ in the ideal world are identical.

We must, however, prove that the *joint* distribution of $\mathcal{A}$'s and Bob's output in the real world execution is identical to the *joint* distribution of $\mathcal{S}$'s and Bob'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 Bob outputs `abort` with the same probability in the ideal and real worlds. In the ideal world Bob outputs `abort` if and only if $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\mathsf{SFE}}$. By construction of $\mathcal{S}$ this happens if and only if Bob aborts in the simulation. But the probability that Bob aborts may in general depend on his input, which is y in the real execution and $0^{\bar{n}}$ in the simulation. So we must argue that the probability that Bob aborts does not depend on whether his input is y or 0^n.
2. In the cases where Bob 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 $f(x', y)$ where x' is the same input for Alice that $\mathcal{S}$ inputs to $\mathcal{F}_{\mathsf{SFE}}$ 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 Alice is committed to her inputs x'^j for each of the ℓ garbled circuits.
   (b) We show how $\mathcal{S}$ can extract x'^j for all $j \in [\ell]$ from $\mathcal{A}$.
   (c) Then we show that both $\mathcal{S}$ and honest Bob will use the x'^j as input for a circuit that is correctly garbled, committed and where the output decryption table of $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ and $\mathbb{H}^{\mathsf{Out}}$ 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 Alice is committed to is different.
   (f) Finally, we argue that even with a malicious Alice, Bob outputs $f(x', y)$ except with negligible probability, in particular that the real Bob will compute $f(x'^j, y)$ for the same x'^j as the simulator, if not he will abort.

Before we show each of these steps, consider the following characterization: We let the *circuit keys* for a garbled circuit be the symmetric keys corresponding to the input and output wires of that circuit which Alice commits to in the *Commitment* phase. We say that a garbled circuit is *good* if the plaintext circuit that is obtained by decrypting (degarbling/opening) the garbled circuit using its corresponding circuit keys results in the correct circuit (that is, the circuit that correctly computes the augmented circuit). A garbled circuit that decrypts to some other plaintext circuit, or that cannot be decrypted at all, using its circuit keys, is called a *bad* circuit. Note that after the *Commitment* phase all garbled circuits are either good or bad. In a similar manner we call an output decryption table good if it is possible to decrypt it using the keys for the output wire and that it decrypts to 0 or 1 in accordance with the semantic value of the key used.

Proof of Case 1 We now consider all the steps in which Bob can output `abort`:

Polynomial setup If the opening of the polynomials, after the cut-and-choose step, is wrong or the polynomials are not of correct degree then Bob aborts. However, which polynomials are checked is completely determined by Bob's challenge and is thus independent of his input.

Cut-and-choose Bob terminates in this phase if any of the garbled check circuits are incorrect, if Alice has not committed to the correct input keys for each of the garbled circuits, if Alice has committed to wrong output decryption tables for $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ or $\mathbb{H}^{\mathsf{Out}}$. All of these cases depend on the cut-and-choose challenge which is chosen by coin tossing[13] and thus independent of Bob's input.

Evaluation If Alice does not give the correct opening of her input commitments (or if she committed to garbage) or if she does not give correct output decryption tables for $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ then Bob will terminate. However, $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ is only based on Alice's input and thus independent of Bob's input. Finally, Bob will terminate if he cannot evaluate any of the garbled evaluation circuits. This might be based on Bob's input as he learns only one of his keys in the *Oblivious Transfer* phase. However, if Alice tries to do a selective failure attack during the OTs she cannot learn anything about Bob's true input because what Bob actually inputs to the protocol is $\bar{y}$, which by construction seems uniformly random. Still, Alice can always learn one bit of $\bar{y}$ using a selective failure attack, and even more bits if she is lucky. However, it was as proved in [LP07] if $|\bar{y}| = \max(4n, 8s)$ this does not give Alice any information about Bob's true input except with probability 2^{-s}.

[13] Failure of $\mathbb{H}^{\mathsf{Out}}$ also depends on the cut-and-choose of polynomials but we have already showed this is independent of Bob's input.

Proof of Case 2.a and 2.b We first observe that in the protocol even a malicious Alice is committed to a value associated with each of her input wires for all the $j \in [\ell]$ garbled circuits after the *Commitment* phase. The simulator can furthermore extract each of these values since Alice commits to these using $\mathcal{F}_{\mathsf{COM}}$. Furthermore, the simulator can find the semantic values (if it exists) for each of Alice's commitments by extracting Alice's "order commitments" (The *Commitment* phase, step 1). More specifically what $\mathcal{S}$ does is as follows:

- In Step 1 of the *Commitment* phase $\mathcal{S}$ learns $k^0_{i,j} \| k^1_{i,j} = k^0_{i,j} \| (k^0_{i,j} \oplus \Delta_j)$ for all $i \in [\bar{m}]$ and $j \in [\ell]$ directly by extracting them from Alice's calls to $\mathcal{F}_{\mathsf{COM}}$.
- Then in Step 2 of the same phase $\mathcal{S}$ learns $k^{x'_i}_{i,j}$ for all $i \in [m]$ and $j \in [\ell]$.
- For each $i \in [\bar{m}]$ and $j \in [\ell]$ $\mathcal{S}$ learns the bits Alice committed to by checking if $k^{x'_i}_{i,j} = k^0_{i,j}$ or $k^{x'_i}_{i,j} = k^1_{i,j}$. If the first is true then $\bar{x}'^j_i = 0$, if the second is true then $\bar{x}'^j_i = 1$.

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 and both $\mathcal{S}$ and Bob will discard their associated garbled circuits during the *Evaluation* phase (or terminate the protocol during *Cut-and-Choose*). So, unless the protocol has terminated, by the end of the *Evaluation* phase, at least one garbled circuit for evaluation will exist where each of the inputs committed to by Alice are actual keys for the corresponding garbled circuit. However, there might be more than one such circuit and Alice might have committed to keys with different semantic meaning 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}.

To see this we must show that it will not be possible for Alice to find a collision on $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ with probability larger than 2^{-s}. That is that it is not possible for her to give two inputs $x \| a, x' \| a' \in \{0,1\}^{\bar{m}}$ with $x \neq x'$ such that $(M^{\mathsf{In}} x) \oplus a = (M^{\mathsf{In}} x') \oplus a'$ with probability larger than 2^{-s}. Now remember that we have from Lemma 1 that $\mathbb{H}^{\mathsf{In}}$ computes a universal hash function with the uniform difference property, i.e., that $\mathsf{Pr}[M^{\mathsf{In}} x \oplus M^{\mathsf{In}} x' = \bar{a}] = 2^{-s}$ for any $\bar{a} \in \{0,1\}^s$. It turns out this property is enough to ensure Alice cannot find a collision. To see this notice that what we actually need to show is $\mathsf{Pr}[\left(M^{\mathsf{In}} x\right) \oplus a = \left(M^{\mathsf{In}} x'\right) \oplus a'] \leq 2^{-s}$ for all $x \| a, x' \| a' \in \{0,1\}^{\bar{m}}$ with $x \neq x'$ when M^{In} is uniformly sampled. However, setting $\bar{a} = a \oplus a'$ we see

$$\begin{aligned} M^{\mathsf{In}} x \oplus M^{\mathsf{In}} x' &= \bar{a} \\ M^{\mathsf{In}} x \oplus M^{\mathsf{In}} x' &= a \oplus a' \\ (M^{\mathsf{In}} x) \oplus a &= (M^{\mathsf{In}} x') \oplus a' \ , \end{aligned}$$

and thus we have $\mathsf{Pr}[\left(M^{\mathsf{In}} x\right) \oplus a = \left(M^{\mathsf{In}} x'\right) \oplus a'] = 2^{-s}$ and we are done.

Next see that both Bob 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, Alice could commit to different inputs without being detected by corrupting the output decryption table of $\mathbb{H}^{\mathsf{In}}$. Still, we force Alice to commit to the output decryption tables before the *Cut-and-Choose* phase, so if she chooses to cheat in the output decryption tables of $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ we then have from Lemma 4 that she cannot successfully do this for all evaluation circuits and thus there will be at least one circuit for which the $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ output decryption tables that 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 Alice along with correctly constructed output decryption tables for an instance 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 Bob and $\mathcal{S}$ can find out which of the garbled circuits and its corresponding $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ augmentation is correct. For this we have two cases:

- Either the semantic output of the garbled circuit is the same in both cases. If this occurs then there is no issue for Bob since the correct output is known in the real world and Bob can output this. In the

simulation $\mathcal{S}$ can easily find out which circuit is correct since it can extract the semantic meaning of Alice'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 $\mathcal{S}$ inputs to $\mathcal{F}_{\mathsf{SFE}}$.

– 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 a real-world Bob would; use the *Reconstruction* phase to extract x'^j for all $j \in \phi$ committed to by Alice 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 the 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 $\mathcal{S}$ and Bob will terminate the protocol follows directly from Lemma 4.

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 Alice'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 Bob and $\mathcal{S}$ will terminate the protocol by Lemma 1 and the discussion of case 2.c.

Proof of Case 2.f We finally argue that even with a malicious Alice, if x' is the semantic value of the input that Alice commits to in the *Commitment* phase associated with a correctly constructed augmented garbled circuit, then $f(x', y)$ will be the output of Bob if he does not abort.

To see this we only need to argue that in the real world, and despite a corrupted Alice, Bob outputs $f(x', y)$ where x' is the input that Alice commits to in the *Commitment* Step. To show this we consider the following theorem:

Lemma 5. *Assume we have c evaluation circuits and w output wires where at most half are bad, then the probability that we cannot find the Δ for each circuit, and thus might fail the reconstruction, is at most*

$$1 + (w/2)(2^c - 1) + \sum_{i=2}^{w/2} \left(\binom{w/2}{i} \cdot c \cdot 2^{i(c-1)} \right) .$$

Proof. See Appendix G. □

According to this theorem we can bound the probability that a corrupted Alice passes the cut-and-choose test of polynomials and that we cannot restore all the Δ values for the evaluable circuits. When we use the following remark we get 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}:

Lemma 6. *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. See Appendix G. □

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. This means that there is a total of $\binom{s_2}{s_2 - (\lceil 4.82s + 4.82 \rceil)}$ possible ways to choose the check polynomials. If more than half of the evaluation polynomials are "bad" then there is a total of $\binom{s_2 - \left(\frac{\lceil 4.82s+4.82 \rceil}{2}+1\right)}{s_2 - (\lceil 4.82s+4.82 \rceil)}$ "bad" cut-and-choose challenges. Since there is only a set of $s_2 - \left(\frac{\lceil 4.82s+4.82 \rceil}{2} + 1\right)$ good polynomials to choose from and the check size remains $s_2 - (\lceil 4.82s + 4.82 \rceil)$. Thus the probability of a "bad" cut-and-choose challenge is

$$\frac{\binom{s_2 - \left(\frac{\lceil 4.82s+4.82 \rceil}{2}+1\right)}{s_2 - (\lceil 4.82s+4.82 \rceil)}}{\binom{s_2}{s_2 - (\lceil 4.82s+4.82 \rceil)}}$$

We must now find a value for s_2 such that this expression is less than 2^{-s}. According to the following lemma this is the case if we set $s_2 \geq 6s + 7$.

Lemma 7.

$$\frac{\binom{6s+7-\left(\frac{\lceil 4.82s+4.82\rceil}{2}+1\right)}{6s+7-(\lceil 4.82s+4.82\rceil)}}{\binom{6s+7}{6s+7-(\lceil 4.82s+4.82\rceil)}} \leq 2^{-s}$$

for $s > 1$.

Proof. See Appendix G. □

Next see that at least half of the polynomials used to augmented the output wires of $\mathbb{H}^{\mathsf{Out}}$ are good (the event in Lemma 6) then we have exactly the case described in the *Reconstruction* phase except if some of the output decryption tables of $\mathbb{H}^{\mathsf{Out}}$ are wrong. However, since Alice commits to the output decryption table of $\mathbb{H}^{\mathsf{Out}}$ before the cut-and-choose phase then Lemma 4 tells us that there will be at least one circuit where this is not the case. In turn Bob will always be able to recover the input x' 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), this can happen only with probability $2^{-s} \leq 2^{-(\ell - \frac{1}{2}\log \ell + \log\left(2\frac{\sqrt{2\pi}}{e^2}\right))}$ (by Lemma 4).

This completes the proof for the case where Alice is corrupt.

E.4 Corrupt Bob

Lemma 8. *Let π be the protocol in Fig. 13. Assume that the garbling scheme $\mathcal{G} = (\mathsf{Yao}, \mathsf{Eval})$ used in π is IND-CIA secure according to Def. 1 and calls to $H(\cdot)$ in the are replaced with calls to a random oracle. Then π securely realizes $\mathcal{F}_{\mathsf{SFE}}$ in Fig. 6 in the ($\mathcal{F}_{\text{BOTCC}}$, $\mathcal{F}_{\text{CT}}$, $\mathcal{F}_{\mathsf{COM}}$)-hybrid model with abort in the presence of static, malicious adversary corrupting* Bob.

The intuition for this part of the proof is quite similar to most other two-party protocols based on garbled circuits and cut-and-choose [LP07, LP11]. Plainly speaking a malicious Bob cannot cheat since all the input he gives to the protocol is his plain input bits (to $\mathcal{F}_{\mathsf{COM}}$) and some random choices, which are used only to protect himself against a potentially malicious Alice.

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, amongst other the literal output of the protocol $f(x, y') = z$. This is not as obvious as in the case of a corrupted Alice and is achieved by $\mathcal{S}$ learning the cut-and-choose challenge directly from $\mathcal{F}_{\text{CT}}$ and then constructing the evaluation garbled circuits "maliciously" such that they always output $f(x, y') = z$ no matter what the semantic value of the input is, in particular it will still give the correct output even if the x' used in the garbled circuit is the all-0 string.

The formal proof proceeds as follows: Given an adversary $\mathcal{B}$ that controls Bob, we construct a simulator $\mathcal{S}$ in a way analogous to the case where Alice is corrupted. Formally the simulator $\mathcal{S}$ works as follows:

- $\mathcal{S}$ starts simulating internally the real world execution of the protocol consisting of Alice, $\mathcal{B}$ and the trusted parties $\mathcal{F}_{\text{BOTCC}}^{\bar{n},\ell,\kappa}$, $\mathcal{F}_{\text{CT}}$ and $\mathcal{F}_{\mathsf{COM}}$. If Alice aborts at any time during the simulation, then $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\mathsf{SFE}}$.
- The simulator then executes the *Setup*, *Polynomial Setup* and *Oblivious Transfer* phases as an honest Alice would but simulating her input as the all-0 string. However, the auxiliary s random bits $\mathcal{S}$ chooses at random. Thus it uses $x = 0^m \| a$ with $a \in_R \{0,1\}^s$.
- After the *Oblivious Transfer* phase $\mathcal{S}$ learns the input $\bar{y}'$ that $\mathcal{B}$ uses in the simulation. It then computes $M^{\mathsf{Sec}}\bar{y}' = y'$ and sends y' to $\mathcal{F}_{\mathsf{SFE}}$, which returns $z = f(x, y')$. The value y' is learned directly as $\mathcal{B}$ sends this to $\mathcal{F}_{\text{BOTCC}}^{\bar{n},\ell,\kappa}$ only once and in plain. Next $\mathcal{S}$ uses the outcome of the coin tossing protocol $\mathcal{F}_{\text{CT}}$ to select the cut-and-choose challenge in the *Cut-and-Choose* phase like an honest Alice would.
- $\mathcal{S}$ then proceeds to constructs the garbled circuits. However, it only constructs half of them correctly, the other half it hardcodes to compute the value z returned from $\mathcal{F}_{\mathsf{SFE}}$ previously. In particular, based on the output of the coin tossing $\mathcal{S}$ constructs exactly the check circuits correctly and hardcodes the evaluation circuits to always output z. Specifically this is done by appropriately switching the bit encrypted in the garbled computation tables. This implies that $\mathcal{B}$ will always be happy with the check circuits supplied, yet all the evaluation circuits output z as defined by the functionality $f(x, y')$.

– In the *Commitment* phase $\mathcal{S}$ proceeds as honest Alice would.
– In the *Augmentation* phase $\mathcal{S}$ receives the specification for $\mathbb{H}^{\mathsf{In}}$ and $\mathbb{H}^{\mathsf{Out}}$ from Bob. $\mathcal{S}$ then generates the output decryption tables of $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ and $\mathbb{H}^{\mathsf{Out}}$ as an honest Alice would for the check circuits. For the evaluation circuits $\mathcal{S}$ decides on a uniformly random string of s bits and hardcodes all of $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$ parts of the evaluation circuits to compute this. For $\mathbb{H}^{\mathsf{Out}}$ is uses the output z received from $\mathcal{F}_{\mathsf{SFE}}$ and constructs the output decryption tables as an honest Alice would.
– $\mathcal{S}$ now proceeds with the *Cut-and-Choose* and *Evaluation* phases as an honest Alice would.

To prove security we must show the following:

1. We must argue that Alice 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 $x\|a$ or $0^m\|a'$ with $a, a' \in_R \{0,1\}^s$.
2. If Alice does not abort then we must argue that, except with negligible probability in s, Alice 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 Alice's input x except $f(x, y') = z$.

The first two steps encompasses the indistinguishability of the *joint* distributions of $\mathcal{B}$'s output and Alice's output in the ideal and in the real world and the last step that the output of the execution is indistinguishable in the real and ideal world.

Proof of Case 1 Notice that Alice is not allowed to abort during the protocol.

Proof of Case 2 This case is quite easy to argue about since Alice does not receive any "real" output. What she receives is perhaps a message saying `abort`. However, since Alice is honest this will happen with the same probability in both the real and ideal world.

Proof of Case 3 This is the hardest case and to this end we must first argue that $\mathcal{S}$ can convince $\mathcal{B}$ to learn the true value $z = f(x, y')$ with the same probability in the real and ideal world. To do this we first observe that in the real protocol even a corrupt Bob is committed to a specific input y' at the time he sends his input to $\mathcal{F}_{\text{BOTCC}}^{\bar{n},\ell,\kappa}$. Since $\mathcal{S}$ fully controls the simulation of $\mathcal{F}_{\text{BOTCC}}^{\bar{n},\ell,\kappa}$, $\mathcal{S}$ can therefore easily extract y' from Bob's communication with $\mathcal{F}_{\text{BOTCC}}^{\bar{n},\ell,\kappa}$ and output this to $\mathcal{F}_{\mathsf{SFE}}$ (as done in the simulation above). This means that in the real world, $\mathcal{B}$ learns $f(x, y') = z$ from $\mathcal{F}_{\mathsf{SFE}}$. This is the same as in the ideal world since $\mathcal{S}$ hardcodes the evaluation garbled circuits to be exactly $f(x, y') = z$.

Furthermore, $\mathcal{B}$ cannot learn anything about the semantic values of Alice's input keys (or any other wire keys in the circuit) for the evaluation circuits because of the IND-CIA security of the garbling scheme. This is needed as we use a dummy 0 input for Alice in the simulation.

In the reduction to IND-CIA we will simply ask the IND-CIA game for an evaluation of the dummy 0 values of Alice or her real input. We will keep Bob's input the same as in the simulation. We then embed the circuit we get back into the simulation. We evaluate the circuit we get back on the input keys we get back and then embed the resulting output keys into the simulation as we would have done in the simulation, i.e., we construct output decryption tables making them hit the desired output.

One subtlety in the embedding done in this reduction is that the output decryption table contains verifiable commitments to all output keys, and we only get to learn one key for each output wire from the IND-CIA game.[14] We handle this by replacing the commitments of the other keys by verifiable commitments to uniformly random values. Since we are in the hybrid model for verifiable commitments this goes unnoticed until the commitment functionality is queried on one of the keys we did not learn, so if the adversary could distinguish between the random commitments and the true commitments, then it is because he can query the functionality for commitment on the other key. But then we have both keys for one of the wires and can compute the value Δ for that wire, which allows us to win the IND-CIA game. Specifically, for all queries x to the ideal functionality for commitment done by the adversary, we will query $\mathcal{O}_\Delta(x \oplus k)$ for all output keys k which we learned from the IND-CIA game.

[14] In fact, we only get one set of *input* keys, but we can get the corresponding key for all wires simply by evaluating the circuit on the given input keys.

Another subtlety in the embedding done in the above reduction is that we need to give the links $L_{i,j} = G\left(P_{\chi[i]}(j), k'^0_{i,j}\right)$ to the adversary. This is only possible for the wires where we have learned the 0 key. However, for all the wires where we learn only the 1 key, it holds that if we did not give the links to Bob, then it would be hard for Bob to compute any point on the corresponding polynomial or the 0 key: The points on the polynomials have high min-entropy in the view of Bob, as he knowns only t points, and computing a 0 key would be the same as breaking the security of the garbling scheme. Hence, if we consider the distribution $(x_1, y_1, \ldots, x_\ell, y_\ell, z)$ generated by running the protocol and letting z be the view of Bob with the links removed which correspond to wires for which he learned the 1 key and if we let the (x_i, y_i) be the corresponding $\left(P_{\chi[i]}(j), k'^0_{i,j}\right)$, then this distribution is hard. Hence we can replace the links $L_{i,j} = G\left(P_{\chi[i]}(j), k'^0_{i,j}\right)$ by $G(a, b)$ for uniformly random a and b. Note that we only need to do this step twice. First we replace all the involved links by random links, then we can do the hybrids where we circuit by circuit replace input a with input 0 in the circuits, and then we can go back and replace the random links by correct links.

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 $\mathcal{S}$ are xor'ed with $M^{\mathsf{In}}x$. 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 a cheating Alice can try to cause can only happen with probability at 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 $\lceil 2.41s + 3.41 \rceil$ 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 $\mathbb{H}^{\mathsf{In}} \oplus \mathsf{a}$.
   - This can only happen if she gives at least two different inputs to the protocol.[15]
4. Alice 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 other than $f(x, y)$.

However, nothing is preventing Alice from trying to cause each of these events. Still, it should be noted that if Alice is unsuccessful in her cheating attempt of event 1, 2 or 3 then Bob will notice and can safely abort the protocol. Thus trying to make more than just a single of the events than 1, 2 or 3 happen will only increase the probability of Alice being caught and Bob terminating the protocol without the compromising the security. Now notice that list of bad events are in the order of which Bob 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 Alice should try to make event 4 and 5 occur. This is so since if there is a collision on $\mathbb{H}^{\mathsf{Out}}$ Bob cannot terminate the protocol without possible leaking some information on his private input and since a selective failure attack does not necessarily make Bob abort the protocol. Notice that she cannot combine an attack for event 4 or 5 with attack 1, 2 or 3 since if one of these are unsuccessful then Bob 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 Alice 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 $n \times \max(4n, 8(s+1))$ matrix and $\mathbb{H}^{\mathsf{Out}}$

[15] There can seemingly be a collision if Alice hardwires the output decryption tables 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.

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 7).

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 player in its simulation (and hence obtain the honest players output from $\mathcal{F}_{\mathsf{SFE}}$).

Therefore, we immediately achieve UC security [Can01] if the underlying OT protocol used for the OTs and coin tossing protocol 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 2 is stated in the hybrid world 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.

Corollary 1. *Let* $(\mathsf{Yao}, \mathsf{Eval})$ *be the concrete Yao garbling scheme of Fig. 4. Let* π *be the protocol in Fig. 13 to Fig. 15, but where the call to* $\mathcal{F}_{\text{BOTCC}}$ *is replaced by invocation of the protocol* π_{BOT} *in Fig. 10 (in which calls to* $\mathcal{F}_{\text{OT}}$ *are replaced by an execution of the OT protocol in [PVW08]) and where calls to* $\mathcal{F}_{COM}$ *are replaced with invocations of the protocol* π_{COM} *from Appendix C.2 and where the calls to* $\mathcal{F}_{\text{CT}}$ *are replaced by invocations of the protocol* π_{CT} *from Appendix C.4 and where calls to* $H(\cdot)$ *in the subprotocols above are done with calls to the random oracle. Then in the random oracle model* π *UC-securely realizes* $\mathcal{F}_{SFE}$ *with abort in the presence of static, malicious adversaries assuming* DLP *is hard and both parties have access to a common reference string.*

Proof. The $(\mathsf{Yao}, \mathsf{Eval})$ scheme in Fig. 4 is shown to be IND-CIA secure in the random oracle model in [PSSW09]. As proved in Appendix C.2 the protocol π_{COM} UC-securely realizes $\mathcal{F}_{\mathsf{COM}}$ in the random oracle model. Next, consider that $\mathcal{F}_{\text{BOTCC}}$ can be implemented by an invocation of π_{BOT} along with a random oracle and a protocol for $\mathcal{F}_{\text{OT}}$ as described in Appendix C.3, where $\mathcal{F}_{\text{OT}}$ can be realized as in [PVW08] using a common reference string and the assumption that DLP is hard. Then $\mathcal{F}_{\text{CT}}$ can be realized as described in Appendix C.4. Finally, all of the above realization are UC-secure in the presence of static and malicious adversaries. Hence, by Theorem 2 and the UC-composition theorem Cor. 1 follows immediately from Theorem 2. □

F The Replication Factor

Using Stirling's approximation formula we find a lower bound the circuit replication factor ℓ required in order to achieve Alice guessing the cut-and-choose challenge with probability at most 2^{-s}. We restate Lemma 4 here:

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* $\log\left(2\frac{\sqrt{2\pi}}{e^2}\right) - \frac{1}{2}\log(\ell) + \ell \geq s$.

Proof (of Lemma 4). First notice that a cheating Alice 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. We now bound this number using Stirling's approximation, assuming $\ell \geq 2$:

$$\begin{aligned}\binom{\ell}{\frac{\ell}{2}} &= \frac{\ell!}{\frac{\ell}{2}!\left(\ell-\frac{\ell}{2}\right)!} = \frac{\ell!}{\left(\frac{\ell}{2}!\right)^2} \geq \frac{\sqrt{2\pi\ell}\left(\frac{\ell}{e}\right)^{\ell}}{\left(e\sqrt{\frac{\ell}{2}}\left(\frac{\frac{\ell}{2}}{e}\right)^{\frac{\ell}{2}}\right)^2} = \frac{\sqrt{2\pi\ell}\left(\frac{\ell}{e}\right)^{\ell}}{e^2\frac{\ell}{2}\left(\frac{\ell}{2e}\right)^{\ell}} = \frac{\sqrt{2\pi\ell}\ell^{\ell}e^{-\ell}}{e^2 2^{-1}\ell\ell^{\ell}2^{-\ell}e^{-\ell}} \\ &= \frac{2\sqrt{2\pi\ell}2^{\ell}}{e^2\ell} = \frac{2\sqrt{2\pi}}{e^2}\cdot\frac{\sqrt{\ell}2^{\ell}}{\ell} = \frac{2\sqrt{2\pi}}{e^2}\cdot\ell^{-\frac{1}{2}}2^{\ell}\end{aligned}$$

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 Alice'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:

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

G Polynomial Cut-and-Choose

We restate Lemma 5 here:

Assume we have c evaluation circuits and w output wires and at most half of the wires have an associated polynomial that is bad (of degree greater than t), then the probability that we cannot find the Δ for each circuit, and thus might fail the reconstruction, is at most

$$\frac{1 + (w/2)(2^c - 1) + \sum_{i=2}^{w/2}\left(\binom{w/2}{i} \cdot c \cdot 2^{i(c-1)}\right)}{2^{cw}}$$

Proof (of Lemma 5). 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 theorem 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 cw 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 of the wires in the circuits that does not make it possible to do interpolations such that we can learn the Δ 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, no matter how its associated bit is set. 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 bits from all c 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'ed 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 Δ values. Next see that because of the potential exsistance of very bad wires the worst cases will always be when we have $w/2$ very bad wires so for simplicity we can simply consider bad cases on the remaining $w/2$ wires. The reason being if we can restore the Δ values with only $w/2$ wires then we can always restore it with more wires, 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 this wire, 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 is not the 1-string. 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 remaning wires must all be the 1-string. So we get a total of $(w/2)(2^c - 1)$ 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, w.l.o.g. 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 a 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[16] $2^{2(c-1)}$, since each wire constitute 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 $c2^{2(c-1)}$ cases. Finally see that we can select two AND0 wires in $\binom{w/2}{2}$ different ways, so we get $\binom{w/2}{2}c2^{2(c-1)}$ 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 this happens 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:

$$1 + (w/2)(2^c - 1) + \sum_{i=2}^{w/2} \left(\binom{w/2}{i} \cdot c \cdot 2^{i(c-1)} \right) .$$

Dividing number with the total amount of possible cases (2^{cw}) 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.

We restate Lemma 6 here:

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 Alice, 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 $\frac{1+(w/2)(2^c-1)+\sum_{i=2}^{w/2}\left(\binom{w/2}{i}\cdot c\cdot 2^{i(c-1)}\right)}{2^{c\cdot w/2}}$ 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.

[16] We count some redundant configurations here.

Under these constraints consider probability of the worst case, i.e. that the output is discrepant on 2 circuits:

$$\begin{aligned}
\frac{1+(w/2)(2^2-1)+\sum_{i=2}^{w/2}\left(\binom{w/2}{i}\cdot 2\cdot 2^{i(2-1)}\right)}{2^{2\cdot w/2}} &= \frac{1+3(w/2)+\sum_{i=2}^{w/2}\left(\binom{w/2}{i}\cdot 2\cdot 2^{i}\right)}{2^{w}} \\
&= \frac{1+3(w/2)+2\cdot\sum_{i=2}^{w/2}\left(\binom{w/2}{i}\cdot 2^{i}\right)}{2^{w}} \\
&= \frac{1+3(w/2)+2\cdot\left(\sum_{i=0}^{w/2}\left(\binom{w/2}{i}\cdot 2^{i}\right)-w-1\right)}{2^{w}} \\
&= \frac{1+3(w/2)+2\cdot\left((1+2)^{w/2}-w-1\right)}{2^{w}} \\
&= \frac{1+3(w/2)+2\cdot 3^{w/2}-2\cdot w-2}{2^{w}} \\
&= \frac{2\cdot 3^{w/2}-(w/2)-1}{2^{w}} \\
&= 2\cdot 2^{-w}\cdot 3^{w/2}-2^{-w}\cdot(w/2)-2^{-w} \\
&= 2^{-w+1}\cdot 2^{\log(3)w/2}-2^{-w}((w/2)+1) \\
&= 2^{-w+1+\log(3)w/2}-2^{-w}2^{\log((w/2)+1)} \\
&= 2^{1+w\left(\frac{\log(3)}{2}-1\right)}-2^{-w+\log((w/2)+1)} \\
&\leq 2^{1+w\left(\frac{\log(3)}{2}-1\right)}-2^{1-w}
\end{aligned}$$

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, $w = -\frac{s}{\frac{\log(3)}{2}-1}-\frac{1}{\frac{\log(3)}{2}-1}$.

$$\begin{aligned}
2^{-s} &\geq^{?} 2^{1+\left(-\frac{s}{\frac{\log(3)}{2}-1}-\frac{1}{\frac{\log(3)}{2}-1}\right)\left(\frac{\log(3)}{2}-1\right)}-2^{1-\left(-\frac{s}{\frac{\log(3)}{2}-1}-\frac{1}{\frac{\log(3)}{2}-1}\right)} \\
&= 2^{1+(-s-1)}-2^{1+\frac{s}{\frac{\log(3)}{2}-1}+\frac{1}{\frac{\log(3)}{2}-1}} \\
&= 2^{-s}-2^{1+\frac{s}{\frac{\log(3)}{2}-1}+\frac{1}{\frac{\log(3)}{2}-1}}
\end{aligned}$$

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 notice that since the amount of evaluation circuits are less than s for $s \geq 1$ then $c < w \Rightarrow 2s < \lceil 4.82s+4.82 \rceil$ is clearly true and we are done. □

We restate Lemma 7 here:

$$\frac{\binom{6s+7-\left(\frac{\lceil 4.82s+4.82\rceil}{2}+1\right)}{6s+7-(\lceil 4.82s+4.82\rceil)}}{\binom{6s+7}{6s+7-(\lceil 4.82s+4.82\rceil)}} \leq 2^{-s}$$

for $s > 1$.

Proof. Consider $2^{-s}/\frac{\binom{6s+7-\left(\frac{\lceil 4.82s+4.82\rceil}{2}+1\right)}{6s+7-(\lceil 4.82s+4.82\rceil)}}{\binom{6s+7}{6s+7-(\lceil 4.82s+4.82\rceil)}}$. If we can show this is an increasing function for $s > 1$, then we are done, as this means that 2^{-s} is growing faster than $\frac{\binom{6s+7-\left(\frac{\lceil 4.82s+4.82\rceil}{2}+1\right)}{6s+7-(\lceil 4.82s+4.82\rceil)}}{\binom{6s+7}{6s+7-(\lceil 4.82s+4.82\rceil)}}$ and in turn that

$\frac{\binom{6s+7-\left(\frac{\lceil 4.82s+4.82\rceil}{2}+1\right)}{6s+7-(\lceil 4.82s+4.82\rceil)}}{\binom{6s+7}{6s+7-(\lceil 4.82s+4.82\rceil)}} < 2^{-s}$. We see this by taking the derivative of $2^{-s}/\frac{\binom{6s+7-\left(\frac{\lceil 4.82s+4.82\rceil}{2}+1\right)}{6s+7-(\lceil 4.82s+4.82\rceil)}}{\binom{6s+7}{6s+7-(\lceil 4.82s+4.82\rceil)}}$ and noticing this is always positive.

□

H Two-Output Functionality

Until now we have only considered functions where Bob receives the output and where Alice does not receive any output. This is an artefact of the Yao garbling approach, caused by its asymmetric nature, in which functions with only output to Bob is simpler. In the general case both Alice and Bob gets private output $f_A(x, y) = z_A$ and $f_B(x, y) = z_B$, respectively. Since Alice can just input a one-time pad r, which – at least using the free-xor technique – is essentially free, constructing two-output functions is essentially a matter of ensuring *authenticity* or *correctness*, meaning mechanisms letting Alice be sure that the result she receives from Bob is indeed the *correct* result $f_A(x, y)$.

In basic Yao, a common technique for authenticating Alice's result is to let Bob send the output keys obtained to Alice. But in circuit cut-and-choose many circuits are being evaluated. If Bob just sends the keys to Alice, she can do the following attack: She can let just one of the circuits be wrong, for instance just outputting Bob's private output. With probability $1/2$ this is not caught in the check phase, and from the output keys she learns Bob's output.

Lindell and Pinkas [LP07] uses a one-time (information theoretic) MAC. This increases Alice's input with $|z_A| + 2s$ bits and the circuit size with $O(s|z_A|)$ gates, that is, $O(s^2|z_A|)$ inexpensive cryptographic operations.

Shelat and Shen [sS11] instead uses digital signatures and witness-indistinguishable proofs, giving $O(s|z_A|)$ *heavy* cryptographic operations. Mohassel and Riva [MR13] brings this down to $O(s|z_A|)$ *inexpensive* operations.