LEGO for Two Party Secure Computation

Jesper Buus Nielsen and Claudio Orlandi

BRICS, Department of Computer Science, Aarhus Universitet,
Åbogade 34, 8200 Århus, Denmark
{buus,orlandi}@cs.au.dk

Abstract The first and still most popular solution for secure two-party computation relies on Yao's garbled circuits. Unfortunately, Yao's construction provide security only against passive adversaries. Several constructions (zero-knowledge compiler, cut-and-choose) are known in order to provide security against active adversaries, but most of them are not efficient enough to be considered practical. In this paper we propose a new approach called LEGO (Large Efficient Garbled-circuit Optimization) for two-party computation, which allows to construct more efficient protocols secure against active adversaries. The basic idea is the following: Alice constructs and provides Bob a set of garbled NAND gates. A fraction of them is checked by Alice giving Bob the randomness used to construct them. When the check goes through, with overwhelming probability there are very few bad gates among the non-checked gates. These gates Bob permutes and connects to a Yao circuit, according to a fault-tolerant circuit design which computes the desired function even in the presence of a few random faulty gates. Finally he evaluates this Yao circuit in the usual way.
For large circuits, our protocol offers better performance than any other existing protocol. The protocol is universally composable (UC) in the OT-hybrid model.

Key words: Two Party Computation, Yao's Circuits, UC security.

Contents

1 Introduction

In secure two-party computation we have two parties, Alice and Bob, that want to compute a function $f(\cdot,\cdot)$ of their inputs a, b, and learn the result $y = f(a, b)$. The term 'secure' refers to the fact that Alice and Bob don't trust each other, and therefore want to run the computation with some security guarantees: informally they want to be sure that y is computed correctly and they don't want the other party to be able to learn anything about their secret (except for what they can efficiently learn from the output of the computation itself). Secure two-party computation was introduced by Yao in [Yao86] and later generalized to the multi-party case in [GMW87]. Those solutions offer computational security and are based on the evaluation of an "encrypted" Boolean circuit. For the multi-party case, assuming that some majority of the parties are honest, it is possible to achieve information theoretically secure protocols [CCD88,BGW88].

Yao's original construction is only secure against a passive adversary, i.e. an adversary that doesn't deviate from the protocol specifications but can inspect the exchanged messages trying to extract more information about the inputs of the other parties. A more realistic definition of security allows the adversary to arbitrarily deviate from the protocol: in this case a cheating party can try to learn the inputs of the other party or force the computation to output an incorrect value. Such an adversary is called malicious or active.

A way to design protocols that are secure against active adversaries is to start with a passive protocol and then compile it into an active one using standard techniques. The main idea here is that the parties commit to their inputs, randomness and intermediate steps in the computation. Then they prove in zero-knowledge that the values inside the commitments were computed according to the protocol. The compiler technique is of great interest from a theoretical point of view, but unfortunately it is not really practical, because of the generic zero-knowledge proofs involved.

Our Contribution: In this paper we focus on the two-party protocol based on Yao's garbled circuits [Yao86]. We assume the reader to be familiar with the protocol.[1]

In Yao's protocol Alice constructs a garbled circuit and sends it to Bob: a malicious Alice can send Bob a circuit that doesn't compute the agreed function, therefore the computation loses both privacy and correctness. In our protocol instead Alice and Bob both participate in the circuit construction. The main idea of our protocol is to have Alice prepare and send a bunch of garbled NAND gates (together with some other components) to Bob. Then Bob uses these gates to build a circuit that computes the desired function. Bob can in fact, with a limited help from Alice, solders the gates as he likes. Then the circuit is evaluated by Bob as in Yao's protocol: Bob gets his keys running oblivious transfers (OT) with Alice and he evaluates the circuit.

As usual in Yao's circuit, it is easy to cope with a malicious Bob, as he can't really deviate from the protocol in a malicious way. It is more challenging (and interesting) to try to prevent Alice from cheating, as Alice can cheat in several ways, including:

Circuit generation: In the standard Yao's protocol, Alice can send to Bob a circuit that does not compute the desired function. In particular, she could send a circuit computing the function $f(a, b) = b$, where b is Bob's input, clearly violating Bob's privacy. In our construction, Alice sends to Bob the components of the circuit, and Bob assembles them. So Alice cannot choose a function and force Bob to compute it.

[1] A complete description of the protocol and the proof of its security can be found in [LP04].

However she could still cheat and send some bad gates: Alice could cheat by sending something that is not a NAND gate, meaning either another Boolean function or some gate that outputs error on some particular input configuration. In this case the circuit will output an incorrect value and eventually compromise Bob's security.

Input phase: As in Yao's protocol, suppose Bob has an input bit b and he needs to retrieve the corresponding key K_b for an input gate. To do so, Alice and Bob run an OT where Alice inputs K_0, K_1 and Bob learns K_b. If Alice is malicious, she might input just one real key, together with a bad formed one. For instance, Alice could input K_0, R, with R a random string. In this case, if Bob's input is 0, he gets a real key and he succeed in the circuit evaluation. If Bob's input is 1, he gets a bad key, so he cannot evaluate the circuit and therefore he aborts the protocol. Observing Bob's behavior, Alice learns Bob's input bit. This kind of cheating is usually referred to as *selective failure*.

To deal against a malicious Alice, Bob has the following tools:

Gate verification: When Alice provides the components to Bob, he will ask with some probability to open the components to check their correctness. The test on the components is destructive, and will induce on the untested components a probability of being correct. So at the end of this preprocessing phase Bob knows that the amount of bad components is limited by some constant with some small probability — s bad gates with probability 2^{-s}, say.

Gate shuffling: Once Bob gets a suitable amount of gates, he randomly permutes them in a circuit computing the desired function. In this way, the bad gates that might have passed the first test are placed in random positions, so Alice can't force adversarial errors, but just random errors.

Fault-tolerant circuit design: To cope with the residual bad gates, Bob computes every gate with some redundancy and takes a majority vote of the outputs.

Leakage-tolerant input layer: To protect against possible selective error in the input phase, Bob encodes his input in a way that even if Alice learns some bits of the encoded input, she doesn't actually learn anything but some random bits.

These techniques lead to an efficient protocol for two-party secure computation. The sketch of the protocol is the following: first Alice prepares and sends to Bob components of the following kind:

NAND gates: A gate is defined by 6 keys $L_0, L_1, R_0, R_1, K_0, K_1$, representing the two possible values for the input wires L, R and the output wire K. When Bob inputs two keys L_a, R_b, he gets as output the key $K_{\overline{a \cdot b}}$.

Key check (KC): A KC is defined by two keys K_0, K_1 representing the Boolean values on that wire. When Bob gets a key K' for that wire, the wire check signals whether K' is correct (i.e. $K' \in \{K_0, K_1\}$) or not.

Any of these components can be malfunctioning and lead to an erroneous computation. To check that most of the components are actually working, Bob checks the correctness of a fraction of them, in a cut-and-choose flavor.

Then Alice and Bob agree on a function to be computed. Bob randomly permutes the components and solder them, with Alice's help. Bob replaces every gate of the original circuit with a small amount of (possibly malfunctioning) gates and KCs, in a redundant way. There is no fan-out limit, i.e. any output wire can be soldered to any number of

input wires of the next layer of gates. Alice sends the keys corresponding to her input bits, while Bob retrieves his keys using OTs. Finally Bob evaluates the Yao circuit and sends the output keys to Alice, that can therefore retrieve the output of the circuit.[2]

We prove the protocol to be UC secure [Can01] in the OT hybrid model.

Related Work: In the last years many solutions have been proposed to achieve two-party computation secure against malicious adversaries: in Lindell and Pinkas' solution [LP07], Alice sends s copies of the circuit to be computed. Bob checks half of them and computes on the remaining circuits. Due to the circuit replication, they need to introduce some machinery in order to force the parties to provide the same inputs to every circuit, resulting in an overhead of s^2 commitments per input wire for a total of $O(s|C| + s^2|I|)$, where $|C|$ is the size of the circuit and $|I|$ is the size of the input. The protocol offers a non-UC simulation proof of security. The complexity of our protocol is of the order $O(s \log(|C|)^{-1}|C|)$ i.e. our replication factor is reduced by the logarithm of the circuit size. Therefore, our construction is especially appealing for big circuits.

Going more into the details, the protocol in [LP07] requires s copies of a circuit of size $|C|+|D|$, where D is an input decoder, added to the circuit to deal with the selective failure attack. They propose a basic version of it with size $O(s|I|)$ and a more advanced with size $O(s+|I|)$. However, because of the $s^2|I|$ commitments, their optimized encoding gives them just a benefit in the number of OT required. With our construction, we can fully exploit their encoding. In fact we need just to replicate $s/\log(|C|)$ times a circuit of size $O(|C| + s + |I|) = O(|C|)$.

The protocol from [LP07] was recently implemented in [LPS08]. It would be interesting to implement also our protocol, to find out which protocol offers better performance for different circuit sizes. In particular, we note that we can instantiate our primitives (encryption scheme, commitments, OT) with the ones they used. In this case our protocol should achieve a weaker security level, the same as [LPS08]. Clearly we can't preserve UC security if we instantiate our protocol with a non-UC OT.

Other related works include: To optimize the cut-and-choose construction, Woodruff [Woo07] proposed a way of proving input consistency using expander graphs: using this construction it is possible to get rid of the dependency on the input size and therefore achieving communication and computational complexity of $O(s|C|)$. Considering UC security, in [JS07] a solution for two party computation on committed inputs is presented. This construction uses public key encryption together with efficient zero-knowledge proofs, in order to prove that the circuit was built correctly. Their asymptotic complexity is $O(|C|)$. However, due to their massive use of factorization-based public key cryptography, we believe that our protocol will offer better performance.

2 Preliminaries and Notation

Security parameters: the protocol has two security parameters: κ is a computational security parameter for the commitment schemes, oblivious transfers, encryption schemes and hash functions used, while s is a statistical security parameter that deals with the

[2] If both parties need to get the output, standard constructions can be used.

probability that the computation fails due to bad components.[3] Alice's security relates just to κ, while Bob's security relates to both κ and s.

For the protocol to work, we choose a prime number p of size $O(\kappa)$. The keys for the Yao gates are going to live in the group $(\mathbb{Z}_p, +)$, and every time we write $K + K'$ we actually mean $K + K' \mod p$.

Hash function: The protocol needs a hash-function $H : \mathbb{Z}_p \to \mathbb{Z}_p$. In the analysis we model it as a non-programmable and non-extractable random oracle for convenience. This assumption can be replaced by a complexity theoretical assumption very similar to the notion of correlation robustness in [IKNP03]. Most hash functions of course work on bit-strings. We can use any non-programmable and non-extractable random oracle $G : \{0,1\}^* \to \{0,1\}^\ell$ as $H(x) = G(x) \bmod p$, where $\ell > \log_2(p) + s$ and $x \in \{0,1\}^\ell$ is considered a number $x \in [0, 2^\ell)$.

Encryption scheme: We implement our symmetric encryption scheme with the hash function in the following way: Our encryption scheme has the functionality $E : \mathbb{Z}_p \times \mathbb{Z}_p \to \mathbb{Z}_p$, $D : \mathbb{Z}_p \times \mathbb{Z}_p \to \mathbb{Z}_p$. To encrypt we compute $C = E_K(M) = H(K) + M$. To decrypt we compute $M = D_K(C) = C - H(K)$.

3 The Ideal Functionality

The ideal functionality is described in Fig. 1. It is "insecure" in the sense that it allows Alice to guess input bits. This can be solved in a block-box manner by computing a function of a randomized encoding of the input, where any s bits are uniformly random and independent. This allows Alice to guess s or more bits with probability at most 2^{-s} and ensures that guessing less bits will not leak information: The guessed bits are uniformly random and independent. One method for this is given in [LP07]. The extra number of gates used is $O(|b| + s)$, where $|b|$ is the length of Bob's input and s the statistical security parameter. From now on we can therefore focus on implementing the slightly insecure ideal functionality above.

The ideal functionality $\mathcal{F}^{\text{SFE}}$ runs as follows:

Circuit and inputs: Alice inputs (a, c_A), Bob inputs (b, c_B), and c_A and c_B are leaked to the adversary. If $c_A \neq c_B$, then the ideal functionality outputs `disagreement!` to both parties and terminates. Otherwise, let $c = c_A$ and parse c as $(\ell_A, \ell_B, \ell_y, C, \ell)$, where $\ell_A, \ell_B, \ell \in \mathbb{N}$ and C is a NAND circuit with $\ell_A + \ell_B$ input wires and ℓ_y output wires. The size of the circuit C is L NAND gates. Parse a as $a \in \{0,1\}^{\ell_A}$ and $b \in \{0,1\}^{\ell_B}$. Here ℓ is a replication parameter.

Corrupt Alice: If Alice is corrupted, she can specify a set $\{(i, \beta_i)\}_{i \in I}$, where $I \subseteq \{1, \ldots, \ell_B\}$ and $\beta_i \in \{0,1\}$. If $\beta_i = b_i$ for $i \in I$, then output `correct!` to Alice. Otherwise, output `You were nicked!` to Alice and output `Alice cheats!` to Bob.

Evaluation: If both parties are honest or Alice was not caught above, then the ideal functionality computes $y = f(a, b)$ and outputs y to Alice. The adversary decides the time of delivery.

Figure 1. The ideal functionality for secure circuit evaluation.

[3] We use κ to instantiate cryptographic primitives. Given that all cryptographic primitives were perfectly secure, the insecurity in s would be 2^{-s}, independent of the running time of the adversary. For a given desired security level $2^{-\sigma}$ one therefore needs κ to grow with the computational capability of the adversary, whereas s can be fixed. We therefore think of s as smaller than κ.

4 The Protocol

The main structure of the protocol is as follows.

Setup: Alice and Bob agree on the parameters (κ, s, p). They run in the hybrid world with an ideal functionality for OT and agree on a hash function H.

Commitment scheme: Alice and Bob samples a public key pk for the trapdoor commitment scheme using an UC-secure coin flip protocol (see Appendix H). All later commitments from Alice to Bob are performed under this key.

Global difference: Alice samples $\Delta, r_\Delta \in_R \mathbb{Z}_p$ with $\Delta \neq 0$ and sends the commitment $[\Delta; r_\Delta]$ to Bob. She also gives a zero-knowledge UC-secure proof of knowledge of Δ (see Appendix I). We return to the use of Δ later.

Component production: Alice produces a number of components and sends them to Bob. For each type of component Alice sends a constant factor more components than needed.

Component checking: For each type of component Bob randomly picks a subset of the components to be checked. Alice sends the randomness used to generate them, and Bob checks the components.

Soldering: Bob permutes the remaining components, and use them to build a Yao circuit which computes the desired function. The circuit is constructed such that it can tolerate a few bad components when they occur at random positions. To connect the components we require Alice to open some commitments towards Bob.

Evaluation: Bob gets his input keys using the OT, and Alice sends her keys. Then Bob evaluates the circuit in a similar way as in Yao's protocol. Bob sends Alice the output keys.

Alice and Bob do not have to agree on the function to be computed until the soldering step and do not have to know their inputs until the evaluation step.

4.1 Commitment Scheme

We need a perfectly hiding trapdoor commitment scheme, homomorphic in $\mathbb{Z}_p$: for this purpose we can use a Pedersen's commitment scheme over the group of points of an elliptic curve of size p. Once the elliptic curve is selected, we have a group $(G, p, \cdot, g)$ where G is a multiplicative abelian group of order p, and g generates G. To generate a key for the commitment scheme, a random element h will be sampled, using an UC coin-flip protocol. For the proof to work we need in fact the simulator to be able to extract the trapdoor $t \in \mathbb{Z}_p$ s.t. $h = g^t$. Now the public key is $pk = (G, p, \cdot, g, h)$ and the trapdoor is t. To commit compute $\mathrm{comm}_{pk}(x; r) = g^x h^r$. The commitment is perfectly hiding and computationally binding, and given the trapdoor t it is possible to open the commitment to any value.

To keep the notation simple we will write $[K]$ for $\mathrm{comm}_{pk}(K; r)$, for some randomness r. When computing with the commitments we will write $[K_0 + K_1] = [K_0] + [K_1]$ meaning $\mathrm{comm}_{pk}(K_0 + K_1; r + s) = \mathrm{comm}_{pk}(K_0; r) \cdot \mathrm{comm}_{pk}(K_1; s)$ for some r, s. When we want to stress the randomness, we write $[K; r]$ for $\mathrm{comm}_{pk}(K; r)$. When we write $x = \mathrm{open}([x])$ we mean: Alice sends Bob x, r, and Bob checks that $g^x h^r = [x]$. If not Bob aborts the protocol, else he outputs x.

4.2 Global Difference

To each wire we associate one uniformly random key K_0, which we call the *zero-key*. We then define $K_c = K_0 + c\Delta$ for $i = 1, 2, \ldots$. In the evaluation of the circuit, if a wire carries the value $c \in \{0, 1, 2, \ldots\}$ [4] then Bob will learn the key K_c, and *only* the key K_c. Since K_0 is uniformly random, K_c is uniformly random, and thus does not reveal the value of c. We here note that it is important that the global difference Δ is hidden from Bob, as Δ would allow Bob to compute all keys for a given wire from just one of the keys.

We will have gates with the functionality $e : \{0, 1, \ldots, C\} \times \{0, 1, \ldots, D\} \to \{0, 1, \ldots, E\}$ for positive integers C, D and E. Let L_0 be the zero-key for the left wire, let R_0 be the zero-key for the right wire and let O_0 be the zero-key for the output wire. Then the garbled version of e will map (L_c, R_d) to $O_{e(c,d)}$ for $(c, d) \in \{0, 1, \ldots, C\} \times \{0, 1, \ldots, D\}$.

We already now note that the addition gate can be garbled in a trivial manner.[5] The garbling consists of the shift value $S = L_0 + R_0 - O_0$. Given L_c and R_d one evaluates the garbling by computing $L_c + R_d - S = (L_0 + c\Delta) + (R_0 + d\Delta) - L_0 - R_0 + O_0 = (c + d)\Delta + O_0 = O_{c+d}$.

4.3 Component Production

We describe how the components are generated and checked, and describe their intended use. Later we describe how to connect components to get a Yao circuit. For all components we call the randomness used by Alice to compute the component the *generator* of the component. It is insecure for Alice to send the entire generator to Bob in the check, as it would reveal Δ to Bob, which would violate the security of Alice. Therefore a finite challenge set E will be given. In the check Bob will send $e \in E$ to Alice and Alice returns part of the generator. Bob then checks that part. In Appendix L it is argued that the checks are complete in the sense that given an answer to all challenges one can compute a generator.

NT Gates The first component is the *not-two gate* or *NT gate*. It is a slightly fuzzy garbling of the unary function $\mathrm{nt} : \{0, 1, 2\} \to \{0, 1\}$ where $\mathrm{nt}(c) = 0$ iff $c = 2$. It will be used to implement a NAND gate by applying it to the sum of the two input bits.[6] The generator for an NT gate consists of two keys $I_0, O_0 \in \mathbb{Z}_p$, two randomizers for the commitment scheme $r_I, r_O \in \mathbb{Z}_p$ and a permutation π of $\{0, 1, 2\}$. The gate is of the form

$$([I_0; r_I], [O_0; r_O], C_0, C_1, C_2) = \mathrm{NT}(I_0, O_0, r_I, r_O, \pi) \ ,$$

where $C_{\pi(0)} = E_{I_0}(O_1)$, $C_{\pi(1)} = E_{I_1}(O_1)$, $C_{\pi(2)} = E_{I_2}(O_0)$. Here $I_c = I_0 + c\Delta$ and $O_d = O_0 + d\Delta$ for the global difference Δ. Note that $C_{\pi(c)} = E_{I_c}(O_{\mathrm{nt}(c)})$ for $c = 0, 1, 2$.

Intended Use: Given an NT gate $NT = ([I_0, r_I], [O_0, r_O], C_0, C_1, C_2) = \mathrm{NT}(I_0, O_0, r_I, r_O, \pi)$ and a valid input key $I_c = I_0 + c\Delta$, for $c \in \{0, 1, 2\}$, one can compute $K_d = D_{I_c}(C_d)$ for $d = 0, 1, 2$. One of these keys will be the key $D_{I_c}(C_{\pi(c)}) = O_{\mathrm{nt}(c)}$. I.e., the NT gate NT maps I_c to a set of three *candidate keys* which contains $O_{\mathrm{nt}(c)}$. For a given NT gate NT and a key I we denote the set of candidate keys by $NT(I)$. Another component, called

[4] Usually $c \in \{0, 1\}$, but other values are possible.

[5] In fact any linear function can be garbled in this way, see Appendix E.

[6] In Appendix G how to implement any Boolean gate using just one nt gate is explained.

key checks, will be used to find the correct key among the three candidates $NT(I)$. Key checks are described below.

It is convenient to introduce the concept of a shifted NT gate, as we will need it in the soldering phase. A *shifted NT* (SNT) is of the form (Σ_I, Σ_O, NT) with $\Sigma_I, \Sigma_O \in \mathbb{Z}_p$ being *shift values*. The two shifts define a new zero-key $I'_0 = I_0 - \Sigma_I$ for the input wire and a new zero-key $O'_0 = O_0 - \Sigma_O$ for the output wire. These zero-keys define $I'_c = I'_0 + c\Delta$ and $O'_d = O'_0 + d\Delta$. Given I'_c one can evaluate the gate by computing the candidate keys $SNT(I'_c) = NT(I'_c + \Sigma_I) - \Sigma_O$, which contains $O_{\mathrm{nt}(c)} - \Sigma_O = O'_{\mathrm{nt}(c)}$. We call SNT a SNT for the keys I'_0, O'_0. We can consider an NT gate a SNT gate with $\Sigma_I = \Sigma_O = 0$. Alice will only produce NT gates. The SNT gates are produced as part of the soldering.

Check: Clearly Alice cannot send the whole generator to Bob when an NT gate is being checked, as it would leak Δ to Bob. Therefore Bob will pick a uniformly random challenge $e \in \{0, 1, 2\}$ and Alice will reveal partial information on the generator as follows:

- If $e = 0$, then Alice sends $\pi_0 = \pi(0)$ and $I_0 = \mathrm{open}([I_0])$ and $O_1 = \mathrm{open}([O_0] + [\Delta])$ to Bob who checks that $C_{\pi_0} = E_{I_0}(O_1)$.
- If $e = 1$, then Alice sends $\pi_1 = \pi(1)$ and $I_1 = \mathrm{open}([I_0] + [\Delta])$ and $O_1 = \mathrm{open}([O_0] + [\Delta])$ to Bob who checks that $C_{\pi_1} = E_{I_1}(O_1)$.
- If $e = 2$, then Alice sends $\pi_2 = \pi(2)$ and $I_2 = \mathrm{open}([I_0] + 2[\Delta])$ and $O_0 = \mathrm{open}([O_0])$ to Bob who checks that $C_{\pi_2} = E_{I_2}(O_0)$.

We note that while checking the correctness of the NT gates, Bob is also checking the that $\Delta \neq 0$. It's trivial to notice that $\Delta \neq 0$ iff $C_0 \neq C_1 \neq C_2$ or Alice can find collisions for the hash function.

Key Checks A generator for a *key check* (KC) consists of a key $K_0 \in \mathbb{Z}_p$, a randomizer $r_K \in \mathbb{Z}_p$ and a bit $\pi \in \{0, 1\}$. The KC is of the form

$$([K_0; r_K], T_0, T_1) = \mathrm{KC}(K_0, r_K, \pi) \ ,$$

where $T_\pi = H(K_0), T_{1-\pi} = H(K_0 + \Delta)$.

When a KC is checked, the challenge is $e \in \{0, 1\}$. The replies are as follows:

- If $e = 0$, then Alice sends π and $K_0 = \mathrm{open}([K_0])$ and Bob checks that $T_\pi = H(K_0)$.
- If $e = 1$, then Alice sends π and $K_1 = \mathrm{open}([K_0] + [\Delta])$ and Bob checks that $T_{1-\pi} = H(K_1)$.

Intended Use: Given a KC $KC = ([K_0; r_K], T_0, T_1) = \mathrm{KC}(K_0, r_K, \pi)$ and an arbitrary key K one outputs $KC(K) = 1$ if $H(K) \in \{T_0, T_1\}$ and outputs $KC(K) = 0$ otherwise.

It is clear that if $K \in \{K_0, K_0 + \Delta\}$, then $KC(K) = 1$. Assume then that $KC(K) = 1$ but $K \notin \{K_0, K_0 + \Delta\}$. Assume that we are in addition given Δ and the generator of the KC. Since $KC(K) = 1$ we have that $T_b = H(K)$ for some b. We know that $T_b = H(K_0 + c\Delta)$ for some $c \in \{0, 1\}$, where K_0 is given in the generator. Since $K \notin \{K_0, K_0 + \Delta\}$ we have that $K \neq K_0 + c\Delta$. It follows that we can efficiently compute the collision $H(K) = H(K_0 + c\Delta)$. This means that if we know Δ and a generator such that $KC = \mathrm{KC}(K_0, r_K, \pi)$, then we know that $KC(K) = 1$ iff $K \in \{K_0, K_0 + \Delta\}$, except with negligible probability.

A *shifted KC* (*SKC*) is of the form (Σ, KC) for $([K_0; r_K], T_0, T_1) = \mathrm{KC}(K_0, r_K, \pi)$ and $\Sigma \in \mathbb{Z}_p$. It defines a new zero-key $K'_0 = K_0 - \Sigma$ and $K'_b = K'_0 + b\Delta$. We call SKC a SKC for the key K'_0. We evaluate SKC as $SKC(K) = KC(K + \Sigma)$. Since $KC(K) = 1$ iff $K \in \{K_0, K_1\}$ (except with negligible probability), we have that $SKC(K') = 1$ iff $K' \in \{K'_0, K'_1\}$. Alice will only produce KCs. The SKCs are generated as part of soldering.

Details of Component Production We now describe the details of the component production.

NT gates: We let L be the number of NAND gates in the circuit, and ℓ be the replication factor. Then $N = (\ell+1)L$ is the number of NT gates needed. Alice prepares $\phi_1 N$ NT gates and sends them to Bob. Bob picks a random subset C of size about and at most $(\phi_1 - 1)N$ for testing, such that about and at most a fraction $\epsilon_1 = (\phi_1 - 1)/\phi_1$ is checked. For each $i \notin C$ Bob sends $e^{(i)} = 3$ to Alice to indicate that component i is not checked. For $i \in C$ Bob picks random $e^{(i)} \in \{0, 1, 2\}$ and sends $e^{(i)}$ to Alice. Alice reveals part $e^{(i)}$ of the randomizer of component i to Bob, and Bob checks it. If any check fails, then Bob terminates with output `Alice cheats!`. If all checks succeed, N of the remaining NT gates are used.

KCs: Here $N = (2\ell + 1)L$ is the number of KCs needed. Alice prepares $\phi_2 N$ and Bob checks about and at most $(\phi_2 - 1)N$, or about and at most a fraction $\epsilon_2 = (\phi_2 - 1)/\phi_2$. If any check fails, then Bob terminates with output `Alice cheats!`. If all checks succeed, N of the remaining KCs are used.

For the checks to be zero-knowledge, as detailed in the analysis in Section 5, we need the challenge from Bob to be flipped using a UC secure coin-flip protocol. This can be implemented via the OT functionality (see Appendix H). Since UC secure coin-flipping is inefficient, we use an optimization trick, which to the best of our knowledge is new: We use the UC coin flip to flip a short seed $S \in \{0, 1\}^{O(s)}$. We then define the set C and the challenges $e^{(i)}$ from S. The set C makes up at most a fraction ϵ_i of the whole set and has a distribution such that for all fixed subsets I with $|I| \leq \beta_i$ (where $\beta_i = O(s)$ is some threshold fixed by the analysis) and for fixed challenges $e_0^{(i)} \in E_i$ for $i \in I$, it holds that

$$\Pr[\nexists i \in I(e^{(i)} \neq e_0^{(i)})] \leq \frac{4}{3}(1 - \epsilon_i/|E_i|)^{|I|} + 2^{-\beta_i} \ ,$$

where $E_1 = \{0, 1, 2\}$ and $E_2 = \{0, 1\}$ are the challenge spaces used when checking components. Using challenges with such a distribution is sufficient to extract witnesses for all but $O(s)$ components. We discuss in Appendix F how to construct such challenges by using a $2^{O(s)}$-almost $O(s)$-wise independent sampling space with seed length $O(s)$.

4.4 Key Alignment

An important part of the circuit construction done by Bob consists of key alignment, which allows to take a key associated with some wire and securely shift it to become identical to a key associated to some other wire. Since all keys are committed to, this is done simply by opening the difference between the two commitments and using this difference as a key shift. We go over the different types of key alignment.

Aligning Output Wires with Input Wires Given two NT gates

$$NT^{(1)} = ([I_0^{(1)}], [O_0^{(1)}], C_0^{(1)}, C_1^{(1)}, C_2^{(1)}) \ , \quad NT^{(2)} = ([I_0^{(2)}], [O_0^{(2)}], C_0^{(2)}, C_1^{(2)}, C_2^{(2)})$$

and an NT gate $NT = ([I_0], [O_0], C_0, C_1, C_2)$, Bob can ask Alice to make $NT^{(1)}$ and $NT^{(2)}$ the input gates of NT. In response to this Alice shows $\Sigma_I = \text{open}([I_0] - [O_0^{(1)}] - [O_0^{(2)}])$ to Bob and they both replaces NT with $SNT = (\Sigma_I, 0, NT)$. Now SNT is an SNT for input key $I'_0 = I_0 - \Sigma_I = O_0^{(1)} + O_0^{(2)}$.

In particular, given $O_a^{(1)} = O_0^{(1)} + a\Delta$ and $O_b^{(2)} = O_0^{(2)} + b\Delta$ for $a, b \in \{0, 1\}$, Bob can compute $O_a^{(1)} + O_b^{(2)} = O_0^{(1)} + O_0^{(2)} + (a + b)\Delta = I'_{a+b}$. If in addition the NT gate NT is correct, Bob can then compute $SNT(I'_{a+b}) = O_{\text{nt}(a+b)} = O_{a \,\text{NAND}\, b}$.

Intuitively, the alignment is secure as the only value which is leaked is $\Sigma = I_0 - O_0^{(1)} - O_0^{(2)}$. Since I_0 is uniformly random, and NT is placed in only one position in the circuit, I_0 acts as a one-time pad encryption of the sensitive information $O_0^{(1)} - O_0^{(2)}$.

Aligning NT Gates The above way to connect NT gates allows to build a circuit of NAND gates. Such a circuit can, however, be incorrect, and even insecure, if just one NT gate is incorrect. To deal with this we use replication of NT gates. For this we need to be able to take two NT gates and make their input keys the same and their output keys the same. We say that we align the keys of the NT gates.

Given two NT gates

$$NT^{(1)} = ([I_0^{(1)}], [O_0^{(1)}], C_0^{(1)}, C_1^{(1)}, C_2^{(1)}] \quad , NT^{(2)} = ([I_0^{(2)}], [O_0^{(2)}], C_0^{(2)}, C_1^{(2)}, C_2^{(2)}] \ ,$$

Bob can ask Alice to align the keys of $NT^{(2)}$ with those of $NT^{(1)}$. In response to this Alice will send $\Sigma_I = \text{open}([I_0^{(2)}] - [I_0^{(1)}])$ and $\Sigma_O = \text{open}([O_0^{(2)}] - [O_0^{(1)}])$ to Bob and they both let $SNT^{(2)} = (\Sigma_I, \Sigma_O, NT^{(2)})$. It is clear that if $NT^{(2)}$ was a correct NT gate for keys $(I_0^{(2)}, O_0^{(2)})$, then it is now a correct SNT gate for keys $(I_0^{(1)}, O_0^{(1)})$.

Intuitively, the alignment is secure as $I_0^{(2)}$ acts as a one-time pad encryption of $I_0^{(1)}$ and $O_0^{(2)}$ acts as a one-time pad encryption of $O_0^{(1)}$ — each $NT^{(2)}$ will have its keys aligned with at most one NT gate.

Given $NT, NT^{(1)}, \ldots, NT^{(\ell)}$ we can produce $NT, SNT^{(1)}, \ldots, SNT^{(\ell)}$ by doing the ℓ alignments from $(NT, NT^{(i)})$ to $(NT, SNT^{(i)})$ for $i = 1, \ldots, \ell$. As a result all the (S)NT gates $NT, SNT^{(1)}, \ldots, SNT^{(\ell)}$ will be for the same keys (I_0, O_0).

The intended use of aligned NT gates is as follows: Given I_c for $c \in \{0, 1, 2\}$ each correct $SNT^{(i)}$ (let $SNT^{(0)} = NT$ with the all-zero shifts) will have the property that $O_{\text{nt}(c)} \in SNT^{(i)}(I_c)$. The incorrect $SNT^{(i)}$ might produce only wrong keys, but if there is just one correct gate among the $\ell + 1$ gates, then $\cup_i SNT^{(i)}(I_c)$ will contain $O_{\text{nt}(c)}$. We use KCs to identify the correct key, as described now.

Aligning KCs with NT gates Given an NT gate $NT = ([I_0], [O_0], C_0, C_1, C_2)$ and a key check $KC = ([K_0], T_0, T_1)$ we align KC with NT by Alice sending $\Sigma = \text{open}([K_0] - [O_0])$ to Bob and both letting $SKC = (\Sigma, KC)$. If SKC was a correct KC for K_0, then it is now a correct SKC for O_0.

Intuitively, the alignment is secure as the only leakage is $\Sigma = K_0 - O_0$, where K_0 acts as a one-time pad encryption of O_0 — each KC will be aligned with at most one NT.

The intended use of aligned KCs is as follows: Assume that we are given a set $\{K_i\}$ of keys with $O_b \in \{K_i\}$ for $b \in \{0,1\}$ and that we are given a correct SKC SKC for O_0. Now compute $SKC(K)$ for all keys $K \in \{K_i\}$. Since $SKC(O_b) = 1$, there will be at least one K for which $SKC(K) = 1$, and if $SKC(K) = 1$ for exactly one key K, then $K = O_b$ was found.

If $SKC(K) = 1$ and $SKC(K') = 1$ for $K \neq K'$, then except with negligible probability $\{K, K'\} = \{O_0, O_0 + \Delta\}$. Even thought we cannot determine the right key, we can compute Δ from these two keys. We return to how this case is dealt with in Section 5.

4.5 Fault-Tolerant Circuit Design

Given a pool of unused NT gates and KCs, Alice and Bob constructs a Yao circuit for $f(a, b)$ as follows.

Circuit: First Alice and Bob agree on a NAND circuit C which computes the function $f(a, b)$. We assume that they agree on f, so we can assume that they agree on the circuit too. Before each input wire w, where one of the parties is to provide the input bit a, the parties prepend one NAND gate which outputs to wire w — we call this *input gate* w. This creates two new input wires. When evaluating the circuit, we will input $\bar{a}$ on both of them, to make $a = \bar{a} \,\text{NAND}\, \bar{a}$ appear on wire w.[7]

Backbone: For each NAND gate g in C, Bob picks a uniformly random unused NT gate NT and associates NT with g, we write $NT^{(g)} = NT$, and announces this association to Alice. For each $l, r, g \in C$, where l and r are the input gates of g, Alice and Bob then makes $NT^{(l)}$ and $NT^{(r)}$ the input gates of $NT^{(g)}$ by a key alignment.

Replication: For each $g \in C$, Bob

- picks ℓ uniformly random unused NT gates $NT^{(g,1)}, \ldots, NT^{(g,\ell)}$ and aligns their keys with the keys of $NT^{(g)}$. Let $NT^{(g,0)} = NT^{(g)}$.
- picks $2\ell+1$ unused KCs $KC^{(g,1)}, \ldots, KC^{(g,2\ell+1)}$ and aligns these with the output key of $NT^{(g)}$.

4.6 Circuit Evaluation

The circuit is evaluated as follows.

Alice's Input: For each input gate $g \in C$ for which Alice is to provide the input bit a_i, Alice sends the key $I_{a_i+1} = I_{2-a_i}$ to Bob, where I_0 is the input key of $NT^{(g)}$. Bob lets $I^{(g)} = I_{2-a_i}$ denote the received key. Note that $\text{nt}(2 - a_i) = a_i$. So, if $NT^{(g)}$ is correctly evaluated it will output $O^{(g)}_{a_i}$.

Bob's Input: For each input gate $g \in C$ for which Bob is to provide the input bit b_i, Alice and Bob run an OT. Alice offers the values $((I_2, r_2), (I_1, r_1))$, where $(I_2, r_2) = \text{open}([I_0] + 2[\Delta])$ and $(I_1, r_1) = \text{open}([I_0] + [\Delta])$ and $[I_0]$ is the input commitment of $NT^{(g)}$. Bob uses the input bit b_i as selection bit, to learn (I_{2-b_i}, r_{2-b_i}). Bob checks that (I_{2-b_i}, r_{2-b_i}) is an opening of $[I_0] + (2 - b_i)[\Delta]$, Note that $\text{nt}(2 - b_i) = b_i$. So, if $NT^{(g)}$ is correctly evaluated it will output $O^{(g)}_{b_i}$.

Evaluation: Bob then computes an output key $O^{(g)}$ for all gates $g \in C$ as follows:

[7] The reason for this construction is that the gate will be implemented by an NT gate, which has only one input key. The construction is sketched in Fig. 4 on page 19.

- If g is an input gate, then let $I^{(g)}$ be the key defined above. Otherwise, let l and r be the input gates of g and let $I^{(g)} = O^{(l)} + O^{(r)}$.
- Let $\mathcal{K} = \cup_{i=0}^{\ell} NT^{(g,i)}(I^{(g)})$.
- Compute $KC^{(g,1)}(\mathcal{K}), \ldots, KC^{(g,2\ell+1)}(\mathcal{K})$ and let $\mathcal{O}^{(g)}$ consist of the keys K which are in at least $\ell + 1$ of these sets. If $|\mathcal{O}^{(g)}| = 0$, then output `Alice cheats!`. If $|\mathcal{O}^{(g)}| = 1$, then let $O^{(g)}$ be the element in $\mathcal{O}^{(g)}$. If $|\mathcal{O}^{(g)}| > 1$, then $\mathcal{O}^{(g)} = \{O_0^{(g)}, O_1^{(g)}\}$. If g is an input gate then Bob outputs `Alice cheats!`. Otherwise, Bob can extract Δ as explained in Appendix J and use it to determine c such that $I^{(g)} = I_c^{(g)}$. Then Bob lets $O^{(g)} = O_{\mathrm{nt}(c)}^{(g)}$.

Output: For each output gate $g \in C$ Bob sends $O^{(g)}$ to Alice. If $O^{(g)} \notin \{O_0^{(g)}, O_1^{(g)}\}$ for some output gate, then Alice outputs `Bob cheats!`. Otherwise she determines for each output gate y_i such that $O^{(g)} = O_{y_i}^{(g)}$, which defines the output y.

5 Analysis

In the analysis with use the following lemma (formalized and proved in Appendix K).

Informal Lemma 1 *Assume that Alice is corrupted and Bob is honest. Let ϵ_1 denote the fraction of NT gates being checked and let ϵ_2 denote the fraction of KCs being checked.*

- *Let $\gamma_1 = 1 - \epsilon_1/3$. If Bob accepts the checks of the NT gates with probability $\geq 2\gamma_1^{\beta_1}$, then generators for all NT gates used in the protocol, except at most β_1, can be extracted in expected poly-time.*
- *Let $\gamma_2 = 1 - \epsilon_2/2$. If Bob accepts the checks of the KCs with probability $\geq 2\gamma_2^{\beta_2}$, then generators for all KCs used in the protocol, except at most β_2, can be extracted in expected poly-time.*

The extractor uses black-box rewinding access to the environment and the adversary. I.e., it is not a "UC extractor".

It is fairly easy to see that if each gate is made out of $\ell+1$ NT gates of which at most ℓ are bad[8] and $2\ell + 1$ KCs of which at most ℓ are bad, then Bob will compute correct keys. We are therefore interested in the probability that all L gates are composed of at least one good NT gate and $\ell + 1$ good KCs.

Here is a ball game: There are L buckets. The player picks $B = (\ell+1)L$ balls, where β_1 balls are red and the rest green. A player picking β_1 red balls is let into the second phase of the game with probability $2\gamma_1^{\beta_1}$ for some fixed $\gamma_1 \in [0, 1]$. In the second phase, the balls will be distributed uniformly at random into the L buckets. The player wins if it is let into the second phase of the game and at least one bucket ends up containing only red balls. There is a second variant of the game, where β_2 balls are let into the second phase with probability $2\gamma_2^{\beta_2}$, where buckets have size $2\ell+1$ and where the player wins if it gets at least $\ell + 1$ red balls in the same bucket. It is straight-forward to prove the following lemma (see Appendix A).

Lemma 1. *Consider a player which plays both games in parallel and wins if it wins either game. There is no strategy which allows it to win with probability better than $L^{-\ell}(2(0.37(\ln \gamma_1^{-1})^{-1})^{\ell+1} + (0.99(\ln \gamma_2^{-1})^{-1})^{\ell+1})$.*

[8] We could not extract a correct generator using the above lemma.

Combining the above lemmas we get the following informal lemma.

Informal Lemma 2 *Assume that Alice is corrupted and Bob is honest. Let ϵ_1 denote the fraction of NT gates being checked, let ϵ_2 denote the fraction of KCs being checked. There exists an expected poly-time extractor with black-box rewinding access to the environment and the adversary which can extract generators for at least one NT gate and at least $\ell+1$ KCs per gate in the circuit when Bob accepts the check with probability at least $P = L^{-\ell}(2(0.37(\ln \gamma_1^{-1})^{-1})^{\ell+1} + (0.99(\ln \gamma_2^{-1})^{-1})^{\ell+1})$ with $\gamma_1 = 1 - \epsilon_1/3, \gamma_2 = 1 - \epsilon_2/2$*

We pick the parameters $(\ell, \epsilon_1, \epsilon_2)$ such that $P \leq 2^{-s}$. For fixed constants ϵ_1, ϵ_2, the size of the garbled circuit is $O(\ell L) = O(sL/\log L)$. Note that P drops and the communication complexity of the protocol grows with all three parameters ℓ, ϵ_1 and ϵ_2. It is therefore by far trivial for a given size L of the circuit to pick the parameter triple $(\ell, \epsilon_1, \epsilon_2)$ which optimizes the protocol under the constraint that $P \leq 2^{-s}$. Some possible choices for the parameters are provided in Appendix B.

5.1 Cheating Alice

We argue security against a cheating Alice by showing that all ways to cheat can be simulated in the ideal world. The sketch easily translates into a simulation proof in the UC model.

When Bob rejects the checks in the simulation, then the simulator will input `abort!` to the ideal functionality on behalf of Alice. Assume now that Bob accepts the proof with probability $\leq 2^{-s}$. This is negligible in the security parameter s, so this case is simulated by Alice inputting `abort!` to the ideal functionality, except with negligible probability.

We can now assume that Bob accepts the proof with probability $\geq 2^{-s}$. This means that there exists an expected poly-time extractor which could extract a generator for one NT gate and $\ell + 1$ KCs per gate in the circuit. We call the components for which generators are known correct, the others incorrect. Given the generator for the correct components we can define[9] the *correct output keys* $O_0^{(g)}, O_1^{(g)}$ for each gate as the output keys occurring in the correct NT gate, properly shifted. In the same way we can compute the *correct input keys* $I_0^{(g)}, I_1^{(g)}$.

It is easy to see that $\mathcal{O}^{(g)} \subseteq \{O_0^{(g)}, O_1^{(g)}\}$, except with negligible probability, where $\mathcal{O}^{(g)}$ is the set computed by Bob.[10] Furthermore, if $I^{(g)} = I_c^{(g)}$ for $c \in \{0, 1, 2\}$, then $O_{\mathrm{nt}(c)}^{(g)} \in \mathcal{O}^{(g)}$, as the correct NT gate outputs $O_{\mathrm{nt}(c)}^{(g)}$. We can also assume Alice, so the simulator, knows Δ as she gives a proof of knowledge detailed in Appendix I.

Alice's Input: When Alice sends $I^{(g)}$ she could herself compute the $\mathcal{K}$ and $\mathcal{O}^{(g)}$ computed by Bob in **Evaluation**. If $|\mathcal{O}^{(g)}| = 0$, then Bob aborts, which Alice can accomplish in the ideal world by inputting `abort!`. If $|\mathcal{O}^{(g)}| = 1$, then Bob assigns the element in $\mathcal{O}^{(g)}$ to $O^{(g)}$. This element is either $O_0^{(g)}$ or $O_1^{(g)}$. Let a_i be the smallest bit such that $O^{(g)} = O_{a_i}^{(g)}$. Alice can compute $O^{(g)}$ and can use the $\ell + 1$ correct KCs to compute $O_0^{(g)}$ and $O_1^{(g)}$, and therefore a_i.[11] In the ideal world Alice would simply use a_i as her input

[9] We are not describing a behavior of the simulator, just making definitions!

[10] By the majority voting, each K in $\mathcal{O}^{(g)}$ was accepted by at least one correct KC.

[11] If Alice knows one correct output key, then because she knows Δ, she can compute the other one. A correct KC can then be used to tell which is which. Since a majority of the KCs are correct, this allows Alice to determine a_i.

bit. If $|\mathcal{O}^{(g)}| > 1$, then $\mathcal{O}^{(g)} = \{O_0^{(g)}, O_1^{(g)}\}$. Bob lets $O^{(g)}$ be a random of these elements. In the ideal world Alice could simply have input a uniformly random a_i to the ideal functionality.

Bob's Input: When Alice offers values $((I_2, r_2), (I_1, r_1))$ then call (I_c, r_c) *bad* if it is not an opening of the corresponding commitment. If both values are bad for some OT, then Bob will always see a bad value and terminate. In the ideal world Alice could have input `abort!`. Otherwise, let I be the indices of OTs for which Alice inputs exactly one bad value, and let β_i be the value of b_i which would lead Bob to not see the bad value. If Bob terminates then Alice knows that $b_i \neq \beta_i$ for some $i \in I$. If Bob does not terminate, then Alice knows that $b_i = \beta_i$ for all $i \in I$. Alice could accomplish the same in the ideal world by inputting I and $\{\beta_i\}_{i \in I}$ to the ideal functionality.

Evaluation: If Bob did not yet abort, then Bob now has $O^{(g)} = O_{a_i}^{(g)}$ for each of Alice's input gates and $O^{(g)} = O_{b_i}^{(g)}$ for each of his own input gates, corresponding to Alice and Bob having input a and b in the ideal world. From a and b we can compute the value c that should go into all NT gates given these initial inputs to the circuit and the bit $d = \mathrm{nt}(c)$ that the gates should output. The output wire of input gates holds the corresponding correct keys (as defined above). So, we can assume inductively that the input key to an internal NT gate g is $I_c^{(g)}$ for the correct $c \in \{0, 1, 2\}$. This means that $O_d^{(g)}$ will be in $\mathcal{O}^{(g)}$. In particular, $|\mathcal{O}^{(g)}| = 1$ or $|\mathcal{O}^{(g)}| > 1$. If $|\mathcal{O}^{(g)}| = 1$ then because $O_d^{(g)}$ is in $\mathcal{O}^{(g)}$, Bob set $O^{(g)} = O_d^{(g)}$, as desired. If $|\mathcal{O}^{(g)}| > 1$, then $\mathcal{O}^{(g)} = \{O_0^{(g)}, O_1^{(g)}\}$. This gives him $\Delta' = \pm\Delta$. It is easy to see that Bob can use the good components to find Δ from Δ' (details in Appendix J).

He then determines the correct output value d for gate g and lets $O^{(g)} = O_d^{(g)}$. The value d is found as follows: For each input gate g of Alice, Bob takes the key $O^{(g)} = O_{a_i}^{(g)}$, which at this point we can assume correct. Then he uses Δ and the $\ell + 1$ KCs of g to determine the value of a_i: He checks if there is a hash of $O^{(g)} - \Delta$ or $O^{(g)} + \Delta$ and set $a_i = 1$ respectively $a_i = 0$ and he takes the majority vote. Then he computes $y = f(a, b)$ and the value c and d for all internal NT gates along the way.

At the end Bob sends $O^{(g)}$ to Alice for all output gates. We argued above that all keys were computed correctly, so $O^{(g)} = O_{y_i}^{(g)}$ for the correct output bit y_i in $y = f(a, b)$.

In the ideal world Alice can learn y from the ideal functionality by not gambling for the input. Given y_i she could then have computed $O^{(g)} = O_{y_i}^{(g)} = O_0^{(g)} + y_i\Delta$ herself. Note that to compute $O^{(g)} = O_{y_i}^{(g)} = O_0^{(g)} + y_i\Delta$ she needs Δ and $O_0^{(g)}$. The value Δ she knows, as she gave a proof of knowledge. There is no *a priori* reason why she should know $O_0^{(g)}$ as she could have prepared the components used to build gate g in any fashion, in particular without knowing the correct output key. Note, however, that she does know the values she offered in the OT — we are in the OT-hybrid model. This means that Alice knows a correct key $I^{(w)}$ for all of Bob's input wires w. She also knows the keys that she sent to Bob — we assume an ideal secure channel. From these she can (as done by Bob) compute at least one correct key $O^{(w)}$ for all input gates w. She can evaluate the circuit on these keys —as done by Bob— and learn one output key $O_{y'_i}^{(g)}$ for each output gate g, along with the bit y'_i. She knows Δ and can therefore compute $O_0^{(g)}$, as desired.

5.2 Cheating Bob

We consider the case where Alice is honest and Bob is cheating. Here the simulator gets the input b of Bob and nothing else.

The simulator discards the value b, as the real input of Bob is going to be defined via how Bob behaves in the protocol. The simulator runs Alice honestly, with input $a = 0^*$. If Bob sends keys which makes Alice reject in the simulation, then the simulator inputs `abort!` to the ideal functionality on behalf of Bob. Otherwise, the simulator takes the input bit b_i of Bob to be the negation[12] of his selection bit in the OT for the corresponding input wire.

Correctness: For each output wire w, Bob learns $O^{(w)}_{y_i}$, where $y = f(a, b')$, and Bob knows which value $O'^{(w)}_{y_i}$ he is sending to Alice, as we assume ideal secure channels. We can therefore define $\Delta^{(w)} = O'^{(w)}_{y_i} - O^{(w)}_{y_i}$ for each output wire. It is clear that if Alice accepts the keys, then it holds for each output wire w that $\Delta^{(w)} = 0$ or $\Delta^{(w)} \in \{\Delta, -\Delta\}$. I.e., either all keys are correct or Bob can guess Δ with probability negligibly close to $\frac{1}{2}$.[13] It is easy to see that when H is a correlation robust hash function, then the probability with which Bob can guess Δ is negligible. It follows that when Alice accepts, then all keys sent by Bob are correct.

Privacy: The above argument, that Alice only accepts correct keys, did not depend on the use of the correct input a. It also holds in the simulation where $y = f(0^*, b')$. In fact, Bob will know when he sends the keys whether Alice rejects or not. The simulator can therefore simulate Bob by inputting `abort!` to the ideal functionality on behalf of Bob if there is some output wire w for which Bob uses $\Delta^{(w)} \neq 0$.

What remains is then to show that the view of Bob is indistinguishable in the simulation and the real protocol. The only difference between the two is in whether Alice uses input $a' = 0^*$ or the real a.

For now, let us ignore the way the components are checked and assume that Alice simply prepares enough good components to build the circuit. It is straight-forward to verify that when H is a uniformly random function, the soldered circuit leaks no information on a to a party which did not query H on a pair of points P_1 and P_2 of the form $P_2 = P_1 + \Delta$ or $P_2 = P_1 + 2\Delta$. And, until this happens Δ is uniformly random in the view of Bob. We can therefore use the birthday bound to pick p such that no adversary can make such queries except with negligible probability.

Let's now also consider the way the components are checked: The problem with the previous argument for the indistinguishability of circuits with different a's is that during the check Alice sends parts of the generators to Bob, and the generators include Δ. The way around this problem is that the simulator can make Bob accepts the checks even without knowing Δ: The checks are simulatable by knowing the challenges and the trapdoor for the commitments, which allows to answer exactly one challenge without knowing Δ, as explained in Appendix L. The simulator gets the trapdoor as it simulates the coin flip protocol for the commitment pk. To foresee which challenge Bob is going to send, the simulator picks a random seed S and prepares the incorrect components to pass the check against the challenges sampled given S. When it is time to generate the seed, it simulates the coin flip protocol to hit the seed S.

[12] Negated because we added the extra NAND gates on the input wires acting as negations.

[13] He knows the $\Delta^{(w)}$, picks one which is not 0 and outputs $(-1)^r \Delta^{(w)}$ for a random bit r.

References

[AGHP92] Noga Alon, Oded Goldreich, Johan Håstad, and René Peralta. Simple construction of almost k-wise independent random variables. *Random Struct. Algorithms*, 3(3):289–304, 1992.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, pages 1–10. ACM, 1988.

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

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *STOC*, pages 11–19, 1988.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229, 1987.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2003.

[JS07] Stanislaw Jarecki and Vitaly Shmatikov. Efficient two-party secure computation on committed inputs. In Moni Naor, editor, *EUROCRYPT*, volume 4515 of *Lecture Notes in Computer Science*, pages 97–114. Springer, 2007.

[LP04] Yehuda Lindell and Benny Pinkas. A proof of yao's protocol for secure two-party computation. *Electronic Colloquium on Computational Complexity (ECCC)*, (063), 2004.

[LP07] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *EUROCRYPT*, pages 52–78, 2007.

[LPS08] Yehuda Lindell, Benny Pinkas, and Nigel P. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In Rafail Ostrovsky, Roberto De Prisco, and Ivan Visconti, editors, *SCN*, volume 5229 of *Lecture Notes in Computer Science*, pages 2–20. Springer, 2008.

[Woo07] David P. Woodruff. Revisiting the efficiency of malicious two-party computation. In *EUROCRYPT*, pages 79–96, 2007.

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

A Proof of Lemma 1

For the second variant of the game the player picks $B = (2\ell+1)L$ balls. Consider a player picking β_2 red balls. It is let into the second phase with probability $2\gamma_2^{\beta_2}$. Now the balls are distributed at random into the buckets under the condition that each bucket contains exactly $2\ell+1$ balls. Let ρ denote the probability that some bucket contains at least $\ell+1$ red balls. We assume that $\beta_2 \geq \ell+1$ (or $\rho = 0$). It is then easy to see that the probability that at least $\ell+1$ balls in bucket j are red is no more than

$$\begin{aligned}
\sum_{i=\ell+1}^{2\ell+1} \binom{2\ell+1}{i} (\beta_2/B)^i &\leq \sum_{i=\ell+1}^{2\ell+1} \binom{2\ell+1}{i} (\beta_2/B)^{\ell+1} \\
&= (\beta_2/B)^{\ell+1} \sum_{i=\ell+1}^{2\ell+1} \binom{2\ell+1}{i} \\
&= (\beta_2/B)^{\ell+1} 2^{2\ell} \\
&= 2^{2\ell} (\beta_2/(2\ell+1))^{\ell+1} L^{-\ell-1} \ .
\end{aligned}$$

We use the union bound to get

$$\rho < L 2^{2\ell} (\beta_2/(2\ell+1))^{\ell+1} L^{-\ell-1} = 2^{2\ell} (\beta_2/(2\ell+1))^{\ell+1} L^{-\ell} \ .$$

The probability of winning is then

$$2\gamma_2^{\beta_2} \rho < 2\gamma_2^{\beta_2} 2^{2\ell} (\beta_2/(2\ell+1))^{\ell+1} L^{-\ell} \ .$$

It is easy to see that for fixed γ_2, L and ℓ the right-hand side is maximal when $\gamma_2^{\beta_2}\beta_2^{\ell+1}$ is maximal, which happens when $\beta_2 = -(\ell+1)(\ln \gamma_2)^{-1}$. We plug this into $2\gamma_2^{\beta_2} 2^{2\ell} (\beta_2/(2\ell+1))^{\ell+1} L^{-\ell}$ and get the following bound on winning

$$\begin{aligned}
2\gamma_2^{-(\ell+1)(\ln \gamma_2)^{-1}} & 2^{2\ell} (-(\ell+1)(\ln \gamma_2)^{-1}/(2\ell+1))^{\ell+1} L^{-\ell} \\
&\leq e^{-(\ell+1)} 2^{2(\ell+1)} (-(2/3)(\ln \gamma_2)^{-1})^{\ell+1} L^{-\ell} \\
&\leq e^{-(\ell+1)} 2^{2(\ell+1)} (-(2/3)(\ln \gamma_2)^{-1})^{\ell+1} L^{-\ell} \\
&= (e^{-1} 4 (2/3)(\ln \gamma_2^{-1})^{-1})^{\ell+1} L^{-\ell} \\
&< (0.99(\ln \gamma_2^{-1})^{-1})^{\ell+1} L^{-\ell} \ .
\end{aligned}$$

Consider then a player picking β_1 red balls and playing the first variant of the game. It is let into the second phase with probability $\gamma_1^{\beta_1}$. Now the balls are distributed at random into the buckets under the condition that each bucket contains exactly $\ell+1$ balls. Let ρ denote the probability that some bucket contains only red balls. We assume that $\beta_1 \geq \ell+1$. It is easy to see that the probability that all $\ell+1$ balls in bucket j are red is no more than $(\beta_1/B)^{\ell+1} = (\beta_1/(\ell+1))^{\ell+1} L^{-\ell-1}$. We use the union bound to get $\rho < L(\beta_1/(\ell+1))^{\ell+1} L^{-\ell-1} = (\beta_1/(\ell+1))^{\ell+1} L^{-\ell}$. The probability of winning is then $2\gamma_1^{\beta_1}\rho < \gamma_1^{\beta_1} (\beta_1/(\ell+1))^{\ell+1} L^{-\ell}$. It is easy to see that for fixed γ_1, L and ℓ the right-hand side is maximal when $\beta_1 = -(\ell+1)(\ln \gamma_1)^{-1}$. We plug this into $2\gamma_1^{\beta_1} (\beta_1/(\ell+1))^{\ell+1} L^{-\ell}$ and get a maximum of $2e^{-\ell-1}(-(\ln \gamma_1)^{-1})^{\ell+1} L^{-\ell} = 2(-e^{-1}(\ln \gamma_1)^{-1})^{\ell+1} L^{-\ell} < 2(0.37(\ln \gamma_1^{-1})^{-1})^{\ell+1} L^{-\ell}$.

B Parameters choice

In Fig. 2 we show some possible values for $(\ell, \epsilon_1, \epsilon_2)$, for different security parameters s and circuit sizes L. In particular, we have chosen parameters that minimize the cost per gate – defined as 1 for any commitment or hash that needs to be transmitted per gate in the original circuit. The quantity we tried to minimize is therefore given by the cost expression

$$c = 5(\ell + 1)/(1 - \epsilon_1) + 3(2\ell + 1)/(1 - \epsilon_2) \ ,$$

under the constraint that $P \leq 2^{-s}$, with P as defined in Lemma 2. Every entry in the table is of the form $(\ell, \epsilon_1, \epsilon_2); c$.

L \ s	30	50	70
10^3	$(5, 0.15, 0.21); 77$	$(8, 0.13, 0.21); 117$	$(11, 0.13, 0.20); 156$
10^4	$(4, 0.07, 0.09); 57$	$(6, 0.08, 0.11); 82$	$(8, 0.08, 0.15); 109$
10^6	$(2, 0.17, 0.26); 39$	$(4, 0.03, 0.04); 54$	$(5, 0.05, 0.08); 68$
10^9	$(2, 0.01, 0.02); 31$	$(2, 0.18, 0.29); 40$	$(3, 0.05, 0.07); 44$
10^{12}	$(1, 0.08, 0.12); 22$	$(2, 0.01, 0.01); 31$	$(2, 0.18, 0.38); 43$

Figure 2. Parameters choice for different security parameters s and circuit sizes L, and resulting cost per gate.

To better understand the significance of the numbers in the table, it's useful to recall that in the standard (passive secure) Yao protocol the cost per gate is 4, while in the protocol from [LPS08] the cost per gate is $4s$, plus $O(s^2)$ commitments for every input gate.

C A Replicated Gate

Fig. 3 illustrates the structure of a replicated gate with key checks.

D Input/Output Structure

Fig. 4 illustrates how Alice and Bob provide their inputs to the garbled circuit, and how Alice gets her output.

E NOT gates for free

In Section 4.2 we noted that our keys are constructed in a way that allows to compute addition gates for free. During the circuit soldering we use this fact to align the output wires of two gates with the input wire of the next gate. This process of key alignment is described in Section 4.4. What we note here is that in fact we can compute for free any linear combination gate. Plugging this construction in the key alignment phase gives us the possibility to get NOT gates for free.

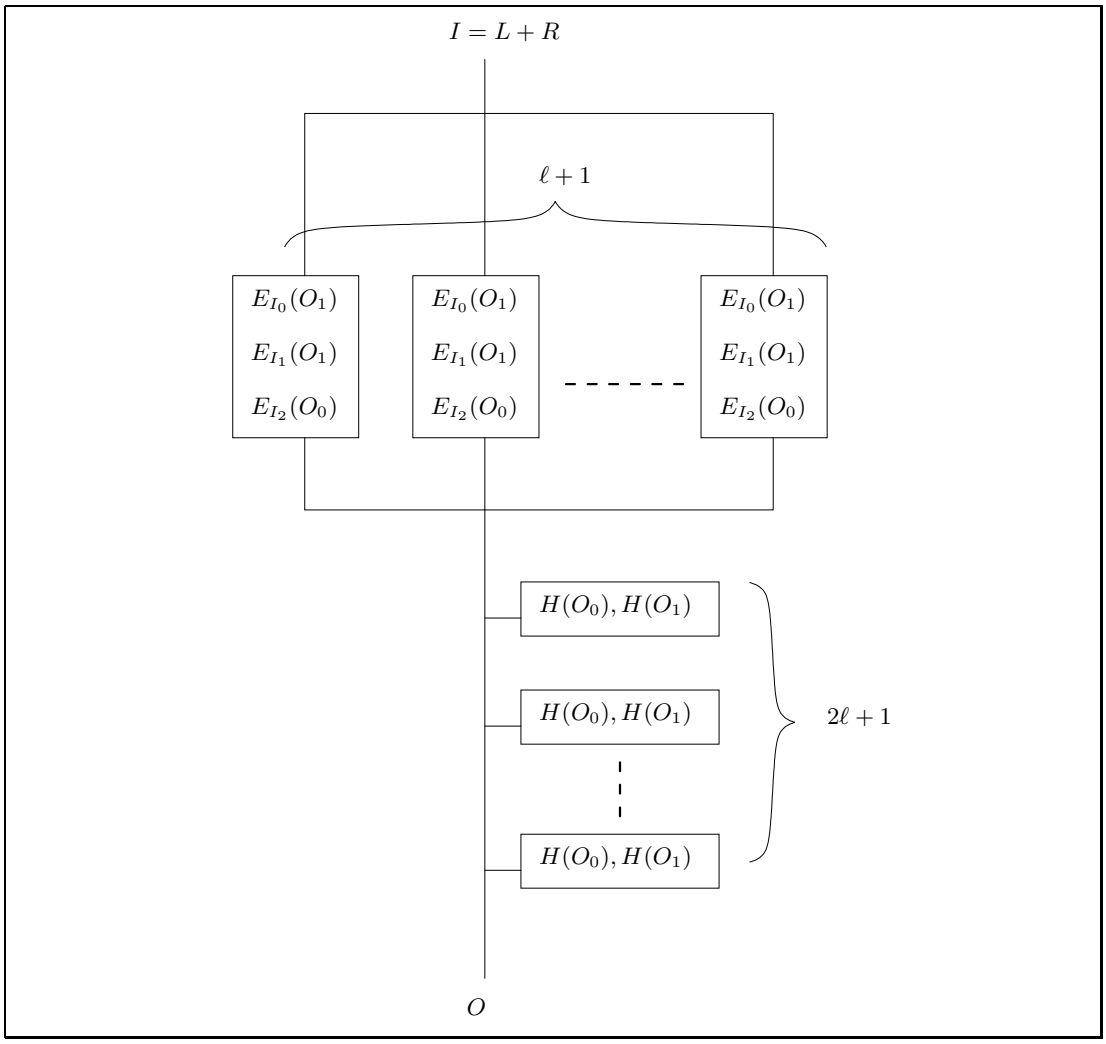

Figure 3. The alignment of $\ell+1$ NT gates and $2\ell+1$ KCs. Illustrated for the case where all components are correct.

In particular, if we have a set of keys $K^i_{x_i} = K^i_0 + x_i\Delta$, with $i \in \{1, n\}$, and some known integers $a_0, \ldots, a_n \in \mathbb{N}$, we can compute a key O_y with $y = \sum_{i=1}^n a_i x_i + a_0$ in the following way: Alice opens the value $S = \text{open}(\sum_{i=1}^n a_i[K^i_0] - a_0[\Delta] - [O_0])$, then Bob can compute $\sum_{i=1}^n a_i K^i_{x_i} - S = \sum_{i=1}^n a_i(K^i_0 + x_i\Delta) - \sum_{i=1}^n a_i K^i_0 + a_0\Delta + O_0 = O_y$.

A special case of this is when $n = 1$, $a_0 = 1, a_1 = -1$, in which case we get a NOT gate. When we align two output wires with an input wire we can negate any of the inputs by setting the three values (a_0, a_1, a_2) to: $(0, 1, 1)$ for straight connection, $(1, -1, 1)$ to negate the left input, $(1, 1, -1)$ to negate the right input and $(2, -1, -1)$ to negate both inputs.

F Sampling the Challenges using a Short Seed

To challenge Alice, Bob will for each component send a challenge $e \in \{0, \ldots, |E|\}$ with $e = |E|$ meaning that the component is not checked and $e < |E|$ meaning that part e of the generator must be shown.

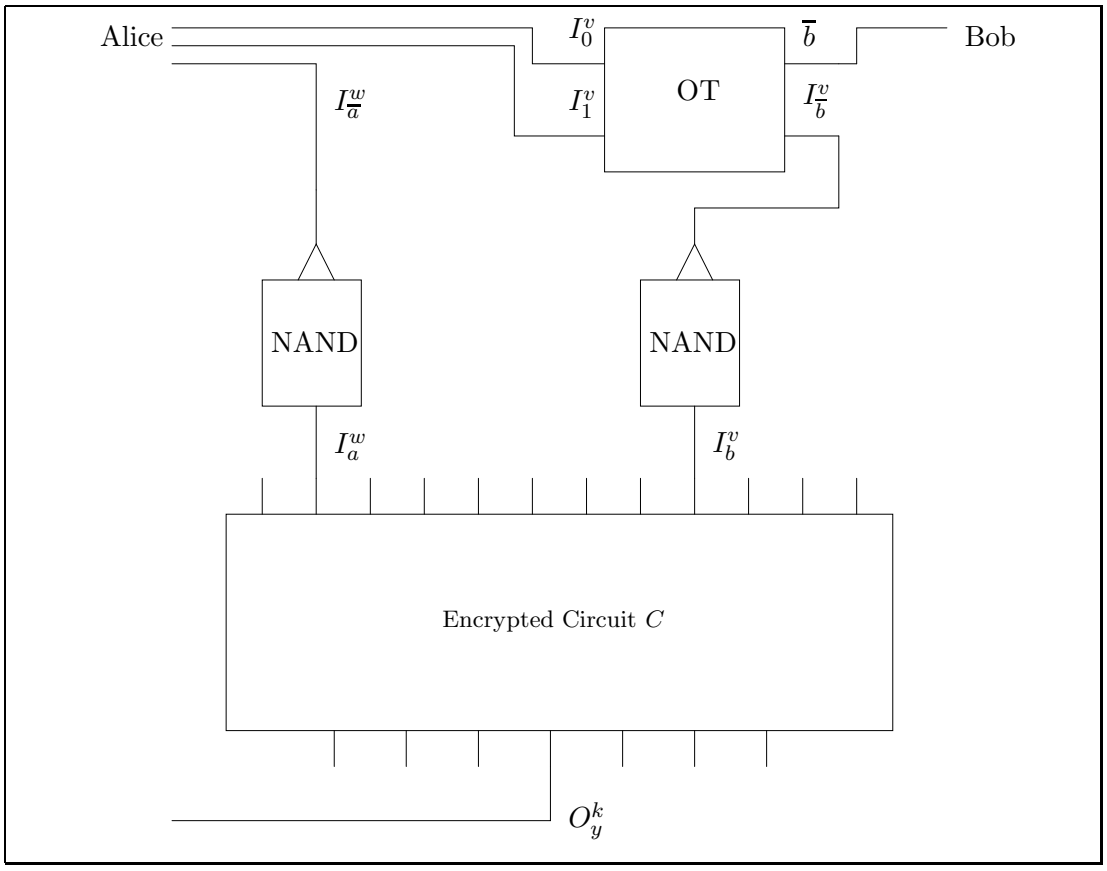

Figure 4. Alice inputs a bit a into the input wire I^i, while Bob inputs a bit b into the input wire I^j. After the evaluation, Alice gets the k-th output key, encoding the bit y.

Fix a subset I of the indices of the components and for each $i \in I$ pick a challenge $e_0^{(i)} \in \{0, \ldots, |E| - 1\}$. If we picked each $e^{(i)}$ independently with $\Pr[e^{(i)} = e_0] = \epsilon/|E|$ for $e_0 \in \{0, \ldots, |E| - 1\}$ and $\Pr[e^{(i)} = |E|] = 1 - \epsilon$, then we would have that

$$\Pr[\nexists i \in I(e^{(i)} = e_0^{(i)})] = (1 - \epsilon/|E|)^{|I|} \ .$$

For simplicity we assume that the distribution $(\epsilon/|E|, \epsilon/|E|, \ldots, \epsilon/|E|, 1 - \epsilon)$ can be sampled using a finite number of uniformly random bits, such that we can sample $e = \mathrm{e}(r)$ for a uniformly random $r \in \{0,1\}^c$ for a constant c. We can then sample all N elements $e^{(i)}$ from $r \in \{0,1\}^{cN}$. If we instead use r which is δ-close to uniform on $\{0,1\}^{cN}$, then

$$\Pr[\nexists i \in I(e^{(i)} = e_0^{(i)})] \leq (1 - \epsilon/|E|)^{|I|} + \delta \ . \tag{1}$$

In fact, it is enough that r is δ-almost k-wise independent on $\{0,1\}^{cN}$ with $k \geq c|I|$, as the $|I|$ values $e^{(i)}$ depend on at most $c|I|$ bits from r.

To sample r we use a δ-almost k-wise independent sampler space $\mathrm{sspc} : \{0,1\}^\sigma \to \{0,1\}^m$, where for a uniformly random *seed* $S \in \{0,1\}^\sigma$ the string r_I is δ-close to uniformly random, where $r = \mathrm{sspc}(S)$, $I \subset \{1, \ldots, m\}$, $|I| \leq k$ and r_I denotes the $|I|$-bit string consisting of r projected to coordinates $i \in I$. We can e.g. use a construction from [AGHP92], where

$$\sigma = 2(\log_2 \log_2 m - \log_2 \delta - \log_2 k - 1) \ .$$

We need $k = O(s)$ and $\delta = 2^{-O(s)}$, and can safely assume that the bit length of r is less that 2^{2^s}, giving us a seed length of $O(s)$, as desired.

The distribution of each $e^{(i)}$ is δ-close to the distribution $(\epsilon/|E|, \epsilon/|E|, \ldots, \epsilon/|E|, 1-\epsilon)$, so the expected number of i for which $e^{(i)} = |E|$ is at least $N(1-\epsilon) - N\delta$. We always need to be left with $N(1-\epsilon)$ components. By picking $\delta \leq 2^{-s}$ and by starting with N any small constant fraction larger than now (like 10%), we can by Chebychev's inequality, ensure that the probability that we end up with $N(1-\epsilon)$ components not checked being at least $\frac{3}{4}$. This very soon holding for large enough L and still with (1) holding for all I with $|I| \leq k/c$ (we will soon fix k).

Now, let H denote the event that $\nexists i \in I(e^{(i)} = e_0^{(i)})$ and let E denote the event that we are left with enough components ($N(1-\epsilon)$). We have that $\Pr[H|E] \leq \Pr[H]/\Pr[E] \leq \frac{4}{3}\Pr[H] \leq \frac{4}{3}((1-\epsilon/|E|)^{|I|} + \delta)$.

For any $\beta = O(s)$, let $k = \beta c$ and $\delta = \min(2^{-s}, 2^{-\beta-2})$. Then for all I with $|I| \leq \beta$ we have that

$$\Pr[\nexists i \in I(e^{(i)} \neq e_0^{(i)})] \leq \frac{4}{3}(1-\epsilon/|E|)^{|I|} + 2^{-\beta-1} \ . \tag{2}$$

By sampling S until it occurs that there are enough components left, we get enough components by using an expected $\frac{4}{3}$ tries. Formally the UC model does not allow expected poly-time protocols. We can handle this by terminating with some fixed C and $e^{(i)}$ if $\beta + 1$ tries failed. This occurs with probability at most $2^{-\beta_i - 1}$. Since $(\frac{4}{3}(1-\epsilon/|E|)^{|I|} + 2^{-\beta-1}) + 2^{-\beta-1} = \frac{4}{3}(1-\epsilon/|E|)^{|I|} + 2^{-\beta}$ we get that

$$\Pr[\nexists i \in I(e^{(i)} \neq e_0^{(i)})] \leq \frac{4}{3}(1-\epsilon/|E|)^{|I|} + 2^{-\beta} \ , \tag{3}$$

as desired.

G How to Construct the Circuit

In Fig. 5 we implement the 16 binary Boolean functions using nt() gates and linear computation, achievable for free as explained in Appendix E. In this way we can implement any of the Boolean functions using just one nt() gate.

H Coin Flipping via OT

During the protocol we need to sample some random strings in a secure way. In particular we need to sample a random element h in the group G and the seed S for the s-biased source of Appendix F.

Given that we are in the OT-hybrid model, we will present a way to reduce UC coin flipping to UC OT. The protocol allows the simulator to arbitrarily select the outcome of the coin flip.

The protocol is the following: To flip a random bit r Alice transfers two random strings $A_0 \neq A_1 \in_R \{0, 2^s - 1\}$. Bob retrieves one of the two strings at random, say he picks $b \in_R \{0, 1\}$ and he gets A_b. After the OT Alice sends a random bit a. Finally Bob sends Alice the retrieved string A_b. The outcome of the protocol is the random bit $r = a \oplus b$. The protocol is intuitively secure, as the probability that Bob guesses A_{1-b} is negligible in s, and Alice gets no information at all about Bob's bit b in the OT-hybrid model.

To see that the protocol is UC secure, consider the following simulator: when Alice is corrupted, the simulator gets both A_0, A_1 from the OT. Suppose that the simulator

x	0 0 1 1		
y	0 1 0 1	**Gate**	**Circuit**
	0 0 0 0	0	—
	0 0 0 1	AND	$1 - \mathrm{nt}(x + y)$
	0 0 1 0	$x > y$	$1 - \mathrm{nt}(1 + x - y)$
	0 0 1 1	x	x
	0 1 0 0	$x < y$	$1 - \mathrm{nt}(1 - x + y)$
	0 1 0 1	y	y
	0 1 1 0	XOR	$x + y + 2\,\mathrm{nt}(x + y) - 2$
	0 1 1 1	OR	$\mathrm{nt}(2 - x - y)$
	1 0 0 0	NOR	$1 - \mathrm{nt}(2 - x - y)$
	1 0 0 1	NXOR	$3 - x - y - 2\,\mathrm{nt}(x + y)$
	1 0 1 0	NOT y	$1 - y$
	1 0 1 1	$x \geq y$	$\mathrm{nt}(1 - x + y)$
	1 1 0 0	NOT x	$1 - x$
	1 1 0 1	$x \leq y$	$\mathrm{nt}(1 + x - y)$
	1 1 1 0	NAND	$\mathrm{nt}(x + y)$
	1 1 1 1	1	—

Figure 5. How to implement each of the binary Boolean functions using one NT gate.

wants to force the output to be r': The simulator waits for Alice to send a and then he replies $b = a \oplus r'$. If Bob is malicious the simulator simply gets Bob's choice b from the OT, and he replies with $a = b \oplus r'$.

This protocol is composable, so if the parties need to sample a k-long bit-string, they just need to run the protocol k time in parallel.

I Proof of Knowledge via OT

During the protocol Alice needs to prove in zero-knowledge that she knows an opening of the commitment $[\Delta; r_\Delta]$. She doesn't know the commitment trapdoor, so this defines a value for Δ.

Given that we are in the OT-hybrid model, we will present a way to reduce UC zero-knowledge proof of knowledge to UC OT. The protocol allows the simulator to extract Δ.

The protocol is the following: Alice picks at random $K_0, r_0 \in_R \mathbb{Z}_p$. Define $K_1 = K_0 + \Delta, r_1 = r_0 + r_\Delta$. Then Alice offers $((K_0, r_0), (K_1, r_1))$ to the OT. Bob chooses a random bit $e \in_R \{0, 1\}$ and accepts if K_e, r_e is an opening of $[K_0] + e[\Delta]$. They repeat the protocol s times in parallel. Therefore, Alice cannot guess Bob's choice with probability better than 2^{-s}. Given that we are in the OT-hybrid model, If Alice is corrupted the simulator gets to see K_0, K_1, and therefore it can compute $\Delta = K_1 - K_0$.

Note that this protocol is essentially a copy of the one that Alice and Bob run when Bob needs to retrieve his keys. The main difference is that here Bob chooses his selection bits at random, and therefore Alice cannot guess them with non-negligible probability, while during the protocol we cannot exclude that Alice knows Bob's input. We can exploit it to reduce the number of needed OTs: Alice and Bob run the proof of knowledge, where Bob uses random selection bits. At the end of the protocol, Bob

saves the tuples $(e_i, K^i_{e_i}, [K^i_0])$ he gets during the protocol. Later, when he needs an input key $I^{(g)}_b$ for an input gate g, he can 'recycle' those keys permuting them randomly and aligning them to the input wires. As shown in Appendix E this can be done even if $b \neq e_i$, using the NOT gate for free construction.

J Extracting Δ

We describe here what Bob should do if, for some internal gate g, he gets a set of keys $\mathcal{O}^{(g)}$ s.t $|\mathcal{O}^{(g)}| > 1$. In this case, except with negligible probability, $\mathcal{O}^{(g)} = \{O^{(g)}_0, O^{(g)}_1\}$. We call $O^{(g)}_d, O^{(g)}_{1-d}$ those two keys, as Bob doesn't know which key is which. Clearly, $\Delta = (-1)^d(O^{(g)}_{1-d} - O^{(g)}_d)$, and therefore Bob needs to find d in order to find the right value of Δ. The value d is found as follows: For each input gate g' where Bob is supposed to provide input, Bob takes the key $O^{(g')}_{b_i}$, which at this point we can assume correct. Then he uses his input bit b_i and the $\ell + 1$ KCs of g' to determine the value of d: He checks if there is a hash of $O^{(g')}_{b_i} + (-1)^{b_i}(O_{1-d} - O_d)$ or $O^{(g')}_{b_i} + (-1)^{b_i}(O_d - O_{1-d})$ and set $d = 0$ respectively $d = 1$ and he takes the majority vote.

K Extracting Most Component Generators with High Probability

We now describe an extractor which allows to extract generators for most components when Bob is honest and accepts the checks. We note that the extractor rewinds the environment. Therefore the extractor is not a sub-routine which can be run by the simulator in a proof in the UC model. This is not a problem, as this is not the intended use. Our UC simulator does not run this extraction. The fact that some imaginary algorithm X *could* have rewound the environment and the entire execution to produce generators will be used to *analyze* the simulator, not construct it.

In a bit more details, if Alice is corrupted and Bob is honest, then for all adversaries and all environments we consider an extractor X which can be run on the terminal global state of the environment, the adversary and the execution of the protocol to try to extract components generators for the components used by Alice and Bob. By the *augmented execution* we mean: First run the protocol with the given environment and adversary until it terminates, and then *if* Bob accepts the checks, *then* run the extractor on the terminal global state of the environment, the adversary and the execution of the protocol. The extractor X has the following properties:

- The augmented execution is expected poly-time.
- The extractor in the augmented execution computes generators for all components except a few, where the exactly number of missing components relates to the probability the Bob accepts.

We start by some technical definitions and lemmas.

Definition 1. *Given a poly-time binary relation R we call $(E, \text{reply}, \text{accept})$ a simple proof of knowledge for R if E is a finite set and $\text{accept}(x, e, \text{reply}(w, e)) = 1$ for all $(x, w) \in R$ and $e \in E$, and if given z_e such that $\text{accept}(x, e, z_e) = 1$ for all $e \in E$ one can in poly-time compute w such that $(x, w) \in R$, except with negligible probability.*

The check we do for NT gates and KCs are clearly simple proofs of knowledge (see Appendix L), with the witness being the generator of the components and the relation being that the generator gives rise to the component.

For a simple proof of knowledge we are interested in the following protocol for generating proved instances, parametrized by some constant $\epsilon \in (0,1)$ and integers $\delta, L \in \mathbb{N}$.

1. The prover P sends L instances $x^1, \ldots, x^L$ for which it knows witnesses $w^1, \ldots, w^L$.
2. The verifier V selects a random subset C of the instances, and sends $e^i = |E|$ for $i \notin C$. For each $i \in C$, V in addition sends a random challenge $e^i \in E = \{0, \ldots, |E|-1\}$.
3. For each $e^i \neq |E|$, P sends $z^i = \text{reply}(w^i, e^i)$ to P.
4. V checks that $\text{verify}(x^i, e, z^i) = 1$ for all $e^i \neq |E|$. If so, the output is $(\texttt{accept!}, \{x^i\}_{e^i=|E|})$. Otherwise, the output is `reject!`.

We pick the challenges e^i such that for any fixed subset I with $|I| \leq \delta$ and for fixed challenges $e_0^i \in E$ for $i \in I$, it holds that

$$\Pr[\nexists i \in I(e^i \neq e_0^i)] \leq \frac{4}{3}(1 - \epsilon/|E|)^{|I|} + 2^{-\delta} \ .$$

For each $\beta \in \{1, \ldots, \delta - 3\}$ we describe an extractor X_β. It can be run on a state of the protocol execution right after the verifier accepted the checks, and it will try to extract witnesses for $L - \beta$ of the instances.

1. For each x^i and each $e \in E$ for which no correct reply z_e^i is stored: rewind the execution to where the challenges are sent and send a random seed S giving rise to $e^i = e$, run the execution to receive some z_e^i, and store it if it is correct. Note the an independent rewind and rerun is used for each such (x^i, e), meaning that the step can include as many as $L|E|$ reruns.
2. If for all but β of the instances correct replies z_e^i are stored for all $e \in E$, then compute witnesses for all these $L - \beta$ instances and terminate. Otherwise, go to the above step.

By the *β-augmented protocol* we mean the following: First run the protocol until V accepts or rejects. If V rejected then stop. Otherwise, run X_β on the trace of the execution to extract $L - \beta$ witnesses.

Lemma 2. *If $\beta \leq \delta - 3$ and the probability that V accepts the proof is $\geq 2(1 - \epsilon|E|^{-1})^\beta$, then the expected running time of the β-augmented protocol is polynomial.*

Proof. Fix a state of the protocol execution right after Step 1 is executed, i.e., right after the instances are sent. We make some definitions relative to such a state.

For an index $i \in \{1, \ldots, L\}$ and a challenge $e \in E$ we define $\text{Accept}(i, e) \in [0, 1]$ to be the probability that the prover replies with a correct z^i if the verifier sends a random seed S giving rise to $e^i = e$. For each i we let ϵ^i be an element $e \in E$ which minimizes $\text{Accept}(i, e)$ and we let $\text{Accept}(i) = \text{Accept}(i, \epsilon^i)$.

For each $\beta \in \{1, \ldots, L\}$ we can pick A_β such that there exists $\mathcal{A}_\beta \subseteq \{1, \ldots, L\}$ for which $|\mathcal{A}_\beta| = L - \beta$ and $\text{Accept}(i) \geq A_\beta$ for $i \in \mathcal{A}_\beta$ and $\text{Accept}(i) \leq A_\beta$ for $i \notin \mathcal{A}_\beta$. Think of A_β as the largest accept probability we can pick such that at most β instances have a challenge which is answered correctly with smaller probability. We call these β ways of challenging the *good challenges* (for the verifier)— as they are the hardest to answer for the prover.

We first compute the probability a that the check accepts, expressed via A_β. We say that the value e^i sent by V *hits* (a good challenge) if $e^i = \epsilon^i$ for some $i \notin \mathcal{A}_\beta$. Since $|\mathcal{A}_\beta| = L - \beta$ and $\beta \leq \delta$, the probability that there is no hit is at most

$$\bar{h} = \frac{4}{3}(1 - \epsilon|E|^{-1})^\beta + 2^{-\delta} \ .$$

If there is no hit, then the check might accept with probability 1. If there is a hit, then the check accepts with probability $\leq A_\beta$. I.e.,

$$a \leq \bar{h} \cdot 1 + (1 - \bar{h})A_\beta \leq \bar{h} + A_\beta \ . \tag{4}$$

We estimate the expected running time of A_β *given* that the check succeeds. Consider a given index $i \in \mathcal{A}_\beta$ and any $e \in E$. Since $\text{Accept}(i, e) \geq A_\beta$, the expected number of rounds until a correct z_e^i is stored is A_β^{-1}. There are $(L-\beta)|E|$ such pairs (i, e). Therefore the expected number of rounds until a correct z_e^i is stored for all of them is at most $A_\beta^{-1} \log((L - \beta)|E|) = A_\beta^{-1}\,\text{poly}$. Each round is poly. So, the expected running time of X_β given that the test succeeds is $A_\beta^{-1}\,\text{poly}$.

The running time of the augmented execution is poly when the check fails, as then no extraction is run. When the test succeeds, it is poly (for running the protocol) plus the expected running time of the extraction. I.e.,

$$T \leq (1 - a)\,\text{poly} + a(\text{poly} + A_\beta^{-1}\,\text{poly}) \leq \text{poly}(1 + aA_\beta^{-1}) \ .$$

For T to be poly it is therefore sufficient that $a \leq \alpha A_\beta$ for $\alpha = 1 + \frac{1}{4}$, for which (by (4)) it is sufficient that $A_\beta \geq (\alpha - 1)\bar{h}$. So, assume that $A_\beta < (\alpha - 1)\bar{h}$. Then it follows from $a \leq \bar{h} + A_\beta$ that

$$a \leq \alpha\bar{h} \leq \alpha\frac{4}{3}(1 - \epsilon|E|^{-1})^\beta + \alpha \cdot 2^{-\delta} \ .$$

We have $\delta \geq \beta - 3$ and $(1 - \epsilon|E|^{-1}) \geq \frac{1}{2}$, meaning that $\alpha \cdot 2^{-\delta} \leq \alpha \cdot 2^{-3}2^{-\beta} \leq 2^{-2}(1 - \epsilon|E|^{-1})^\beta$. So,

$$a \leq \left((1 + \frac{1}{4})\frac{4}{3} + 2^{-2}\right)\left(1 - \epsilon|E|^{-1}\right)^\beta = \frac{23}{12}\left(1 - \epsilon|E|^{-1}\right)^\beta < 2\left(1 - \epsilon|E|^{-1}\right)^\beta \ .$$

This contradicts the premise of the lemma that $a \geq 2(1 - \epsilon|E|^{-1})^\beta$.

L Extraction and Simulation of Components

For the checks of the components we will argue that the check is *extraction complete* in the sense that giving correct replies to all $e \in E$ will allow Bob to either: 1) compute a correct generator for the component, 2) compute a double opening of the commitment scheme $[\cdot;\cdot]$, or 3) compute a collision for the hash function H. Since Alice is assumed to not be able to break the commitment scheme or H, this will be sufficient for later use. We also argue that the checks are *simulatable* by Alice without knowing Δ in the following sense: knowing the trapdoor of the commitment scheme and the challenge e, Alice can compute a component for which she can correctly reply to the challenge e. The simulated components and reply are computationally indistinguishable from a real component and a real reply.

L.1 NT gates

Extraction Complete: We argue that the three tests are extraction complete. For this purpose, assume that Alice can answer all three challenges for $([I_0], [O_0], C_0, C_1, C_2)$.

The answer to $e = 0$ gives an opening of $[I_0]$ to some I_0 and an opening of $[O_0] + [\Delta]$ to some O_1.

The answer to $e = 1$ gives an opening of $[I_0] + [\Delta]$ to some I_1 and an opening of $[O_0] + [\Delta]$ to some O_1. From the opening of $[I_0]$ to I_0 (from the reply to $e = 0$) and the opening of $[\Delta]$ to Δ we can compute an opening of $[I_0] + [\Delta]$ to $I_0 + \Delta$. So, if $I_1 \neq I_0 + \Delta$ we can compute a double opening of $[I_0] + [\Delta]$. We can therefore assume that $I_1 = I_0 + \Delta$. Using a similar argument we can assume that O_1 equals the O_1 from the reply to $e = 0$.

The answer to $e = 2$ gives an opening of $[I_0] + 2[\Delta]$ to some I_2 and an opening of $[O_0]$ to some O_0. Using arguments as above we can conclude that $I_2 = I_0 + 2\Delta$ and $O_1 = O_0 + \Delta$ or that a double opening can be computed efficiently.

From the other parts of the checks we get that the obtained keys fulfill that $C_{\pi_0} = E_{I_0}(O_1)$, $C_{\pi_1} = E_{I_0+\Delta}(O_1)$, $C_{\pi_2} = E_{I_0+2\Delta}(O_0)$ for some π_0, π_1, π_2. Let $\pi(i) = \pi_i$ for $i \in \{0, 1, 2\}$. Then $(I_0, r_I, O_0, r_O, \pi)$ is a generator, unless π_0, π_1, π_2 are not distinct so that π is not a bijection. Below we will not assume that the extracted witnesses have π which are bijections.

Simulatable: We show how to simulate for $e = 2$. The other two cases are handled using the same technique. Alice picks uniformly random keys I_2 and O_0. Then she defines $I_1 = I_2 - \Delta$ and $I_0 = I_2 - 2\Delta$ and $O_1 = O_0 + \Delta$.[14] She picks π at random and randomizers r'_I and r_O and sends $([I_0], [O_0], C_0, C_1, C_2)$, where $[I_0] = [I_2; r'_I] - 2[\Delta]$, $[O_0] = [O_0; r_O]$ and $C_{\pi(2)} = E_{I_2}(O_0)$. She lets $C_{\pi(0)}$ and $C_{\pi(1)}$ be uniformly random group elements. On the challenge $e = 2$ she has to open $[I_0] + 2[\Delta] = [I_2; r_I]$ and $[O_0] = [O_0; r_0]$ to values such that $C_{\pi(2)} = E_{I_2}(O_0)$, which is possible by construction. The simulation is computationally indistinguishable from the protocol by our assumptions on the encryption scheme.

L.2 KCs

Extraction Complete: We argue that the two tests are extraction complete. For this purpose, assume that Alice can answer both challenges for $([K_0], T_0, T_1)$.

As for the NT gates we can assume that $K_1 = K_0 + \Delta$, as we would otherwise be able to compute a double opening of the commitment scheme. It follows that $T_{\pi_0} = H(K_0)$ and $T_{1-\pi_1} = H(K_0 + \Delta)$ for some π_0, π_1. If $\pi_0 = \pi_1$, then $H(K_0) = H(K_0 + \Delta)$, which gives a collision. If $\pi_0 \neq \pi_1$, then (K_0, r_K, π_0) is a generator.

Simulation This is handled as for NT gates. Here we use that $H(V_1 + \Delta), \ldots, H(V_m + \Delta)$ are computationally indistinguishable from uniformly random values given $V_1, \ldots, V_m$ when the V_i are distinct and Δ is uniformly random.

[14] *Defines*, as she does not know Δ.