J. Cryptol. (2012) 25: 680–722
DOI: 10.1007/s00145-011-9107-0

Journal of
CRYPTOLOGY

Secure Two-Party Computation via Cut-and-Choose Oblivious Transfer*

Yehuda Lindell and Benny Pinkas
Dept. of Computer Science, Bar-Ilan University, Ramat-Gan, Israel
lindell@cs.biu.ac.il; benny@pinkas.net

Communicated by Hugo Krawczyk

Received 15 March 2011
Online publication 3 September 2011

Abstract. Protocols for secure two-party computation enable a pair of parties to compute a function of their inputs while preserving security properties such as privacy, correctness and independence of inputs. Recently, a number of protocols have been proposed for the efficient construction of two-party computation secure in the presence of malicious adversaries (where security is proven under the standard simulation-based ideal/real model paradigm for defining security). In this paper, we present a protocol for this task that follows the methodology of using cut-and-choose to boost Yao's protocol to be secure in the presence of malicious adversaries. Relying on specific assumptions (DDH), we construct a protocol that is significantly more efficient and far simpler than the protocol of Lindell and Pinkas (Eurocrypt 2007) that follows the same methodology. We provide an exact, concrete analysis of the efficiency of our scheme and demonstrate that (at least for not very small circuits) our protocol is more efficient than any other known today.

Key words. Secure two-party computation, Malicious adversaries, Cut-and-choose, Concrete efficiency.

1. Introduction

1.1. *Background*

Protocols for secure two-party computation enable a pair of parties P_1 and P_2 with private inputs x and y, respectively, to compute a function f of their inputs while preserving a number of security properties. The most central of these properties are *privacy* (meaning that the parties learn the output $f(x, y)$ but nothing else), *correctness* (meaning that the output received is indeed $f(x, y)$ and not something else), and *independence of inputs* (meaning that neither party can choose its input as a function of the other party's input). The standard way of formalizing these security properties is to compare the output of a real protocol execution to an "ideal execution" in which the

* An extended abstract of this work appeared at *TCC 2011*.

parties send their inputs to an incorruptible trusted party who computes the output for the parties. Informally speaking, a protocol is then secure if no real adversary attacking the real protocol can do more harm than an ideal adversary (or simulator) who interacts in the ideal model [2,3,14,15,32]. An important parameter when considering this problem relates to the power of the adversary. The two most studied models are the *semi-honest model* (where the adversary follows the protocol specification exactly but tries to learn more than it should by inspecting the protocol transcript) and the *malicious model* (where the adversary can follow any arbitrary polynomial-time strategy).

In the 1980s powerful feasibility results were proven, showing that *any* probabilistic polynomial-time two-party functionality can be securely computed in the presence of semi-honest adversaries [41] and in the presence of malicious adversaries [14]. These results showed that it is possible to achieve such secure protocols, but did not demonstrate how to do so efficiently (where by efficiency we mean a protocol that can be implemented and run in practice). To be more exact, the protocol of [41] for semi-honest adversaries is efficient. However, achieving security efficiently for the case of malicious adversaries is far more difficult. In fact, until recently, no efficient general protocols were known at all, where a general protocol is one that can be used for computing *any* functionality.

This situation has changed in the past few years, possibly due to increasing interest from outside the cryptographic community in secure protocols that are efficient enough to be used in practice. The result has been that a number of secure two-party protocols were presented that are secure in the presence of malicious adversaries, where security is rigorously proven according to the aforementioned ideal/real model paradigm [21, 23,28,35]. Interestingly, these protocols all take novel, different approaches and so the secure-protocol skyline is more diverse than before, providing the potential for taking the protocols a step closer to very high efficiency. These protocols are discussed in more detail in Sect. 1.3.

We remark that the protocol of [28] has been implemented for the non-trivial problem of securely computing the AES block cipher (pseudorandom function), where one party's input is a secret key and the other party's input is a value to be "encrypted" [38]. A Boolean circuit for computing this function was designed with approximately 33,000 gates, and the protocol of [28] was implemented for this circuit. Experiments showed that the running-time of the protocol was between 18 and 40 minutes, depending on the assumptions taken on the primitives used to implement the protocol. Although this is quite a long time, for some applications it can be reasonable. In addition, it demonstrates that it is *possible* to securely compute functions with large circuits, and motivates the search for finding even more efficient protocols that can widen the applicability of such computations in real-world settings.

1.2. *Our Results*

In this paper, we follow the construction paradigm of [28] and significantly simplify and improve the efficiency of their construction. The approach of [28] is to carry out a basic cut-and-choose on the garbled circuit construction of Yao [41]. (See also [33] for a different protocol based on the cut-and-choose approach, but which does not provide security according to the standard ideal/real model paradigm.) We assume familiarity with Yao's protocol and refer to Appendix A for those not familiar. The cut-and-choose

approach works by party P_1 constructing s copies of a garbled circuit and sending them to P_2, who then asks P_1 to open half of them in order to verify that they are correctly constructed. If all of the opened circuits are indeed correct, then it is guaranteed that a *majority* of the unopened half are also correct, except with probability that is negligible in s.[1] Thus, P_1 and P_2 evaluate the remaining $s/2$ circuits, and P_2 takes the output that appears in most of the evaluated circuits. As discussed in [28], P_2 cannot abort in the case that not all of the $s/2$ circuits evaluate to the same value, even though in such a case it knows that P_1 is cheating. The reason for this is that P_1 may construct a circuit that computes f in the case that P_2's first bit equals 0, and otherwise it outputs random garbage. Now, with probability $1/2$ this faulty circuit is not opened and so is one of the circuits to be evaluated. In this case, if P_2 would abort when it saw random garbage then P_1 would know that P_2's first input bit equals 1. For this reason, P_2 takes the majority output and ignores minority values without aborting.

Although intuitively appealing, the cut-and-choose approach introduces a number of difficulties which significantly affect the efficiency of the protocol of [28]. First, since the parties need to evaluate $s/2$ circuits rather than one, there needs to be a mechanism to ensure that they use the same input in all evaluations (the solution for this for P_2's inputs is easy, but for P_1's inputs turns out to be hard). The mechanism used in [28] required constructing and sending $2s^2\ell$ commitments, where ℓ is the length of P_2's input. In the implementation by [38], they used $s = 160$ and $\ell = 128$. Thus, the overhead due to these consistency proofs alone is the computation and transmission of 6,553,600 commitments! Another problem that arises in the cut-and-choose approach is that a malicious P_1 can input an incorrect key into one of the oblivious transfers used for P_2 to obtain the keys associated with its input wires in the garbled circuit. For example, it can set all the keys associated with 0 for P_2's first input bit to be garbage, thereby making it impossible for P_2 to decrypt any circuit if its first input bit indeed equals 0. In contrast, P_1 can make all of the other keys be correct. In this case, P_1 is able to learn P_2's first input bit, merely by whether P_2 obtains an output or not. The important observation is that the checks on the garbled circuit carried out by P_2 do not detect this because there is a separation between the cut-and-choose checks and the oblivious transfer. This adversarial strategy is called a selective failure attack [24]. The solution to this problem in [28] requires making the circuit larger and significantly increasing the size of the inputs by replacing each input bit with the exclusive-or of multiple random input bits. Finally, the analysis of [28] yields an error of $2^{-s/17}$. Thus, in order to obtain an error level of 2^{-40} the parties need to exchange 680 circuits. We remark that it has been conjectured in [38] that the true error level of the protocol is $2^{-s/4}$; however, this has not been proven.

Our Protocol We solve the aforementioned problems in a way that is far simpler and far more efficient than in [28]. In addition, we reduce the error probability to $2^{-0.311s}$, for large enough s. A concrete calculation yields that for an error of 2^{-40} it suffices to send only 132 circuits. This is an important improvement because the experiments

[1] The parameter s is a statistical security parameter, and it models the negligible probability that the adversary is not caught in cut-and-choose type checks. Typically, this negligible probability is exponentially small, and the exact constant in the exponent has a significant ramification on the efficiency of the protocol, because it influences how many garbled circuits need to be sent in order to obtain a small enough error.

of [38] demonstrate that the bottleneck in efficiency is not the exponentiations, but rather the number of circuits and the commitments for proving consistency. Thus, in our protocol we moderately increase the number of exponentiations, while reducing the number of circuits, completely removing the commitments, and also removing the need to increase the size of the inputs. We remark that the price for these improvements is that our protocol relies heavily on the decisional Diffie–Hellman (DDH) assumption, while the protocol of [28] used general assumptions only. We now proceed to describe our two main techniques:

1. Our solution for ensuring consistency of P_1's inputs is to have P_1 determine the keys associated with its own input bits via a Diffie–Hellman pseudorandom synthesizer [34]. That is, P_1 chooses values $g^{a_1^0}, g^{a_1^1}, \ldots, g^{a_\ell^0}, g^{a_\ell^1}$ and $g^{r_1}, \ldots, g^{r_s}$ and then sets the keys associated with its ith input bit in the jth circuit to be $g^{a_i^0 \cdot r_j}, g^{a_i^1 \cdot r_j}$. Given all of the $\{g^{a_i^0}, g^{a_i^1}, g^{r_j}\}$ values and any subset of keys of P_1's input generated in this way, the remaining keys associated with its input are pseudorandom by the DDH assumption. Furthermore, it is possible for P_1 to efficiently prove that it used the same input in all circuits when the keys have this nice structure. We stress that the garbled values for the rest of the circuit are chosen as usual. Thus, it is still possible to use garbled-circuit optimizations like that presented in [25].
2. As we have described, the reason that the inputs and circuits were needed to be made larger in [28] is due to the fact that the cut-and-choose circuit checks were separated from the oblivious transfer. In order to solve this problem, we introduce a new primitive called *cut-and-choose oblivious transfer*. This is an ordinary oblivious transfer [11,39] with the sender inputting many pairs $(x_1^0, x_1^1), \ldots, (x_s^0, x_s^1)$, and the receiver inputting many bits $\sigma_1, \ldots, \sigma_s$. However, the receiver also inputs a set $\mathcal{J} \subset [s]$ of size exactly $s/2$. Then, the receiver obtains $x_i^{\sigma_i}$ for every i (as in a regular oblivious transfer) along with *both values* (x_j^0, x_j^1) for every $j \in \mathcal{J}$, while the sender learns nothing about $\sigma_1, \ldots, \sigma_s$ and $\mathcal{J}$. The use of this primitive in our protocol intertwines the oblivious transfer and the circuit checks and solves the aforementioned problem. We also show how to implement this primitive in a highly efficient way, under the DDH assumption. We believe that this primitive is of independent interest, and could be useful in many cut-and-choose scenarios.

Efficiency Analysis Our entire protocol, including all subprotocols, is explicitly written and analyzed in a concrete and exact way for efficiency. Considerable effort has been made to optimize the constructions and reduce the constants throughout. We believe that this is of great importance when the focus of a result is *efficiency*. See Sect. 1.3 for a summary of the exact complexity of our protocol, and Sect. 4.3 for a complete analysis, with optimizations in Sect. 4.4.

Variants Another advantage of our protocol over that of [28] is that we obtain a universally composable [4] variant that is only slightly less efficient than the stand-alone version. This is because our simulator only rewinds during zero-knowledge protocols. These protocols are also Σ protocols and so can be efficiently transformed into universally composable zero-knowledge. As with our basic protocol, we provide an explicit

description of this transformation and analyze its exact efficiency. In addition, we show how our protocol yields a more efficient construction for security in the presence of covert adversaries [1], when high values of the deterrent factor ϵ are desired.

1.3. *Comparison to Other Protocols*

We provide an analysis of the efficiency of recent protocols for secure two-party computation. Each protocol takes a different approach, and thus the approaches may yield more efficient instantiations in the future. Nevertheless, as we will show, our protocol is significantly more efficient than the current best instantiations of the other approaches (at least, for not very small circuits).

- **Committed input method (Jarecki-Shmatikov [23]):** The secure two-party protocol of [23] works by constructing a single circuit and proving that it is correct. The novelty of this protocol is that this can be done with only a constant number of (large modulus) exponentiations per gate of the circuit. Thus, for circuits that are relatively small, this can be very efficient. However, an exact count gives that approximately 720 exponentiations are required per gate. Thus, even for small circuits, this protocol is not yet practical. For large circuits like AES with 33,000 gates, the number of exponentiations is very large (23,760,000), and is not realistic. (The authors comment that if efficient batch proofs can be found for the languages they require then this can be significantly improved. However, to the best of our knowledge, no such improvements have yet been made.)
- **LEGO (Nielsen-Orlandi [35]):** The LEGO protocol [35] follows the cut-and-choose methodology in a completely different way. Specifically, the circuit constructor first sends the receiver many gates, and the receiver checks that they are correctly constructed by asking for some to be opened. After this stage, the parties interact in a way that enables the gates to be securely soldered (like Lego blocks) into a correct circuit. Since it is not guaranteed that all of the gates are correct, but just a vast majority, a fault tolerant circuit of size $O(s \cdot |C|/\log|C|)$ is constructed, where s is a statistical security parameter. The error as a function of s is 2^{-s} and the constant inside the "O" notation for the number of exponentiations is 32 [36]. Thus, for an error of 2^{-40} we find that the overall number of exponentiations carried out by the parties is $1280 \cdot |C|/\log|C|$. For large circuits, like that of AES, this is unlikely to be practical. (For example, for the AES circuit with 33,000 gates we see that the parties need to carry out 2,816,000 exponentiations. Observe that due to the size of the circuit, the $\log|C|$ factor is significant in making the protocol more efficient than [23], as predicted in [35]. This protocol also relies on the DDH assumption. It is worthy to note that exponentiations in this protocol are in a regular "Diffie–Hellman" group and so Elliptic curves can be used, in contrast to [23] who work in $\mathbb{Z}_N^*$.)
- **Virtual multiparty method (Ishai et al. [21,22]):** This method works by having the parties simulate a virtual multiparty protocol with an honest majority. The cost of the protocol essentially consists of the cost of running a semi-honest protocol for computing additive shares of the product of additive shares, for every multiplication carried out by a party in a multiparty protocol with honest majority. Thus, the actual efficiency of the protocol depends heavily on the multiparty protocol

to be simulated, and the semi-honest protocols used for simulating the multiparty protocol. An asymptotic analysis demonstrates that this method may be competitive. However, no concrete analysis has been carried out, and it is currently an open question whether or not it is possible to instantiate this protocol in a way that will be competitive with other known protocols.

- **Cut-and-choose on circuits (Lindell-Pinkas [28]):** Since this protocol has been discussed at length above, we just briefly recall that the complexity of the protocol is $O(\ell)$ oblivious transfers for input-length ℓ (where the constant inside here is not small because of the need to increase the number of P_2's inputs), and the construction and computation of s garbled circuits and of $2s^2\ell$ commitments. In addition, the proven error of the protocol is $2^{-s/17}$ and its conjectured error is $2^{-s/4}$. The actual error value has a significant impact on the efficiency.

In contrast to the above, the complexity of our protocol is as follows. The parties need to compute $15s\ell + 39\ell + 10s + 6$ exponentiations, where ℓ is the input length and s is a statistical security parameter discussed below. We further show that with optimizations the $15s\ell$ component can be brought down to just $5.66s\ell$ *full exponentiations*, and if preprocessing can be used then only $s\ell/2$ full exponentiations need to be computed after the inputs become known. (We remark that with preprocessing, the protocols of [35] and [21,22] are also much more efficient.) In addition, the protocol requires the exchange of $7s\ell + 22\ell + 7s + 5$ *group elements*, and has 12 *rounds of communication*. Finally, there are $6.5|C|s$ *symmetric encryptions* for constructing and decrypting the garbled circuits and $4|C|s$ ciphertexts sent for transmitting these circuits. An important factor here is the value of s needed. The error of our protocol is $2^{-0.311s}$ for large enough s; a concrete calculation yields that for an error of 2^{-40} it suffices to set $s = 132$. (The overhead of computing an AES circuit with $|C| = 33{,}000$, $\ell = 128$ and $s = 132$, is therefore about 96,000 exponentiations, 28,300,000 symmetric encryptions, and communicating 270 Mbytes, where about 95% of the communication is spent on sending the garbled circuits.) Finally, we stress also that all of our exponentiations are of the basic Diffie–Hellman type and so can be implemented over Elliptic curves, which is much cheaper than RSA-type operations.

2. Preliminaries and Definitions

Throughout the paper we denote the computational security parameter by n, the statistical security parameter by s, and the length of inputs by ℓ. The computational security parameter is the usual one that is used to model the security of the underlying computational assumptions (e.g., the DDH assumption). In contrast, the statistical security parameter models a probability of cheating that is not due to any computational hardness, but rather holds in an information-theoretic sense. For example, the probability that an adversary passes a "cut-and-choose test" depends on how large the test is and how many items are opened, but does otherwise not directly depend on computational hardness. The distinction between the two types of security parameter is important because the size of the cut-and-choose test has a dramatic effect on the efficiency of the protocol and thus the exact probability of cheating is important to analyze. For example, in our protocol, we prove that the error due to the statistical security parameter s

is approximately $2^{-0.311s}$. In general, we define the notion of indistinguishability with respect to n and s, so that the error due to s is allowed only to be $2^{-O(s)}$ (if this does not hold, then there is probably no reason to differentiate between n and s).

We consider ensembles that are indexed by integers (security parameters) n and s, and by arbitrary strings a. Security is required to hold for all a, and this value a represents the parties' inputs; thus, we obtain security for all inputs, and for large enough values of n and s. Formally, we have the following definition:

Definition 2.1. Let $X = \{X(a,n,s)\}_{n,s\in\mathbb{N};a\in\{0,1\}^*}$ and $Y = \{Y(a,n,s)\}_{n,s\in\mathbb{N};a\in\{0,1\}^*}$ be probability ensembles, so that for any $n,s \in \mathbb{N}$ the distribution $X(a,n,s)$ (resp., $Y(a,n,s)$) ranges over strings of length polynomial in $n+s$. We say that the ensembles are (n,s)-indistinguishable, denoted $X \overset{n,s}{\equiv} Y$, if there exists a constant $0 < c \leq 1$ such that for every non-uniform polynomial-time distinguisher D, every $a \in \{0,1\}^*$, every $s \in \mathbb{N}$, every polynomial $p(\cdot)$ and all large enough $n \in \mathbb{N}$:

$$\left|\Pr\left[D(X(a,n,s),a,n,s)=1\right] - \Pr\left[D(Y(a,n,s),a,n,s)=1\right]\right| < \frac{1}{p(n)} + \frac{1}{2^{c\cdot s}}.$$

Observe that the above is required to hold for *all* s and all large enough n. This reflects the fact that we will be *concrete* in s and asymptotic only in n.

Definitions of Security We refer the reader to [13, Chap. 7] for the definition of security for two-party computation in the presence of malicious adversaries. The bulk of this paper is in this model. The only difference is that we require (n,s)-indistinguishability between the ideal and real distributions, rather than just regular computational indistinguishability. We also consider the models of universal composability and covert adversaries, and refer the reader to [4] and [1], respectively, for appropriate definitions.

3. Cut-and-Choose Oblivious Transfer

3.1. *The Functionality and Construction Overview*

Our protocol for secure two-party computation uses a new primitive that we call *cut-and-choose oblivious transfer*. Loosely speaking, a cut-and-choose OT is a batch oblivious transfer protocol (meaning an oblivious transfer for multiple pairs of inputs) with the additional property that the receiver can choose a subset of the pairs (of a predetermined size) for which it learns *both* values. This is a very natural primitive which has clear applications for protocols that are based on cut-and-choose, as is our protocol here for general two-party computation.

The cut-and-choose OT functionality, denoted $\mathcal{F}_{\text{ccot}}$, with parameter s, is formally defined in Figure 3.1, together with a variant functionality that we will need, which considers the case that R is forced to use the *same* choice σ in every transfer. This variant is denoted $\mathcal{F}^S_{\text{ccot}}$.

In order to motivate the usefulness of this functionality, we describe its use in our protocol. Oblivious transfer is used in Yao's protocol so that the party computing the garbled circuit (call it P_2) can obtain the keys (garbled values) on the wires corresponding

FIGURE 3.1 The cut-and-choose OT functionalities.

The cut-and-choose OT functionality $\mathcal{F}_{\mathbf{ccot}}$:

- **Inputs:**
  - S inputs a vector of pairs $\overline{x} = \{(x_0^i, x_1^i)\}_{i=1}^s$
  - R inputs $\sigma_1, \ldots, \sigma_s \in \{0, 1\}$ and a set of indices $\mathcal{J} \subset [s]$ of size exactly $s/2$.
- **Output:** If $\mathcal{J}$ is not of size $s/2$ then S and R receive $\perp$ as output. Otherwise,
  - For every $j \in \mathcal{J}$ the receiver R obtains the pair (x_0^j, x_1^j).
  - For every $j \notin \mathcal{J}$ the receiver R obtains $x_{\sigma_j}^j$.

The single-choice cut-and-choose OT functionality $\mathcal{F}_{\mathbf{ccot}}^{\boldsymbol{S}}$:

- **Inputs:** The same as above, but with R having only a single input bit σ.
- **Output:** The same as above, but with R obtaining the value x_σ^j for every $j \notin \mathcal{J}$.

with its input while keeping its input secret; see Appendix A. When applying cut-and-choose, many circuits are constructed and then half of them are opened, where opening means that P_2 receives all of the input keys to the circuit. By using cut-and-choose OT, P_2 receives all of its keys in the circuits to be opened directly, in contrast to having P_1 send them separately after the indices of the circuits to be opened are sent from P_2 to P_1. The advantage of this approach is that P_1 cannot use *different* keys in the OT and when opening the circuit. See Sect. 4.1 for discussion on why this is important.

In cut-and-choose on Yao's protocol, one oblivious transfer is needed for every bit of P_2's input, and P_2 should receive the keys associated with the bit in all of the circuits. In order to ensure that P_2 uses the same input in all circuits, we use the *single-choice* variant. We present the basic variant since it is of independent interest and may be useful in other applications.

Constructing Cut-and-Choose OT The starting point for our construction of cut-and-choose OT is the universally composable protocol of Peikert et al. [37]; we refer only to the instantiation of their protocol based on the DDH assumption because this is the most efficient. However, our protocol can use any of their instantiations. The protocol of [37] is cast in the common reference string (CRS) model, where the CRS is a tuple (g_0, g_1, h_0, h_1) where g_0 is a generator of a group of order q (in which DDH is assumed to be hard), $g_1 = (g_0)^y$ for some random y, and it holds that $h_0 = (g_0)^a$ and $h_1 = (g_1)^b$ where $a \neq b$. We first observe that it is possible for the receiver to choose this tuple itself, as long as it proves that it indeed fulfills the property that $a \neq b$. Furthermore, this can be proven very efficiently by setting $b = a + 1$; in this case, the proof that $b = a + 1$ is equivalent to proving that $(g_0, g_1, h_0, \frac{h_1}{g_1})$ is a Diffie–Hellman tuple (note that the security of [37] is based only on $a \neq b$ and not on these values being independent of each other). We thus obtain a highly efficient version of the protocol of [37] in the stand-alone model.

Next, observe that the protocol of [37] has the property that if (g_0, g_1, h_0, h_1) *is* a Diffie–Hellman tuple (i.e., if $a = b$) then it is possible for the receiver to learn both values (of course, in a real execution this cannot happen because the receiver proves

that $a \neq b$). This property is utilized by [37] to prove universal composability; in their case the simulator can choose the CRS so that $a = b$ and then obtain both inputs of the sender, something that is needed for proving simulation-based security. However, in our case, we *want* the receiver to be able to sometimes learn both inputs of the sender. We can therefore utilize this exact property and have the receiver choose $s/2$ pairs (h_0, h_1) for which $a \neq b$ (ensuring that it learns only one input) and $s/2$ pairs (h_0, h_1) for which $a = b$ (enabling it to learn both inputs by actually running the simulator strategy of [37]). This therefore provides the exact cut-and-choose property in the OT that is needed. Of course, the receiver must also prove that it behaved in this way. Specifically, it proves in zero-knowledge that $s/2$ out of s pairs are such that $a \neq b$. We show that this too can be computed at low cost using the technique of Cramer et al. [7]; see Appendix B.2 for a full description and efficiency analysis of the zero-knowledge protocol.

3.2. *Background—The OT Protocol of Peikert et al. [37]*

Our cut-and-choose oblivious transfer protocol is based on the oblivious transfer of [37]. Their protocol is universally composable in the common reference string model. We present an efficient instantiation of the protocol in the *plain model*, where there is no common reference string. This protocol is secure against malicious parties, and forms the basis for our protocol. See Protocol 3.2 for a full description.

PROTOCOL 3.2 The Oblivious Transfer Protocol of [37] – Plain-Model Variant.

- **Inputs:** The sender's input is a pair (x_0, x_1) and the receiver's input is a bit σ
- **Auxiliary input:** Both parties hold a security parameter 1^n and $(\mathbb{G}, q, g_0)$, where $\mathbb{G}$ is an efficient representation of a group of order q with a generator g_0, and q is of length n.
- **The protocol:**
  1. The receiver R chooses random values $y, \alpha_0 \leftarrow \mathbb{Z}_q$ and sets $\alpha_1 = \alpha_0 + 1$. R then computes $g_1 = (g_0)^y$, $h_0 = (g_0)^{\alpha_0}$ and $h_1 = (g_1)^{\alpha_1}$ and sends (g_1, h_0, h_1) to the sender S.
  2. R proves, using a zero-knowledge proof of knowledge, that $(g_0, g_1, h_0, \frac{h_1}{g_1})$ is a DH tuple; see Protocol B.1. ((g_0, g_1, h_0, h_1) is used as the common reference string in the protocol of [37].)
  3. R chooses a random value r and computes $g = (g_\sigma)^r$ and $h = (h_\sigma)^r$, and sends (g, h) to S.
  4. The sender operates in the following way:
     - Define the function $RAND(w, x, y, z) = (u, v)$, where $u = (w)^s \cdot (y)^t$ and $v = (x)^s \cdot (z)^t$, and the values $s, t \leftarrow \mathbb{Z}_q$ are random.
     - S computes $(u_0, v_0) = RAND(g_0, g, h_0, h)$, and $(u_1, v_1) = RAND(g_1, g, h_1, h)$.
     - S sends the receiver the values (u_0, w_0) where $w_0 = v_0 \cdot x_0$, and (u_1, w_1) where $w_1 = v_1 \cdot x_1$.
  5. The receiver computes $x_\sigma = w_\sigma / (u_\sigma)^r$.

In order to see that the receiver obtains the correct values in the last step, observe that

$$\frac{w_\sigma}{(u_\sigma)^r} = \frac{v_\sigma \cdot x_\sigma}{(u_\sigma)^r} = \frac{g^s \cdot h^t \cdot x_\sigma}{((g_\sigma)^s \cdot (h_\sigma)^t)^r} = \frac{g^s \cdot h^t \cdot x_\sigma}{((g_\sigma)^r)^s \cdot ((h_\sigma)^r)^t} = \frac{g^s \cdot h^t \cdot x_\sigma}{g^s \cdot h^t} = x_\sigma .$$

Regarding security, if (g_0, g_1, h_0, h_1) is not a DH tuple, then the receiver can learn only one of the sender's inputs, since in that case one of the two pairs (u_0, w_0), (u_1, w_1) is uniformly distributed and therefore reveals no information about the corresponding input of the sender. This is due to the property of the *RAND* function used in Step 4: upon receiving a non-Diffie–Hellman tuple, the output of *RAND* is a pair of uniformly and *independently* distributed group elements. In contrast, if (g_0, g_1, h_0, h_1) *is* a DH tuple, and the receiver knows $y = \log_{g_0} g_1$, then the receiver can compute both inputs of the server. In order to see this, assume that $\sigma = 0$ and so $g = (g_0)^r$ and $h = (h_0)^r$. Then, it can compute $x_0 = w_0/(u_0)^r$ as in the protocol. In addition, it can compute $x_1 = w_1/(u_1)^{ry^{-1}}$. This works because

$$\begin{aligned}\frac{w_1}{(u_1)^{ry^{-1}}} &= \frac{g^s \cdot h^t \cdot x_1}{((g_1)^s \cdot (h_1)^t)^{ry^{-1}}} = \frac{g^s \cdot h^t \cdot x_1}{((g_1)^{y^{-1}})^s \cdot ((h_1)^{y^{-1}})^{t \cdot r}} \\ &= \frac{g^s \cdot h^t \cdot x_1}{((g_0)^s \cdot (h_0)^t)^r} = \frac{g^s \cdot h^t \cdot x_1}{g^s \cdot h^t} = x_1. \end{aligned} \tag{1}$$

Similarly, if $\sigma = 1$ then x_1 can be computed as in the protocol and x_0 can be computed as $w_0/(u_0)^{ry}$. In order to prevent a malicious receiver from doing this, the zero-knowledge proof of knowledge that $(g_0, g_1, h_0, \frac{h_1}{g_1})$ is a Diffie–Hellman tuple ensures the tuple (g_0, g_1, h_0, h_1) is *not* a DH tuple, and so the receiver can only learn a single value of the sender's input.

The proof of security takes advantage of the fact that a simulator can extract R's input-bit σ because it can extract the value α_0 from the zero-knowledge proof of knowledge proven by R. Given α_0, the simulator can compute $\alpha_1 = \alpha_0 + 1$ and then check if $h = g^{\alpha_0}$ (in which case $\sigma = 0$) or if $h = g^{\alpha_1}$ (in which case $\sigma = 1$). For simulation in the case that S is corrupted, the simulator sets $\alpha_0 = \alpha_1$ and cheats in the zero-knowledge proof, enabling it to extract both sender inputs. For the sake of completeness, we present a zero-knowledge proof of knowledge for DH tuples in Protocol B.1 in Appendix B.

Exact Efficiency In the OT without the zero-knowledge proof, the sender computes eight exponentiations and the receiver computes six. The zero-knowledge proof adds an additional five exponentiations for the prover (who is played by the receiver) and seven for the verifier (who is played by the sender). In addition, the parties exchange 17 group elements (including the zero-knowledge proof), and the protocol takes six rounds of communication (three messages are sent by each party). In summary, there are 26 *exponentiations*, 17 *group elements* sent and *six rounds of communication*.

3.3. *Constructing a Cut-and-Choose OT Protocol*

The idea behind the cut-and-choose OT protocol is essentially to run s copies of Protocol 3.2 in parallel, with the following important modification. Instead of requiring that (g_0, g_1, h_0, h_1) not be a DH tuple in any of the executions, we actually allow the receiver to choose $s/2$ of the executions in which it can set (g_0, g_1, h_0, h_1) to actually be a DH tuple. This means that in these executions, the receiver obtains both of the

sender's inputs. Of course, this must be done without the sender knowing for which of the executions the tuple is of the DH type and for which not. This is achieved by applying the methodology of Cramer et al. [7] for proving a compound statement to the basic DH zero-knowledge proof. The result is surprisingly efficient, and is described in Protocol B.2 in Appendix B. The construction of cut-and-choose OT is described in Protocol 3.3.

PROTOCOL 3.3 Cut-and-Choose Oblivious Transfer.

- **Inputs:** The sender's input is a vector of s pairs (x_0^j, x_1^j) and the receiver's input is comprised of s bits $\sigma_1, \ldots, \sigma_s$ and a set $\mathcal{J} \subset [s]$ of size exactly $s/2$.
- **Auxiliary input:** Both parties hold a security parameter 1^n and $(\mathbb{G}, q, g_0)$, where $\mathbb{G}$ is an efficient representation of a group of order q with a generator g_0, and q is of length n.
- **Setup phase:**
  1. R chooses a random $y \leftarrow \mathbb{Z}_q$ and sets $g_1 = (g_0)^y$.
  2. For every $j \in \mathcal{J}$, R chooses a random $\alpha_j \leftarrow \mathbb{Z}_q$ and computes $h_0^j = (g_0)^{\alpha_j}$ and $h_1^j = (g_1)^{\alpha_j}$.
  3. For every $j \notin \mathcal{J}$, R chooses random $\alpha_j \leftarrow \mathbb{Z}_q$ and computes $h_0^j = (g_0)^{\alpha_j}$ and $h_1^j = (g_1)^{\alpha_j+1}$.
  4. R sends $(g_1, h_0^1, h_1^1, \ldots, h_0^s, h_1^s)$ to S
  5. R proves using a zero-knowledge proof of knowledge to S that $s/2$ of the tuples $(g_0, g_1, h_0^j, \frac{h_1^j}{g_1})$ are DH tuples. (R must actually prove that $s/2$ of the tuples (g_0, g_1, h_0^j, h_1^j) are *not* DH tuples. In order to do this, it proves that the corresponding tuples $(g_0, g_1, h_0^j, h_1^j/g_1)$ are Diffie–Hellman tuples.) See Protocol B.2 in Appendix B. If S rejects the proof then it outputs $\perp$ and halts.
- **Transfer phase (repeated in parallel for every j):**
  1. The receiver chooses a random value $r_j \leftarrow \mathbb{Z}_q$ and computes $\tilde{g}_j = (g_{\sigma_j})^{r_j}, \tilde{h}_j = (h_{\sigma_j}^j)^{r_j}$. It sends $(\tilde{g}_j, \tilde{h}_j)$ to the sender.
  2. The sender operates in the following way:
     - Define the function $RAND(w, x, y, z) = (u, v)$, where $u = (w)^s \cdot (y)^t$ and $v = (x)^s \cdot (z)^t$, and the values $s, t \leftarrow \mathbb{Z}_q$ are random.
     - S sets $(u_0^j, v_0^j) = RAND(g_0, \tilde{g}_j, h_0^j, \tilde{h}_j)$, and $(u_1^j, v_1^j) = RAND(g_1, \tilde{g}_j, h_1^j, \tilde{h}_j)$.
     - S sends the receiver the values (u_0^j, w_0^j) where $w_0^j = v_0^j \cdot x_0^j$, and (u_1^j, w_1^j) where $w_1^j = v_1^j \cdot x_1^j$.
- **Output:**
  1. For every j (both $j \in \mathcal{J}$ and $j \notin \mathcal{J}$), the receiver computes $x_{\sigma_j}^j = \frac{w_{\sigma_j}^j}{(u_{\sigma_j}^j)^{r_j}}$.
  2. For every $j \in \mathcal{J}$, the receiver also computes $x_{1-\sigma_j}^j = \frac{w_{1-\sigma_j}^j}{(u_{1-\sigma_j}^j)^{r_j \cdot z}}$, where $z = y^{-1} \bmod q$ if $\sigma = 0$, and $z = y$ if $\sigma = 1$; see Eq. (1).

The security of the protocol is stated in the following proposition.

Proposition 3.4. *If the Decisional Diffie–Hellman assumption holds in the group* $\mathbb{G}$, *then Protocol* 3.3 *securely realizes the* $\mathcal{F}_{\text{ccot}}$ *functionality in the presence of malicious adversaries.*

Proof. Let $\mathcal{A}$ be an adversary that controls R. We construct a simulator $\mathcal{S}$ that invokes $\mathcal{A}$ on its input and works as follows:

1. $\mathcal{S}$ receives g_1 and $(h_0^1, h_1^1, \ldots, h_0^s, h_1^s)$ from $\mathcal{A}$ and verifies the zero-knowledge proof as the honest sender would.
   (a) If the verification fails, $\mathcal{S}$ sends $\perp$ to the trusted party computing $\mathcal{F}_{\text{ccot}}$ and halts.
   (b) Otherwise, $\mathcal{S}$ runs the extractor that is guaranteed to exist for the proof of knowledge, and extracts a witness set $\{(i_j, \alpha_{i_j})\}$ such that for every i_j it holds that $h_0^{i_j} = (g_0)^{\alpha_{i_j}}$ and $h_1^{i_j} = (g_1)^{\alpha_{i_j}+1}$. $\mathcal{S}$ defines the set $\mathcal{J}$ to be all of the indices *not* in the obtained witness set. (Note that when a pair $h_0^{i_j}, h_1^{i_j}$ is as above, then $\mathcal{A}$ can obtain only one of the strings. Thus, the set $\mathcal{J}$ of the indices where $\mathcal{A}$ receives both strings are those that are not included in this witness set.)

   (We remark that the above procedure does not guarantee that $\mathcal{S}$ runs in expected polynomial-time. Thus, formally $\mathcal{S}$ runs the witness-extended emulator of [26] that achieves the above effect.)
2. $\mathcal{S}$ receives $(\tilde{g}_1, \tilde{h}_1), \ldots, (\tilde{g}_s, \tilde{h}_s)$ from $\mathcal{A}$.
3. For every $j \notin \mathcal{J}$, simulator $\mathcal{S}$ has obtained α_j. $\mathcal{S}$ then sets $\sigma_j = 0$ if $\tilde{h}_j = (\tilde{g}_j)^{\alpha_j}$, and otherwise sets $\sigma_j = 1$.
4. For every $j \in \mathcal{J}$, $\mathcal{S}$ sets σ_j arbitrarily; say to equal 0.
5. $\mathcal{S}$ sends $\mathcal{J}$ and $\sigma_1, \ldots, \sigma_s$ to the trusted party. Then,
   (a) For every $j \in \mathcal{J}$, $\mathcal{S}$ receives back a pair (x_0^j, x_1^j).
   (b) For every $j \notin \mathcal{J}$, $\mathcal{S}$ receives back $x_{\sigma_j}^j$.
6. $\mathcal{S}$ concludes the execution by computing *RAND* as the honest sender would. Then,
   (a) For every $j \in \mathcal{J}$, $\mathcal{S}$ computes (u_0^j, w_0^j) and (u_1^j, w_1^j) exactly like the honest sender (it can do this because it knows both x_0^j and x_1^j).
   (b) For every $j \notin \mathcal{J}$, $\mathcal{S}$ computes $(u_{\sigma_j}^j, w_{\sigma_j}^j)$ like the honest sender using x_σ^j, and sets $(u_{1-\sigma_j}^j, w_{1-\sigma_j}^j)$ to be random elements of $\mathbb{G}$.
7. $\mathcal{S}$ sends all of these values to $\mathcal{A}$ and outputs whatever $\mathcal{A}$ outputs.

If the extraction of the witness set succeeds whenever $\mathcal{A}$ succeeds in proving the zero-knowledge proof, the output of the ideal execution with $\mathcal{S}$ is *identical* to the output of a real execution with $\mathcal{A}$ and an honest sender. This is due to the fact that the only difference is with respect to the way the $(u_{1-\sigma_j}^j, w_{1-\sigma_j}^j)$ are formed. However, if $(g_0, g_1, h_0^j, \frac{h_1^j}{g_1})$ is a Diffie–Hellman tuple, then (g_0, g_1, h_0^j, h_1^j) is *not* a Diffie–Hellman tuple. Now, if $\sigma_j = 0$ then $\tilde{h}_j = (\tilde{g}_j)^{\alpha_j}$ where $h_0^j = (g_0)^{\alpha_j}$. This therefore implies that

$(g_1, \tilde{g}_j, h_1^j, \tilde{h}_j)$ is also not a Diffie–Hellman tuple and so *RAND* applied to this tuple yields a uniformly distributed pair (u_1^j, w_1^j). (Likewise, if $\sigma = 1$ we find that *RAND* applied to (u_0^j, w_0^j) is uniformly distributed.) We conclude that the uniform choice of the pair $(u_{1-\sigma_j}^j, w_{1-\sigma_j}^j)$ by $\mathcal{S}$ yields exactly the same distribution as in a real execution. The proof of this corruption case is concluded by noting that the probability that $\mathcal{S}$ does not succeed in extracting a witness when $\mathcal{A}$ successfully proves is negligible.

We now proceed to the case that $\mathcal{A}$ controls the sender. We construct a simulator $\mathcal{S}$ as follows:

1. $\mathcal{S}$ computes $g_1 = (g_0)^y$ for a random $y \leftarrow \mathbb{Z}_q$, and chooses random values (h_0^j, h_1^j) so that (g_0, g_1, h_0^j, h_1^j) is a Diffie–Hellman tuple *for every* j, and sends the values to $\mathcal{A}$.
2. $\mathcal{S}$ runs the simulator for the zero-knowledge proof of knowledge with the residual $\mathcal{A}$ as the verifier.
3. For every j, $\mathcal{S}$ computes $\tilde{g}_j = (g_0)^{r_j}$ and $\tilde{h}_j = (h_0)^{r_j}$ and sends the pairs $(\tilde{g}_j, \tilde{h}_j)$ to $\mathcal{A}$.
4. Upon receiving back pairs (u_0^j, w_0^j) and (u_1^j, w_1^j), $\mathcal{S}$ computes $x_0^j = \frac{w_0^j}{(u_0^j)^{r_j}}$ and $x_1^j = \frac{w_1^j}{(u_1^j)^{r_j \cdot z}}$ where $z = y^{-1} \bmod q$; see (1).
5. $\mathcal{S}$ sends all pairs (x_0^j, x_1^j) to the trusted party, outputs whatever $\mathcal{A}$ outputs, and halts.

There are two main observations regarding the simulation. First, since all the $(g_0, \tilde{g}_j, h_0, \tilde{h}_j)$ and $(g_1, \tilde{g}_j, h_1, \tilde{h}_j)$ tuples are Diffie–Hellman tuples, $\mathcal{S}$ learns all of the correct (x_0^j, x_1^j) values that the honest receiver would receive in a real execution. Second, by the Decisional Diffie–Hellman assumption, the output of a simulated execution with $\mathcal{S}$ in the ideal model is indistinguishable from the output of a real execution between $\mathcal{A}$ and an honest receiver. Formally, we begin with a real execution between the receiver R and $\mathcal{A}$. Then, we modify R to be a simulator $\mathcal{S}_1$ that works exactly as R does except that instead of honestly proving the zero-knowledge proof, it runs the simulator instead. By the zero-knowledge property, the outcome of the two executions is indistinguishable. Next, we modify $\mathcal{S}_1$ to $\mathcal{S}_2$ by having $\mathcal{S}_2$ work in the same way except that it generates all of the h_0^j, h_1^j pairs so that $h_0^j = (g_0)^{\alpha_j}$ and $h_1^j = (g_1)^{\alpha_j}$ (for *all* j). The fact that these executions are indistinguishable is due to the DDH assumption. In particular, since the receiver only uses the knowledge of α_j to prove the zero-knowledge proof, both $\mathcal{S}_1$ and $\mathcal{S}_2$ can run their executions without knowing the α_j values at all, and even when they receive all of the h_0^j, h_1^j values as external input. A direct reduction to the DDH problem is then straightforward, and is thus omitted. Finally, we modify $\mathcal{S}_2$ to $\mathcal{S}_3$ who instead of outputting the values as the receiver would compute them, it extracts the pair (x_0^j, x_1^j) as the simulator $\mathcal{S}$ would and sets the receiver's output to be $x_{\sigma_j}^j$. Since $\mathcal{S}$ extracts the values in the same way as the honest receiver, this makes no difference to the output. Finally, we modify $\mathcal{S}_3$ to $\mathcal{S}_4$ by having it compute $\tilde{g}_j = (g_0)^{r_j}$ and $\tilde{h}_j = (h_0)^{r_j}$ irrespective of the real input σ_j. The output distribution generated by $\mathcal{S}_4$ is indistinguishable to that generated by $\mathcal{S}_3$ by another direct reduction to the DDH

FIGURE 3.5 Single-Choice Cut-and-Choose OT.

The inputs of the sender are depicted below. The values learned by the receiver are marked in bold. For every $j \in \mathcal{J}$ the receiver learns *both* values (x_0^j, x_1^j). In all pairs, the receiver learns one of the values, depending on σ. In the example here $\sigma = 0$ and so the zero-values are in bold in all pairs.

$$\Big((\boldsymbol{x_0^1}, x_1^1) \quad (\boldsymbol{x_0^2}, x_1^2) \quad \ldots \quad (\boldsymbol{x_0^j}, \boldsymbol{x_1^j}) \quad \ldots \quad (\boldsymbol{x_0^s}, x_1^s) \Big)$$

An example where $j \in \mathcal{J}$ and $\sigma = 0$.

problem. We conclude by noting that the distribution generated by $\mathcal{S}_4$ is identical to that generated by $\mathcal{S}$ in an ideal execution; specifically, it makes no difference if (x_0^j, x_1^j) are sent to the trusted party who then sends $x_{\sigma_j}^j$ to the receiver, or if $\mathcal{S}_4$ sets the receiver's output directly to $x_{\sigma_j}^j$. □

Exact Efficiency The setup phase requires $2s + 1$ exponentiations and the exchange of $2s + 1$ group elements, plus the zero-knowledge proof that adds an additional $7s + 4$ exponentiations and $3s + 4$ group elements sent; see Appendix B.2. Overall, the setup requires $9s + 5$ exponentiations and the exchange of $5s + 5$ group elements. The transfer phase requires $11.5s$ exponentiations and the exchange of $6s$ group elements. Finally, the number of rounds remains unchanged at six. In summary, there are $20.5s + 5$ *exponentiations*, $11s + 5$ *group elements* sent and *six rounds of communication*.

3.4. *Single-Choice Cut-and-Choose Oblivious Transfer*

The protocol for achieving the single-choice cut-and-choose OT functionality $\mathcal{F}_{\text{ccot}}^S$ implements the functionality that is defined formally in Figure 3.1; an intuitive description is provided in Figure 3.5.

The protocol $\mathcal{F}_{\text{ccot}}^S$ is achieved by modifying Protocol 3.3 above in Step 1 of the transfer phase so that the receiver R proves that it used the same σ in every tuple. In order to enable this proof to be carried out efficiently, we modify Step 1 of the transfer phase of Protocol 3.3 as follows:

> The receiver chooses a (single) random value $r \leftarrow \mathbb{Z}_q$ and computes $g' = (g_\sigma)^r$. Then, for every j, it computes $h_j = (h_\sigma^j)^r$. It sends $(g', h_1, \ldots, h_s)$ to the sender, and proves in zero-knowledge that it computed this correctly.

The required zero-knowledge proof is that there exists an $r \in \mathbb{Z}_q$ such that either $g' = (g_0)^r$ and $h_j = (h_0^j)^r$ for every $1 \le j \le s$, or $g' = (g_1)^r$ and $h_j = (h_1^j)^r$ for every $1 \le j \le s$. Equivalently, the required zero-knowledge proof is that either all of $\{(g_0, g', h_0^j, h_j)\}_{j=1}^s$ are Diffie–Hellman tuples, or all of $\{(g_1, g', h_1^j, h_j)\}_{j=1}^s$ are Diffie–Hellman tuples. Thus, the zero-knowledge proof of Protocol B.4 in Appendix B.3 can be used at the exact additional cost of $s + 18$ exponentiations and the exchange of 10 group elements (no additional rounds are needed because this proof can be carried out in parallel to the proof in the setup phase). By the soundness of the zero-knowledge proof, R must use the same σ in every transfer. The other difference in the

protocol is that instead of sending a different (g_j, h_j) pair for every j, the receiver sends a single value g'. In the proof below, we show that this does not leak any information to the sender.

Proposition 3.6. *If the Decisional Diffie–Hellman assumption holds in the group* $\mathbb{G}$, *then the modified protocol for single-choice cut-and-choose oblivious transfer, securely realizes the* $\mathcal{F}^S_{\text{ccot}}$ *functionality in the presence of malicious adversaries.*

Proof. The proof is identical to the proof of Proposition 3.4, except for the following modification. When analyzing the case that the adversary controls the sender, the proof of Proposition 3.4 describes a sequence of simulators $\mathcal{S}_1, \ldots, \mathcal{S}_4$, which replace the operation of the receiver. We use the same simulators except for simulator $\mathcal{S}_4$. This simulator now replaces the message $(g', h_1, \ldots, h_s) = ((g_\sigma)^r, (h^1_\sigma)^r, \ldots, (h^s_\sigma)^r))$, with the message $((g_0)^r, (h^1_0)^r, \ldots, (h^s_0)^r))$, regardless of the value of σ. It must be shown that the distributions of these two messages are indistinguishable by the sender. Note that the sender also receives the values of the set $S = (g_0, g_1, h^1_0, h^1_1, \ldots, h^s_0, h^s_1)$, where $\forall j\ h^j_0 = (g_0)^{\alpha_j}, h^j_1 = (g_1)^{\alpha_j}$. (In the original protocol, h^j_1 can also be equal to $(g_1)^{\alpha_j+1}$, but in simulators $\mathcal{S}_2$ and $\mathcal{S}_3$ the input distribution is as we describe here.)

Reordering the items in S, we can define it as $S = S_0 \cup S_1$ where $S_b = (g_b, (g_b)^{\alpha_1}, \ldots, (g_b)^{\alpha_s})$, for $b = 0, 1$. Let us denote by S^r_b the set $S^r_b = ((g_b)^r, (g_b)^{\alpha_1 r}, \ldots, (g_b)^{\alpha_s r})$. Note that S^r_0 and S^r_1 have exactly the same distribution when r is chosen at random. The sender receives either $\langle S, S^r_0\rangle$ or $\langle S, S^r_1\rangle$, and therefore cannot distinguish between these two options. □

Exact Efficiency The efficiency of this protocol is the same as above except for the following two changes: (1) There is no need to compute s different g_j values (a single g' is computed), and therefore $s - 1$ exponentiations are eliminated. (2) An additional $s + 18$ exponentiations and the exchange of 10 additional group elements are needed for the zero-knowledge protocol. We therefore have $20.5s + 24$ *exponentiations*, $11s + 15$ *group elements* sent and *six rounds of communication*.

3.5. *Batch Single-Choice Cut-and-Choose OT*

In our protocol we need to carry out cut-and-choose oblivious transfers for all wires in the circuit. Furthermore, it is crucial that the subset of indices for which the receiver obtains both pairs is the same in all transfers. We call a functionality that achieves this "batch single-choice cut-and-choose OT" and denote it $\mathcal{F}^{S,B}_{\text{ccot}}$. The functionality is formally defined in Figure 3.7, and an example is given in Figure 3.8.

In order to realize this functionality it suffices to run the setup phase of Protocol 3.3 once, and then the transfer phase of the protocol with single choice ℓ times in parallel (with receiver inputs $\sigma_1, \ldots, \sigma_\ell$ where σ_i is the receiver's choice in execution i). This ensures that the same set $\mathcal{J}$ is used in all transfers, since $\mathcal{J}$ depends only on the values sent in the setup phase. We remark that parallel composition holds here because the simulation only rewinds in the transfer phase for the zero-knowledge protocol, and Protocol B.2 that forms the basis of the zero-knowledge proof is zero-knowledge under parallel composition (as stated in Proposition B.3).

FIGURE 3.7 The Batch Single-Choice Cut-and-Choose OT Functionality $\mathcal{F}_{\text{ccot}}^{S,B}$.

- **Inputs:**
  - S inputs ℓ *vectors* of pairs $\vec{x}_i$ of length s, for $i = 1, \ldots, \ell$. (Every vector is a row of s pairs. There are ℓ such rows. This can be viewed as an $\ell \times s$ matrix of pairs; see Figure 3.8.)
  - R inputs $\sigma_1, \ldots, \sigma_\ell \in \{0, 1\}$ and a set of indices $\mathcal{J} \subset [s]$ of size exactly $s/2$. (For every row the receiver chooses a bit σ_i. It also chooses $s/2$ of the s "columns".)
- **Output:** If $\mathcal{J}$ is not of size $s/2$ then S and R receive for output $\perp$. Otherwise,
  - For every $i = 1, \ldots, \ell$ and for every $j \in \mathcal{J}$, the receiver R obtains the jth pair in vector $\vec{x}_i$. (For every column in $\mathcal{J}$, the receiver obtains the two items of every pair, in all rows.)
  - For every $i = 1, \ldots, \ell$, the receiver R obtains the σ_i value in every pair of the vector $\vec{x}_i$. (For every column not in $\mathcal{J}$, the receiver obtains its choice σ_i of the two items in the pair, where σ_i is the same for all entries in a row.)

FIGURE 3.8 Batch Single-Choice Cut-and-Choose OT.

Let the matrix below denote the inputs of the sender. Then, for $j \in \mathcal{J}$ the receiver learns *both* values in the jth column (values in bold). Furthermore, in each row, the receiver learns one of the values, depending on the receiver input associated with that row. For example, in the ith row in the example here $\sigma_i = 0$ and so the zero-values are in bold in the ith row.

$$\begin{pmatrix} (\boldsymbol{x_0^{1,1}}, x_1^{1,1}) & (\boldsymbol{x_0^{1,2}}, x_1^{1,2}) & \ldots & (\boldsymbol{x_0^{1,j}}, \boldsymbol{x_1^{1,j}}) & \ldots & (\boldsymbol{x_0^{1,s}}, x_1^{1,s}) \\ (x_0^{2,1}, \boldsymbol{x_1^{2,1}}) & (x_0^{2,2}, \boldsymbol{x_1^{2,2}}) & \ldots & (\boldsymbol{x_0^{2,j}}, \boldsymbol{x_1^{2,j}}) & \ldots & (x_0^{2,s}, \boldsymbol{x_1^{2,s}}) \\ \vdots & \vdots & & \vdots & & \vdots \\ (\boldsymbol{x_0^{i,1}}, x_1^{i,1}) & (\boldsymbol{x_0^{i,2}}, x_1^{i,2}) & \ldots & (\boldsymbol{x_0^{i,j}}, \boldsymbol{x_1^{i,j}}) & \ldots & (\boldsymbol{x_0^{i,s}}, x_1^{i,s}) \\ \vdots & \vdots & & \vdots & & \vdots \\ (\boldsymbol{x_0^{\ell,1}}, x_1^{\ell,1}) & (\boldsymbol{x_0^{\ell,2}}, x_1^{\ell,2}) & \ldots & (\boldsymbol{x_0^{\ell,j}}, \boldsymbol{x_1^{\ell,j}}) & \ldots & (\boldsymbol{x_0^{\ell,s}}, x_1^{\ell,s}) \end{pmatrix}$$

An example where $j \in \mathcal{J}$ and $\sigma_i = 0$.

Proposition 3.9. *Assuming that the Decisional Diffie–Hellman assumption holds in* $\mathbb{G}$, *the above-described protocol securely realizes* $\mathcal{F}_{\text{ccot}}^{S,B}$ *in the presence of malicious adversaries.*

Exact Efficiency The setup phase here remains the same, and including the zero-knowledge costs $9s + 5$ exponentiations and the exchange of $5s + 5$ group elements. The transfer phase is repeated ℓ times, where each transfer incurs a cost of $11.5s + 19$ exponentiations and the exchange of $5s + 11$ group elements. We conclude that there are $11.5s\ell + 19\ell + 9s + 5$ *exponentiations*, $5s\ell + 11\ell + 5s + 5$ *group elements* sent and *six rounds of communication*. In Sect. 4.4 we observe that $9.5s\ell$ of the exponentiations are "fixed-base" and thus this can actually be reduced to the equivalent of $5.166s\ell$ *exponentiations*.

4. The Protocol for Secure Two-Party Computation

4.1. *Protocol Description*

Before describing the protocol in detail, we first present an intuitive explanation of the different steps, and their purpose.

Step 1: P_1 constructs s copies of a Yao garbled circuit for computing the function. The keys (garbled values) on the wires of the s copies of the circuit are all random, except for the keys corresponding to P_1's input wires, which are chosen in a special way. Namely, P_1 chooses random values $a_1^0, a_1^1, \ldots, a_\ell^0, a_\ell^1$ (where the length of P_1's input is ℓ) and $r_1, \ldots, r_s$, and sets the keys on the wire associated with its ith input in the jth circuit to be $g^{a_i^0 \cdot r_j}$ and $g^{a_i^1 \cdot r_j}$. Note that the $2\ell + s$ values $g^{a_1^0}, g^{a_1^1}, \ldots, g^{a_\ell^0}, g^{a_\ell^1}, g^{r_1}, \ldots, g^{r_s}$ constitute commitments to all $2\ell s$ keys.[2] (The keys are actually a pseudorandom synthesizer [34], and therefore if some of the keys are revealed, the remaining keys remain pseudorandom.)

Step 2: The parties execute batch single-choice cut-and-choose OT. P_1 inputs the key-pairs for all wires associated with P_2's input, and P_2 inputs its input and a random set $\mathcal{J} \subset [s]$ of size $s/2$. The result is that P_2 learns all the keys on the wires associated with its own input for $s/2$ of the circuits as indexed by $\mathcal{J}$ (called check circuits), and in addition learns the keys corresponding to its actual input in these wires in the remaining circuits (called evaluation circuits).

Step 3: P_1 sends P_2 the garbled circuits, and the values $g^{a_1^0}, g^{a_1^1}, \ldots, g^{a_\ell^0}, g^{a_\ell^1}, g^{r_1}, \ldots, g^{r_s}$ which are commitments to all the keys on the wires associated with P_1's input. At this stage P_1 is fully committed to all s circuits, but does not yet know which circuits are to be opened.

Step 4: P_2 reveals to P_1 its choice of check circuits and proves that this was indeed its choice by sending *both* values on the wire associated with P_2's first input bit in each check circuit. Note that P_2 can know both these values only for circuits that are check circuits.

Step 5: To completely decrypt the check circuits in order to check that they were correctly constructed, P_2 also needs to obtain all the keys on the wires associated with P_1's input. Therefore, if the jth circuit is a check circuit, P_1 sends r_j to P_2. Given all of the $g^{a_i^0}, g^{a_i^1}$ values and r_j, P_2 can compute all of the keys $g^{a_i^0 \cdot r_j}, g^{a_i^1 \cdot r_j}$ in the jth circuit by itself (and P_1 cannot change the values). Furthermore, this reveals nothing about the keys in the evaluation circuits.

Step 6: Given all of the keys on all of the input wires, P_2 checks the validity of the $s/2$ check circuits. This ensures that P_2 will catch P_1 with high probability if many of the garbled circuits generated by P_1 do not compute the correct function. Thus, unless P_2 detects cheating, it is ensured that a majority of the evaluation circuits are correct.

---

[2] The actual symmetric keys used are derived from the $g^{a_i^0 \cdot r_j}, g^{a_i^1 \cdot r_j}$ values using a randomness extractor; a universal hash function suffices for this [6,18]. The only subtlety is that P_1 must be fully committed to the garbled circuits, including these symmetric keys, before it knows which circuits are to be checked. However, randomness extractors are not 1-1 functions. This is solved by having P_1 send the seed for the extractor before Step 4 below. Observe that the $\{g^{a_i^0}, g^{a_i^1}, g^{r_j}\}$ values and the seed for the extractor fully determine the symmetric keys, as required.

Step 7: All that remains is for P_1 to send P_2 the keys associated with its actual input, and then P_2 will be able to compute the evaluation circuits. This raises a problem as to how P_2 can be sure that P_1 sends keys that correspond to the same input in all circuits. This brings us to the way that P_1 chose these keys (via the Diffie–Hellman pseudorandom synthesizer). Specifically, for every wire i and evaluation-circuit j, party P_1 sends P_2 the value $g^{a_i^{x_i} \cdot r_j}$ where x_i is the ith bit of P_1's input. P_1 then proves in zero-knowledge that the same $a_i^{x_i}$ exponent appears in all of the values sent. Essentially, this is a proof that the values constitute an "extended" Diffie–Hellman tuple and thus this statement can be proven very efficiently.

Step 8: Finally, given the keys associated with P_1's inputs and its own inputs, P_2 evaluates the evaluation circuits and obtains their output values. Recall, however, that the checks above only guarantee that a majority of the circuits are correct, and not that all of them are. Therefore, P_2 outputs the value that is output from the majority of the evaluation circuits. We stress that if P_2 sees different outputs in different circuits, and thus knows for certain that P_1 has tried to cheat, it must ignore this observation and output the majority value (or otherwise it might leak information to P_1, as in the example described in Sect. 1.2).

4.2. *Proof of Security*

Before providing the full proof of security, we give some intuition regarding the security of the protocol. We first show that the selective failure attack discussed in [24,28] cannot be carried out here. The concern there was that P_1 would use correct keys for all of P_2's input bits when opening the check circuit, but would use incorrect keys in some of the oblivious transfers. This is problematic because if P_1 input incorrect keys for the zero value of P_2's first input bit, and correct keys for all other values, then P_2 would not detect any misbehavior if its first input bit equals 1. However, if its first input bit equals 0 then it would have to abort (because it would not be able to decrypt any of the evaluation circuits). This results in P_1 learning P_2's first input bit with probability 1. In order to solve this problem in [28] it was necessary to split P_2's input bits into random shares, thereby increasing the size of the input to the circuit and the size of the circuit itself. In contrast, this attack does not arise here at all because P_2 obtains all of the keys associated with its input bits in the cut-and-choose oblivious transfer, and the values are not sent separately for check and evaluation circuits. Thus, if P_1 attempts a similar attack here for a small number of circuits then it will not be the majority and so does not matter, and if it does so for a large number of circuits then it will be caught with overwhelming probability.

Observe also that in Steps 3–5 P_2 checks that half of the circuits, the check circuits, and their corresponding input garbled values, were correctly constructed. This is done by first having P_1 commit to all circuits and then having P_2 choosing half of them. P_2 is therefore ensured that with high probability the majority of the remaining circuits, and their input garbled values, are also correct. Consequently, the result output by the majority of the remaining circuits must be correct.

The security of the protocol is expressed in the following theorem:

PROTOCOL 4.1 Computing $f(x, y)$.

Inputs: P_1 has input $x \in \{0,1\}^\ell$ and P_2 has input $y \in \{0,1\}^\ell$.

Auxiliary input: a statistical security parameter s, the description of a circuit C such that $C(x, y) = f(x, y)$, and $(\mathbb{G}, q, g)$ where $\mathbb{G}$ is a cyclic group with generator g and *prime* order q, and q is of length n.

The protocol:

1. INPUT KEY CHOICE AND CIRCUIT PREPARATION:
   (a) P_1 chooses random values $a_1^0, a_1^1, \ldots, a_\ell^0, a_\ell^1 \in_R \mathbb{Z}_q$ and $r_1, \ldots, r_s \in_R \mathbb{Z}_q$.
   (b) Let $w_1, \ldots, w_\ell$ be the input wires corresponding to P_1's input in C, and denote by $w_{i,j}$ the instance of wire w_i in the jth garbled circuit, and by $k_{i,j}^b$ the key associated with bit b on wire $w_{i,j}$. Then, P_1 sets the keys for its input wires to
   $$k_{i,j}^0 = H(g^{a_i^0 \cdot r_j}) \quad \text{and} \quad k_{i,j}^1 = H(g^{a_i^1 \cdot r_j})$$
   where H is a suitable randomness extractor [6,18]; see also [9].
   (c) P_1 constructs s independent copies of a garbled circuit of C, denoted $GC_1, \ldots, GC_s$, using random keys except for wires $w_1, \ldots, w_\ell$ for which the keys are as above; see Appendix A.
2. OBLIVIOUS TRANSFERS: P_1 and P_2 run batch single-choice cut-and-choose oblivious transfer (Protocol 3.7), with parameters ℓ (the number of parallel executions) and s (the number of pairs in each execution):
   (a) P_1 defines vectors $\vec{z}_1, \ldots, \vec{z}_\ell$ so that $\vec{z}_i$ contains the s pairs of random symmetric keys associated with P_2's ith input bit y_i in all garbled circuits $GC_1, \ldots, GC_s$.
   (b) P_2 inputs a random subset $\mathcal{J} \subset [s]$ of size exactly $s/2$ and bits $\sigma_1, \ldots, \sigma_\ell \in \{0,1\}$, where $\sigma_i = y_i$ for every i.
   (c) P_2 receives all the keys associated with its input wires in all circuits GC_j for $j \in \mathcal{J}$, and receives the keys associated with its input y on its input wires in all other circuits.
3. SEND CIRCUITS AND COMMITMENTS: P_1 sends P_2 the garbled circuits (i.e., the gate and output tables), the "seed" for the randomness extractor H, and the following "commitment" to the garbled values associated with P_1's input wires:
   $$\left\{(i, 0, g^{a_i^0}), (i, 1, g^{a_i^1})\right\}_{i=1}^{\ell} \quad \text{and} \quad \left\{(j, g^{r_j})\right\}_{j=1}^{s}.$$
4. SEND CUT-AND-CHOOSE CHALLENGE: P_2 sends P_1 the set $\mathcal{J}$ along with the *pair* of keys associated with its first input bit y_1 in every circuit GC_j for $j \in \mathcal{J}$. If the values received by P_1 are incorrect, it outputs $\perp$ and aborts. Circuits GC_j for $j \in \mathcal{J}$ are called check circuits, and for $j \notin \mathcal{J}$ are called evaluation-circuits.
5. SEND ALL INPUT GARBLED VALUES IN CHECK CIRCUITS: For every *check circuit* GC_j, party P_1 sends the value r_j to P_2, and P_2 checks that these are consistent with the pairs $\{(j, g^{r_j})\}_{j \in \mathcal{J}}$ received in Step 3. If not, P_2 aborts outputting $\perp$.
6. CORRECTNESS OF CHECK CIRCUITS: For every $j \in \mathcal{J}$, P_2 uses the $g^{a_i^0}, g^{a_i^1}$ values it received in Step 3, and the r_j values it received in Step 5, to compute the values $k_{i,j}^0 = H(g^{a_i^0 \cdot r_j}), k_{i,j}^1 = H(g^{a_i^1 \cdot r_j})$ associated with P_1's input in GC_j. In addition it sets the garbled values associated with its own input in GC_j to be as obtained in the cut-and-choose OT. Given all the garbled values for all input wires in GC_j, party P_2 decrypts the circuit and verifies that it is a garbled version of C. If there exists a circuit for which this does not hold, then P_2 aborts and outputs $\perp$.
7. P_1 SENDS ITS GARBLED INPUT VALUES IN THE EVALUATION-CIRCUITS:
   (a) P_1 sends the keys associated with its inputs in the evaluation circuits: For every $j \notin \mathcal{J}$ and every wire $i = 1, \ldots, \ell$, party P_1 sends the value $k'_{i,j} = g^{a_i^{x_i} \cdot r_j}$; P_2 sets $k_{i,j} = H(k'_{i,j})$.

(b) P_1 proves that all input values are consistent: For every input wire $i = 1, \ldots, \ell$, party P_1 uses Protocol B.4 to prove *in parallel* that there exists a value $\sigma_i \in \{0, 1\}$ such that for every $j \notin \mathcal{J}$, $k'_{i,j} = g^{a_i^{\sigma_i} \cdot r_j}$. (Namely, it proves that all garbled values of a wire are of the same bit.) If any of the proofs fail, then P_2 aborts and outputs $\perp$.

8. CIRCUIT EVALUATION: P_2 uses the keys associated with P_1's input obtained in Step 77 and the keys associated with its own input obtained in Step 22 to evaluate the evaluation circuits GC_j for every $j \notin \mathcal{J}$. If a circuit decryption fails, then P_2 sets the output of that circuit to be $\perp$. Party P_2 takes the output that appears in most circuits, and outputs it.

Theorem 4.2. *Assume that the decisional Diffie–Hellman assumption is hard in* $\mathbb{G}$, *that the protocol used in Step* 2 *securely computes the batch single-choice cut-and-choose oblivious transfer functionality*, *that the protocol used in Step* 7b *is a zero-knowledge proof of knowledge*, *and that the symmetric encryption scheme used to generate the garbled circuits is secure*. *Then*, *Protocol* 4.1 *securely computes the function* f *in the presence of malicious adversaries*.

Proof. We prove Theorem 4.2 in a hybrid model where a trusted party is used to compute the batch single-choice cut-and-choose oblivious transfer functionality and the zero-knowledge proof of knowledge of Step 7b. We separately prove the case that P_1 is corrupted and the case that P_2 is corrupted.

P_1 Is Corrupted Intuitively, P_1 can only cheat by constructing some of the circuits in an incorrect way. However, in order for this to influence the outcome of the computation, it has to be that a *majority* of the evaluation circuits, or equivalently over one quarter of them, are incorrect. Furthermore, it must hold that none of these incorrect circuits are check circuits. The reason why this bad event occurs with such small probability is that P_1 is *committed* to the circuits before it learns which circuits are check circuits and which are evaluation circuits. In order to see this, observe that in the cut-and-choose oblivious transfer, P_2 receives all of the keys associated with its own input wires for the check circuits in $\mathcal{J}$ (while P_1 knows nothing about $\mathcal{J}$). Furthermore, P_1 sends all of the $\{(i, 0, g^{a_i^0}), (i, 1, g^{a_i^1})\}$ and $\{(j, g^{r_j}\}$ values, and the garbled circuit tables, *before* learning $\mathcal{J}$. Thus, it can only succeed in cheating if it successfully guesses over $s/4$ circuits which all happen to not be in $\mathcal{J}$. As we will show, this occurs with probability of approximately $2^{-s/4}$. (Recall also that P_1 is required to prove that all its inputs to the evaluation circuits are consistent, and therefore changing the circuits is its only option to changing the computed functionality.) We remark that it is also crucial that if P_2 aborts by detecting cheating by P_1, then this occurs *independently* of P_2's input. However, this follows immediately from the protocol description. We now proceed to the formal proof.

Let $\mathcal{A}$ be an adversary controlling P_1 in an execution of Protocol 4.1 where a trusted party is used to compute the cut-and-choose OT functionality $\mathcal{F}_{\text{ccot}}^{S,B}$ and the zero-knowledge proof of knowledge of Step 7b. We construct a simulator $\mathcal{S}$ who runs in the ideal model with a trusted party computing f. $\mathcal{S}$ runs $\mathcal{A}$ internally and simulates the honest P_2 for $\mathcal{A}$ as well as the trusted party computing the oblivious transfer and zero-knowledge functionalities. In addition, $\mathcal{S}$ interacts *externally* with the trusted party computing f. $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ invokes $\mathcal{A}$ upon its input and auxiliary input and receives the inputs that $\mathcal{A}$ sends to the trusted party computing the cut-and-choose OT functionality. These inputs constitute an $n \times s$ matrix of pairs $\{(z_0^{i,j}, z_1^{i,j})\}$ for $i = 1, \ldots, n$ and $j = 1, \ldots, s$.
2. $\mathcal{S}$ receives from $\mathcal{A}$ the s garbled circuits $GC_1, \ldots, GC_s$ and values $\{(i, 0, u_i^0)\}$, $(i, 1, u_1^i)\}$ and $\{(j, h_j)\}$ (consistent with Step 3 of the protocol).
3. $\mathcal{S}$ chooses a subset $\mathcal{J} \subset [s]$ of size $s/2$ uniformly at random amongst all such subsets. For every $j \in \mathcal{J}$, $\mathcal{S}$ hands $\mathcal{A}$ the values $\{(z_0^{1,j}, z_1^{1,j})\}$, as it expects to receive from the honest P_2 in Step 4 of the protocol.
4. $\mathcal{S}$ receives the set $\{r_j\}_{j \in \mathcal{J}}$ from $\mathcal{A}$, and checks that for every $j \in \mathcal{J}$ it holds that $h_j = g^{r_j}$. If not, it sends $\perp$ to the trusted party, simulates P_2 aborting, and outputs whatever $\mathcal{A}$ outputs.
5. $\mathcal{S}$ verifies that all garbled circuits GC_j for $j \in \mathcal{J}$ are correctly constructed (it does this in the same way that an honest P_2 would). If not, it sends $\perp$ to the trusted party, simulates P_2 aborting, and outputs whatever $\mathcal{A}$ outputs.
6. $\mathcal{S}$ receives keys $k'_{i,j}$ from $\mathcal{A}$, for every $j \notin \mathcal{J}$ and $i = 1, \ldots, \ell$.
7. $\mathcal{S}$ receives the witnesses that $\mathcal{S}$ sends to the trusted party computing the zero-knowledge proof of knowledge functionality of Step 7b. Thus, for every $i = 1, \ldots, \ell$, $\mathcal{S}$ receives a value a_i such that $k'_{i,j} = (h_j)^{a_i}$ for every $j \notin \mathcal{J}$, and either $u_i^0 = g^{a_i}$ or $u_i^1 = g^{a_i}$ (this is the witness).
   (a) If for some i, $\mathcal{S}$ does not receive a valid witness, then it sends $\perp$ to the trusted party, simulates P_2 aborting, and outputs whatever $\mathcal{A}$ outputs.
   (b) Otherwise, for every $i = 1, \ldots, \ell$, if $u_i^0 = g^{a_i}$ then $\mathcal{S}$ sets $x_i = 0$, and if $u_i^1 = g^{a_i}$ then $\mathcal{S}$ sets $x_i = 1$.
8. $\mathcal{S}$ sends $x = x_1 \cdots x_\ell$ to the trusted party computing f, outputs whatever $\mathcal{A}$ outputs and halts.

Denoting Protocol 4.1 by π, we now show that for every $\mathcal{A}$ corrupting P_1 and every s it holds that

$$\left\{\text{IDEAL}_{f,\mathcal{S}}(x, y, z, n, s)\right\}_{x,y,z \in \{0,1\}^*; n,s \in \mathbb{N}} \stackrel{n,s}{\equiv} \left\{\text{REAL}_{\pi,\mathcal{A}}(x, y, z, n, s)\right\}_{x,y,z \in \{0,1\}^*; n,s \in \mathbb{N}}$$

where $|x| = |y|$. (Note that here we prove (n, s)-indistinguishability, and so the probability of distinguishing must be at most $\mu(n) + 2^{-O(s)}$ for some negligible function μ.)

We begin by defining the notion of a bad circuit. For a garbled circuit GC_j we define the circuit input keys as follows:

1. *Circuit input keys associated with P_1's input:* Let $(i, 0, g^{a_i^0})$, $(i, 1, g^{a_i^1})$, (j, g^{r_j}) be the values sent by P_1 to P_2 in Step 3 of the protocol. Then, the circuit input keys associated with P_1's input in GC_j are the keys $(g^{a_1^0 \cdot r_j}, g^{a_1^1 \cdot r_j}), \ldots, (g^{a_\ell^0 \cdot r_j}, g^{a_\ell^1 \cdot r_j})$.
2. *Circuit input keys associated with P_2's input:* Let $(z_0^{1,j}, z_1^{1,j}), \ldots, (z_0^{\ell,j}, z_1^{\ell,j})$ be the set of symmetric keys input by P_1 to the cut-and-choose oblivious transfer of Step 2. (These keys are the jth pair in each vector $\vec{z}_1, \ldots, \vec{z}_\ell$.) These values are called the circuit input keys associated with P_2's input in GC_j.

Before proceeding, we stress that all of the above circuit keys are *fully determined* after Step 3 of the protocol, as are the garbled circuits GC_j. This is because P_1 sends the $\{g^{a_i^0}, g^{a_i^1}, g^{r_j}\}$ values, the garbled circuits and the seed to the randomness extractor in this step (note that once the seed to the randomness extractor is fixed, the symmetric keys derived from the Diffie–Hellman values are fully determined). Now, simply stated, a garbled circuit GC_j is bad if the circuit keys associated with both P_1's and P_2's input do *not* open it to the correct circuit C. We stress again that after Step 3 of the protocol, each circuit is either "bad" or "not bad", and this is fully determined.

Our aim is now to bound the probability that P_2 does not abort and yet the majority of the *evaluation* circuits are bad. In order to do this, note first that the set $\mathcal{J}$ is *completely hidden* in an information-theoretic sense from P_1 until Step 4 of the protocol (this holds in an information-theoretic sense in the hybrid model where a trusted party computes the cut-and-choose oblivious transfer, which is the model in which we carry out our analysis). Thus, for the sake of computing the probabilities we can consider the case that $\mathcal{J}$ is chosen randomly after Step 3. Now, let badMaj denote the event that at least $s/4$ of the garbled circuits are bad, and let noAbort denote the event that P_2 does not abort in Step 6 of the protocol. We now bound the probability of the event that both badMaj and noAbort occur.

Claim 4.3. *For every $s \in \mathbb{N}$ it holds that*

$$\Pr[\mathsf{noAbort} \wedge \mathsf{badMaj}] = \frac{\binom{\frac{3s}{4}+1}{\frac{s}{2}+1}}{\binom{s}{s/2}} < \frac{1}{2^{s/4-1}},$$

and for large enough s (depending on Stirling's approximation), it holds that

$$\Pr[\mathsf{noAbort} \wedge \mathsf{badMaj}] \approx \frac{1}{2^{0.311s}}.$$

Proof. Let badTotal be the number of bad circuits. First observe that

$$\Pr[\mathsf{noAbort} \wedge \mathsf{badMaj}] = \sum_{i=s/4}^{s/2} \Pr[\mathsf{noAbort} \wedge \mathsf{badTotal} = i]$$

because if badTotal $> s/2$ then P_2 always aborts, and if badTotal $< \frac{s}{4}$ then badMaj is false. Recall that $|\mathcal{J}| = s/2$ and so if i circuits are bad and no abort takes place, then it

must be that $s/2$ of the $s-i$ not-bad circuits were chosen to be checked. Thus,

$$\begin{aligned}\sum_{i=s/4}^{s/2} \Pr[\mathsf{noAbort} \wedge \mathsf{badTotal} = i] &= \sum_{i=s/4}^{s/2} \frac{\binom{s-i}{s/2}}{\binom{s}{s/2}} \\ &= \frac{1}{\binom{s}{s/2}} \sum_{i=s/4}^{s/2} \binom{s-i}{s/2} = \frac{1}{\binom{s}{s/2}} \sum_{i=0}^{s/4} \binom{s/2+i}{s/2} \\ &= \frac{1}{\binom{s}{s/2}} \sum_{i=0}^{3s/4} \binom{i}{s/2} = \frac{1}{\binom{s}{s/2}} \cdot \binom{3s/4+1}{s/2+1},\end{aligned}$$

where the second last equality is due to the fact that for $i < s/2$ it holds that $\binom{i}{s/2} = 0$, and the last equality can be found in [17, Page 174]. We now bound this last value:

$$\begin{aligned}\frac{\binom{\frac{3s}{4}+1}{\frac{s}{2}+1}}{\binom{s}{\frac{s}{2}}} &= \frac{(\frac{3s}{4}+1)!}{(\frac{s}{2}+1)!(\frac{s}{4})!} \cdot \frac{(\frac{s}{2})!(\frac{s}{2})!}{s!} = \frac{(\frac{3s}{4}+1)!}{s!} \cdot \frac{(\frac{s}{2})!}{(\frac{s}{4})!} \cdot \frac{(\frac{s}{2})!}{(\frac{s}{2}+1)!} \\ &= \frac{(\frac{s}{2})(\frac{s}{2}-1)\cdots(\frac{s}{4}+1)}{s(s-1)\cdots(\frac{3s}{4}+2)} \cdot \frac{1}{\frac{s}{2}+1} \\ &= \frac{(\frac{s}{2})}{s} \cdot \frac{(\frac{s}{2}-1)}{s-1} \cdots \frac{(\frac{s}{4}+2)}{(\frac{3s}{4}+2)} \cdot \frac{(\frac{s}{4}+1)}{(\frac{s}{2}+1)}.\end{aligned}$$

Letting $t = s/4$, we find that the above equals

$$\frac{2t}{4t} \cdot \frac{2t-1}{4t-1} \cdots \frac{t+2}{3t+2} \cdot \frac{t+1}{2t+1} = \left(\prod_{i=2}^{t} \frac{t+i}{3t+i}\right) \cdot \frac{t+1}{2t+1}.$$

Now, for every $i < t$ it holds that $\frac{t+i}{3t+i} < \frac{1}{2}$ and thus the above is upper bound by $\frac{1}{2^{t-1}}$. Stated directly, we have for *every* s,

$$\Pr[\mathsf{noAbort} \wedge \mathsf{badMaj}] < \frac{1}{2^{s/4-1}} \tag{2}$$

completing the first part of the claim. We now proceed to the second part by using approximations that hold for values of s that are not too small. (Note that the above bound is quite wasteful because $\frac{t+2}{3t+2}$ is close to $1/3$ and we bounded it by $1/2$.) Writing

$$\prod_{i=2}^{t}(t+i) = \frac{(2t)!}{(t+1)!} \quad \text{and} \quad \prod_{i=2}^{t}(3t+i) = \frac{(4t)!}{(3t+1)!}$$

we have

$$\prod_{i=2}^{t} \frac{t+i}{3t+i} = \frac{(2t)!}{(t+1)!} \cdot \frac{(3t+1)!}{(4t)!}.$$

By Stirling's approximation $t! \approx \sqrt{2\pi t}(\frac{t}{e})^t$. Thus,

$$\frac{(2t)!}{(t+1)!} \approx \frac{\sqrt{2\pi 2t}(\frac{2t}{e})^{2t}}{\sqrt{2\pi(t+1)}(\frac{t+1}{e})^{t+1}} = \sqrt{\frac{2t}{t+1}} \cdot \frac{(2t)^{2t}}{(t+1)^{t+1}} \cdot \frac{e^{t+1}}{e^{2t}}$$

and

$$\frac{(3t+1)!}{(4t)!} \approx \sqrt{\frac{3t+1}{4t}} \cdot \frac{(3t+1)^{3t+1}}{(4t)^{4t}} \cdot \frac{e^{4t}}{e^{3t+1}}.$$

Putting the above together we have

$$\begin{aligned}
\prod_{i=2}^{t} \frac{t+i}{3t+i} &\approx \sqrt{\frac{2t}{4t} \cdot \frac{3t+1}{4t}} \cdot \frac{(2t)^{2t}}{(4t)^{4t}} \cdot \frac{(3t+1)^{3t+1}}{(t+1)^{t+1}} \cdot \frac{e^{4t}}{e^{2t}} \cdot \frac{e^{t+1}}{e^{3t+1}} \\
&\approx \frac{2^{2t}}{2^{8t}} \cdot \frac{t^{2t}}{t^{4t}} \cdot \frac{(3t)^{3t}}{t^t} \cdot \frac{e^{5t+1}}{e^{5t+1}} = \frac{1}{2^{6t}} \cdot \frac{1}{t^{2t}} \cdot 3^{3t} \cdot \frac{t^{3t}}{t^t} \\
&= \frac{1}{2^{6t}} \cdot \frac{1}{t^{2t}} \cdot 3^{3t} \cdot t^{2t} = \frac{3^{3t}}{2^{6t}} \\
&= \left(\frac{3}{4}\right)^{3t} \approx \frac{1}{2^{1.245t}}
\end{aligned}$$

where the second "approximate equality" is just due to removing the value in the square-root (which equals exactly $\sqrt{3/8}$) and some "+1" terms. Recalling that $t = s/4$ we conclude that

$$\Pr[\mathsf{noAbort} \wedge \mathsf{badMaj}] \approx \frac{1}{2^{1.245t}} = \frac{1}{2^{0.311s}} \tag{3}$$

completing the proof of the claim. □

We stress that the approximate bound is significantly better than $2^{-s/4}$. In particular, in order to obtain security of 2^{-40}, it suffices to set $s = 128$ rather than $s = 160$ (based on the fact that $0.311 \times 128 = 39.81$). Nevertheless, we proved the exact bound that holds for all s in order to later analyze a covert version of our protocol; see Sect. 5.2. Finally, before proceeding with the proof we remark that we checked the accuracy of this approximation by calculating the *exact result*. For $s = 128$ we have

$$\frac{\binom{\frac{3s}{4}+1}{\frac{s}{2}+1}}{\binom{s}{\frac{s}{2}}} = \frac{1}{2^{38.975}}$$

which is very close to what we expect. We stress that for an error of 2^{-40} using the exact bound, we find that s must actually be set to 132.

We now use Claim 4.3 to prove that the result of an ideal-model execution with $\mathcal{S}$ is (n, s)-indistinguishable from a real execution of the protocol with adversary $\mathcal{A}$. Specifically, we claim that as long as the event (noAbort $\wedge$ badMaj) does *not* occur, the result of the ideal and hybrid executions (where the oblivious transfer and zero-knowledge are ideal) are identically distributed. In order to see this, observe that if less than $s/4$ circuits are bad, then the majority of circuits evaluated by P_2 compute the correct circuit C which in turn computes f. In addition, by the ideal zero-knowledge and the fact that the $g^{a_i^0}, g^{a_i^1}, g^{r_j}$ values *fully determine* the garbled values associated with P_1's input bits, the input x derived by the simulator $\mathcal{S}$ and sent to the trusted party computing f corresponds exactly to the input x in the computation of every not-bad garbled circuit GC_j. Thus, in every not-bad circuit P_2 outputs $f(x, y)$, and these are a *majority* of the evaluation circuits. We conclude that as long as an abort does not occur, P_2 outputs $f(x, y)$ in both the real and ideal executions, and this corresponds exactly to the view of $\mathcal{A}$ in the executions. Finally, we observe that $\mathcal{S}$ sends $\perp$ to the trusted party whenever P_2 would abort and output $\perp$. (One subtlety to note is that an honest P_2 also outputs $\perp$ if circuit decryption fails for a majority of the evaluation circuits. However, this can only occur in the event of noAbort $\wedge$ badMaj which we are assuming does not occur.) This completes the proof of this corruption case.

P_2 Is Corrupted The intuition behind the security in this corruption case is simple. P_2 receives $s/2$ opened check circuits and $s/2$ evaluation circuits. For each of the evaluation circuits it receives only a single set of keys for decrypting the circuit. Furthermore, the keys that it receives for each of the $s/2$ evaluation circuits are associated with the same pair of inputs x and y. Regarding x, this is due to the fact that P_1 is honest. Regarding y, this is due to the fact that the oblivious transfer enforces "single choice" for the receiver. The above implies that P_2 can do nothing but decrypt $s/2$ circuits, where in each it obtains the same value $f(x, y)$ and learns nothing else. We stress one subtlety which is due to the fact that P_2 can try and send P_1 a different set $\mathcal{J}'$ to the set $\mathcal{J}$ that it input to the cut-and-choose oblivious transfer. If it succeeds in doing this, then there will be at least one evaluation circuit for which P_2 knows all of the keys associated with its input wires. In such a case, it could compute $f(x, y)$ for multiple values of y, and thus learn more than allowed about x. However, in order to successfully do this, a corrupt P_2 must send P_1 both of the keys associated with the input bit y_1 for a circuit GC_j where $j \notin \mathcal{J}$. Since both of these keys are random, and P_2 only learns one of them in the oblivious transfer (in the hybrid model the other key is completely unknown), it follows that it succeeds in doing this with only negligible probability. We now proceed to the formal proof.

Let $\mathcal{A}$ be an adversary controlling P_2 in an execution of Protocol 4.1 where a trusted party is used to compute the cut-and-choose OT functionality $\mathcal{F}_{\text{ccot}}^{S,B}$ and the zero-knowledge proof of knowledge of Step 7b. We construct a simulator $\mathcal{S}$ for the ideal model with a trusted party computing f as follows:

1. $\mathcal{S}$ invokes $\mathcal{A}$ upon its input and auxiliary input and receives the inputs that $\mathcal{A}$ sends to the trusted party computing the cut-and-choose OT functionality. These

inputs consist of a subset $\mathcal{J} \subset [s]$ of size exactly $s/2$ and bits $\sigma_1, \ldots, \sigma_\ell$. (If $\mathcal{J}$ is not of size exactly $s/2$ then $\mathcal{S}$ simulates P_1 aborting, sends $\perp$ to the trusted party computing f, and halts outputting whatever $\mathcal{A}$ outputs.)

2. $\mathcal{S}$ chooses an $\ell \times s$ matrix of random pairs of garbled values of length n: $(x^0_{i,j}, x^1_{i,j})$ for $i = 1, \ldots, \ell$ and $j = 1, \ldots, s$. Then, $\mathcal{S}$ hands $\mathcal{A}$ the appropriate values as its output from the oblivious transfers. Specifically, for every $i = 1, \ldots, \ell$ and $j \in \mathcal{J}$ the simulator $\mathcal{S}$ hands $\mathcal{A}$ the pair $(x^0_{i,j}, x^1_{i,j})$, and for every $i = 1, \ldots, \ell$ and $j \notin \mathcal{J}$ the simulator $\mathcal{S}$ hands $\mathcal{A}$ the value $x^{\sigma_i}_{i,j}$.
3. $\mathcal{S}$ sends $y = \sigma_1 \cdots \sigma_\ell$ to the trusted party computing f and receives back an output z.
4. For every $j \in \mathcal{J}$, $\mathcal{S}$ constructs GC_j as a correct garbled circuit, in the same way that an honest P_1 would construct it.
5. For every $j \notin \mathcal{J}$, $\mathcal{S}$ uses Lemma A.1 to construct a fake garbled circuit $\widetilde{GC}_j$ which always outputs z (where z is the output that $\mathcal{S}$ received from the trusted party). The keys associated with P_1's inputs in these fake garbled circuits are consistent with the $g^{a^0_i}, g^{a^1_i}, g^{r_j}$ values, as an honest party would use them.
6. $\mathcal{S}$ sends the garbled circuits and $(i, 0, g^{a^0_i})$, $(i, 1, g^{a^1_i})$ and (j, g^{r_j}) values to $\mathcal{A}$.
7. $\mathcal{S}$ receives back a set $\mathcal{J}'$ along with a pair of values $(x^0_{1,j}, x^1_{1,j})$ for every $j \in \mathcal{J}$:
   (a) If $\mathcal{J}' \neq \mathcal{J}$ and yet the values received are all correct then $\mathcal{S}$ outputs fail and halts. (This event happens with negligible probability.)
   (b) If $\mathcal{J}' = \mathcal{J}$ and any of the values received are incorrect, then $\mathcal{S}$ sends $\perp$ to the trusted party, simulates P_1 aborting, and halts outputting whatever $\mathcal{A}$ outputs.
   (c) Otherwise, $\mathcal{S}$ proceeds as below.
8. $\mathcal{S}$ hands $\mathcal{A}$ the values $\{r_j\}_{j \in \mathcal{J}}$, where r_j is as chosen above.
9. $\mathcal{S}$ hands $\mathcal{A}$ the keys $k'_{i,j} = g^{a^0_i \cdot r_j}$ for every $j \notin \mathcal{J}$ and $i = 1, \ldots, \ell$, and proves in zero-knowledge that all these keys are consistent with a single input. (Observe that these are the keys that $\mathcal{A}$ expects to receive in Step 7 of the protocol. However, instead of being the keys corresponding to x—which $\mathcal{S}$ does not know—they are the zero keys.)
10. After concluding, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs and halts.

Denoting Protocol 4.1, we now show that for every $\mathcal{A}$ corrupting P_2 and every s it holds that

$$\left\{\text{IDEAL}_{f,\mathcal{S}}(x, y, z, n, s)\right\}_{x,y,z \in \{0,1\}^*; n,s \in \mathbb{N}} \stackrel{\text{c}}{\equiv} \left\{\text{REAL}_{\pi,\mathcal{A}}(x, y, z, n, s)\right\}_{x,y,z \in \{0,1\}^*; n,s \in \mathbb{N}}$$

where $|x| = |y|$. (Note that here we prove standard indistinguishability in n, which holds for all values of s. That is, the value of s has no effect on the ability to distinguish.) We first remark that the probability that $\mathcal{S}$ outputs fail is negligible because for every $j \notin \mathcal{J}$ the adversary $\mathcal{A}$ receives only one of $x^0_{1,j}, x^1_{1,j}$ and these are random keys (note that the garbled circuits contain encryptions under these keys; however, by the security of encryption the probability that such a key can be obtained is negligible). We therefore ignore this event from now on and show that the ideal and real distributions are computationally indistinguishable conditioned on the event not happening.

Observe that the only difference with respect to $\mathcal{A}$'s view between the simulation and a real execution is regarding the construction of the garbled circuits GC_j for $j \notin \mathcal{J}$: in

the simulation these are fake garbled circuits $\widetilde{GC}_j$ outputting z (as received from the trusted party) and in the real execution these are real garbled circuits GC_j computing $f(x, y)$. Thus, by Lemma A.1 indistinguishability follows as long as in a real execution $\mathcal{A}$ obtains the garbled values corresponding to x for P_1's input wires and to y for P_2's input wires, where $z = f(x, y)$. Regarding y, this follows directly from the fact that $\mathcal{S}$ defines y to be the values $\sigma_1 \cdots \sigma_\ell$ which define *exactly* which garbled values $\mathcal{A}$ receives from the oblivious transfers. However, the case for x is more problematic since $\mathcal{A}$ receives all of the $g^{a_i^0}, g^{a_i^1}, g^{r_j}$ values which fully determine the garbled values corresponding to all of P_1's input wires. We therefore first modify $\mathcal{S}$ to $\mathcal{S}'$ so that in every $\widetilde{GC}_j$ for $j \notin \mathcal{J}$, $\mathcal{S}'$ uses $g^{a_i^0 \cdot r_j}$ to derive the value $k'_{i,j}$ that $\mathcal{S}$ hands $\mathcal{A}$ in Step 9 of the simulation, but uses a completely random value for the other key on that wire. The indistinguishability of these two simulations follows via a straightforward reduction to the decisional Diffie–Hellman problem. Once the simulation is with the modified $\mathcal{S}'$ we can apply Lemma A.1 and complete the proof. (Formally a hybrid argument is required for this step because here there are $s/2$ fake garbled circuits whereas Lemma A.1 refers to a single fake garbled circuit. Nevertheless observing that indistinguishability must hold even given x and y, this is a completely standard hybrid argument and so is omitted.) We stress that in order for the proof to hold, it must be that in every evaluation circuit $\mathcal{A}$ learns the garbled value associated with the same bit, for every input wire associated with P_2's input. However, this is guaranteed by the security of the *single-choice* cut-and-choose oblivious transfer. This completes the proof. □

4.3. *Exact Efficiency*

An analysis of the protocol yields the following number of exponentiations: (1) $2s\ell + s + 2\ell$ for the input-key preparation and commitments to the g^a and g^r values in Step 1, (2) $11.5s\ell + 19\ell + 9s + 5$ for the oblivious transfer of Step 2 (see analysis in Sect. 3.5), (3) $s\ell$ for P_2 to recompute in Step 6 the input keys associated with P_1's input, and (4) $s\ell/2 + 18\ell$ for the 2ℓ proofs of consistency for P_1's input bits in Step 7 (see analysis in Appendix B.3). Overall, we have $15s\ell + 39\ell + 10s + 5$ exponentiations.

Regarding the bandwidth, a similar count yields the exchange of $7s\ell + 22\ell + 7s + 5$ group elements and s copies of the garbled circuit. Finally, the protocol takes 12 rounds of communication (including the oblivious transfer and zero-knowledge proof). We conclude that there are $15s\ell + 39\ell + 10s + 5$ *exponentiations*, $7s\ell + 22\ell + 7s + 5$ *group elements* sent and 12 *rounds of communication*. In addition, there are $6.5|C|s$ *symmetric encryptions*, comprised of $4|C|s$ encryptions for constructing all s garbled circuits, $4|C|0.5s$ encryptions for P_2 to check $s/2$ of them, and $|C|0.5s$ encryptions for P_2 to compute the $s/2$ evaluation circuits. Finally, there are $4|C|s$ *ciphertexts* sent for transmitting these circuits.

We stress that this analysis describes the overhead of the basic protocol. The overhead can be substantially improved by different optimizations, as shown in Sect. 4.4 below.

4.4. *Optimizations*

Fixed-Base Exponentiations Exponentiations are commonly computed by repeated squaring, which for a group of order q of length n bits requires on average $1.5n$ multiplications for a full exponentiation. If multiple exponentiations of the same base are

computed, then the repeated binary powers of the base can be computed once for all exponentiations, reducing the amortized overhead of an exponentiation on average to $0.5n$ multiplications (see [31], 14.6.3). Let us examine how this affects the overhead of the protocol (taking into account only the $s\ell$ component of the overhead, which is the most significant): (1) P_1 preparing its input keys in Step 1 requires $2s\ell$ exponentiations which are fixed-base. Their amortized overhead is therefore equivalent to that of about $2/3s\ell$ exponentiations. (2) Of the $11.5s\ell$ exponentiations of the batch single-choice cut-and-choose OT of Step 2, $10.5s\ell$ exponentiations are performed with fixed bases (these are the exponentiations in the *RAND* operation, the computation of h_j by the receiver and half of the exponentiations in the zero-knowledge protocol in the transfer phase of the single-choice cut-and-choose oblivious transfer). Therefore the amortized overhead is equivalent to that of about $(10.5/3 + 1)s\ell = 4.5s\ell$ exponentiations. (3) In Step 6, P_2 uses $s\ell$ exponentiations to compute P_1's input keys in check circuits. They are all fixed-base and therefore are equivalent to about $s\ell/3$ exponentiations. (4) P_1 proving the consistency of its inputs in Step 7 takes about $s\ell/2$ exponentiations which are all fixed-base. They are therefore equivalent to $s\ell/6$ full exponentiations. The overall overhead of the exponentiations is therefore equivalent to that of about $5.66s\ell$ *full exponentiations.*

Reducing the Computation of P_2 in Step 6 In Step 6 of Protocol 4.1, P_2 performs $s\ell$ exponentiations in order to compute the garbled values associated with P_1's input in the check circuits. Namely, given the $(i, 0, g^{a_i^0})$, $(i, 1, g^{a_i^1})$ tuples and r_j for every $j \in \mathcal{J}$, party P_2 computes $g^{a_i^0 \cdot r_j}, g^{a_i^1 \cdot r_j}$ for all $i = 1 \ldots \ell$ and $j \in \mathcal{J}$. This step costs $s\ell$ exponentiations (2ℓ exponentiations for each of the $s/2$ check circuits). As we will see, we can reduce this to about a quarter by having P_1 send the $g^{a_i^0 \cdot r_j}, g^{a_i^1 \cdot r_j}$ values to P_2 and prove that they are correct (not in zero-knowledge). We remark that P_1 has to compute these values in order to construct the circuit and so this results in no additional computation overhead by P_1 (the only addition is in communication bandwidth).

The protocol is modified by changing Step 6 as follows (recall that P_2 already has all of the $(i, 0, g^{a_i^0})$, $(i, 1, g^{a_i^1})$ tuples and r_j values):

1. P_1 sends P_2 all of the values $k'^0_{i,j} = g^{a_i^0 \cdot r_j}$ and $k'^1_{i,j} = g^{a_i^1 \cdot r_j}$ for $i = 1, \ldots, \ell$ and $j \in \mathcal{J}$.
2. P_2 chooses random values $\gamma_i^0, \gamma_i^1 \in [1, 2^L]$ for $i = 1, \ldots, \ell$.
3. For every $j \in \mathcal{J}$, party P_2 computes the two values

$$\alpha_j = \left(\prod_{i=1}^{\ell} \left(g^{a_i^0}\right)^{\gamma_i^0} \cdot \left(g^{a_i^1}\right)^{\gamma_i^1} \right)^{r_j} \quad \text{and} \quad \beta_j = \prod_{i=1}^{\ell} \left(k'^0_{i,j}\right)^{\gamma_i^0} \cdot \left(k'^1_{i,j}\right)^{\gamma_i^1}.$$

   Note that computing α_j requires only a single *full* exponentiation since the value $(g^{a_i^0})^{\gamma_i^0} \cdot (g^{a_i^1})^{\gamma_i^1}$ can be computed once for all j.
4. P_2 accepts P_1's input if and only if $\alpha_j = \beta_j$ for all $j \in \mathcal{J}$.

We first prove that this method is secure.

Claim 4.4. *The probability that P_2 accepts if there exists an $i \in \{1, \ldots, \ell\}$ and $j \in \mathcal{J}$ such that $k'^0_{i,j} \neq g^{a_i^0 \cdot r_j}$ or $k'^1_{i,j} \neq g^{a_i^1 \cdot r_j}$ is at most $\frac{s}{2} \cdot 2^{-L}$.*

Proof. Assume there exists an $i^* \in \{1, \ldots, \ell\}$ and $j \in \mathcal{J}$ such that $k'^0_{i^*,j} \neq g^{a_{i^*}^0 \cdot r_j}$ or $k'^1_{i^*,j} \neq g^{a_{i^*}^1 \cdot r_j}$. Let

$$\alpha_j^{\neg i^*} = \left(\prod_{i \neq i^*} \left(g^{a_i^0}\right)^{\gamma_i^0} \cdot \left(g^{a_i^1}\right)^{\gamma_i^1} \right)^{r_j} \quad \text{and} \quad \beta_j^{\neg i^*} = \prod_{i \neq i^*} \left(k'^0_{i,j}\right)^{\gamma_i^0} \cdot \left(k'^1_{i,j}\right)^{\gamma_i^1}$$

implying that

$$\alpha_j = \alpha_j^{\neg i^*} \cdot \left(g^{a_{i^*}^0 \cdot \gamma_{i^*}^0} \cdot g^{a_{i^*}^1 \cdot \gamma_{i^*}^1}\right)^{r_j} = \alpha_j^{\neg i^*} \cdot \left(g^{a_{i^*}^0 \cdot r_j}\right)^{\gamma_{i^*}^0} \cdot \left(g^{a_{i^*}^1 \cdot r_j}\right)^{\gamma_{i^*}^1}$$

and

$$\beta_j = \beta_j^{\neg i^*} \cdot \left(k'^0_{i^*,j}\right)^{\gamma_{i^*}^0} \cdot \left(k'^1_{i^*,j}\right)^{\gamma_{i^*}^1}.$$

Now, $\alpha_j = \beta_j$ if and only if

$$\alpha_j^{\neg i^*} \cdot \left(g^{a_{i^*}^0 \cdot r_j}\right)^{\gamma_{i^*}^0} \cdot \left(g^{a_{i^*}^1 \cdot r_j}\right)^{\gamma_{i^*}^1} = \beta_j^{\neg i^*} \cdot \left(k'^0_{i^*,j}\right)^{\gamma_{i^*}^0} \cdot \left(k'^1_{i^*,j}\right)^{\gamma_{i^*}^1}$$

which in turn holds if and only if

$$\left(\frac{g^{a_{i^*}^0 \cdot r_j}}{k'^0_{i^*,j}}\right)^{\gamma_{i^*}^0} \cdot \left(\frac{g^{a_{i^*}^1 \cdot r_j}}{k'^1_{i^*,j}}\right)^{\gamma_{i^*}^1} = \frac{\beta_j^{\neg i^*}}{\alpha_j^{\neg i^*}}.$$

Assume now that $g^{a_{i^*}^0 \cdot r_j} \neq k'^0_{i^*,j}$. Equality holds if and only if

$$\left(\frac{g^{a_{i^*}^0 \cdot r_j}}{k'^0_{i^*,j}}\right)^{\gamma_{i^*}^0} = \frac{\beta_j^{\neg i^*}}{\alpha_j^{\neg i^*}} \cdot \left(\frac{k'^1_{i^*,j}}{g^{a_{i^*}^1 \cdot r_j}}\right)^{\gamma_{i^*}^1}.$$

Fixing all of the γ values first, and then choosing $\gamma_{i^*}^0$ at random, we see that this holds with probability at most 2^{-L}. (Note that this assumes that $g^{a_{i^*}^0 \cdot r_j}/k'^0_{i^*,j}$ is a generator. However, in a group of prime order, *all* elements apart from the unity are generators.) Applying the union bound for all $j \in \mathcal{J}$ we obtain the claim. □

We now analyze the efficiency improvement gained by this method. The communication overhead is slightly increased over the basic version of the protocol, since $s\ell$ values, namely all garbled values of P_1's inputs in the check circuits, are sent. This is not significant when the circuit is not very small. Regarding the computation overhead, we find that P_2 now has to perform $s\ell$ exponentiations in order to compute the β_j values (2ℓ exponentiations for $s/2$ values of j) with an exponent which is only L bits long. (Unfortunately, these exponentiations are not fixed-base.) Assuming that exponentiations are carried out in an elliptic curve group of order 2^{160} and that $L = 40$, we see that

the cost of the exponentiations is reduced by a factor of 4. Thus, instead of $s\ell$ exponentiations, we effectively have $s\ell/4$ exponentiations. This is a mild improvement over the $s\ell/3$ cost when using the fixed-base optimization.

Preprocessing The bulk of the exponentiations performed in the protocol can be precomputed. Step 1 of the protocol, where P_1 computes its input keys, can clearly be computed before P_1 receives its inputs. Step 2 executes the oblivious transfers. It can be slightly changed to be run before P_2 receives its inputs: P_2 can execute this step with random inputs $\sigma_1, \ldots, \sigma_\ell$. Then, when it receives its input bits $y_1, \ldots, y_\ell$, it sends P_1 a string of correction bits $y_1 \oplus \sigma_1, \ldots, y_\ell \oplus \sigma_\ell$. P_1 exchanges the roles of the two keys of input wires of P_2 for which it receives a correction bit with the value 1. (The security proof can be easily adapted for this variant of the protocol.) Given this change, both Steps 1 and 2 can be precomputed. These steps account for $13.5s\ell$ of the $15s\ell$ exponentiations of the protocol, where the remaining $1.5s\ell$ exponentiations are fixed-base. This means that if preprocessing is used, then after receiving their inputs the parties need to effectively compute only $s\ell/2$ *full exponentiations*.

Parallel Computation by the Two Parties Many of the computations can be carried out in parallel by the different parties. For example, in the oblivious transfer protocol (Protocol 3.3), after the sender receives the g_0, g_1, h_0^j, h_1^j values from the receiver in Step 4 of the setup phase it can begin carrying out half of the *RAND* computations while the receiver continues its other computations. Similarly, in each transfer phase the sender can send its messages to the receiver one by one, as it computes them. The receiver can begin decrypting the first message immediately as it arrives, in parallel to the sender computing the following messages.

Bandwidth vs. Computation It was shown in [16] that it is possible to reduce the bandwidth of sending the circuits by about 50%, in the following way: P_1 generates each circuit as a deterministic function of a different seed; P_1 first commits to the circuits, and then, instead of sending the tables of check circuits, it only sends the seeds from which these circuits were computed. This reduction in communication comes at the cost of P_2 having to generate long pseudo-random strings from seeds, and repeating P_1's task of generating the circuits. This optimization might be useful, though, in settings in which communication is expensive (e.g., for a roaming cellular user).

5. Variants—Universal Composability and Covert Adversaries

5.1. *Universally Composable Two-Party Computation*

We observe that the simulators in the proof of Theorem 4.2 carry out no rewinding, and likewise the intermediate simulators used to prove the reductions. Thus, if the protocols used to compute the batch cut-and-choose oblivious transfer functionality and the zero-knowledge proof of knowledge of Step 7b are universally composable, then so is Protocol 4.1. Furthermore, the protocol for computing the oblivious transfer is universally composable if the zero-knowledge proof carried out by the receiver in the setup phase is universally composable. Both that proof and the proof of Step 7b are essentially

the proof of Protocol B.2. Thus, all we need is a universally composable variant of the zero-knowledge proof of Protocol B.2 and the *entire* protocol is universally composable. In order to do this, we need an efficient transformation from Σ protocols to universally composable zero-knowledge proofs of knowledge. Efficient constructions of universally composable zero-knowledge have been considered in [10,12,30], but do not work efficiently for all Σ protocols. Another approach was presented recently in [20]. We use this protocol, and for the sake of completeness (and to facilitate an exact complexity analysis) we present it in Appendix C. As we will see below, the cost of the exponentiations is dominated by $2s^2$ Diffie–Hellman group exponentiations. We summarize the above in the following theorem (the number of rounds is fewer here because the zero-knowledge protocol has fewer rounds in the model of universal composability):

Theorem 5.1. *Assume that the decision Diffie–Hellman assumption holds. Then, for every efficiently computable two-party function f with inputs of length ℓ, there exists a universally composable protocol that securely computes f in the presence of malicious adversaries in the common reference string model, with eight rounds of computation and $O(s\ell + s^2)$ exponentiations.*

Efficiency of the UC Variant of Protocol 4.1 We begin by translating the above into the cost of running a universally composable zero-knowledge protocol for *subset DH tuples* (see Protocol B.2). However, note that we only need to obtain security of $2^{-s/4}$ and therefore we use $L = s/4$ in the transformation of Appendix C. Now, the number of exponentiations in the $s/2$ out of s variant Protocol B.2 without the verifier commitment (because we only need the basic Σ protocol portion of it) is exactly $7s$. As we have seen in Appendix C, the cost of converting this to a universally composable zero-knowledge protocol, using the UC commitment scheme of [27], is L times the cost of the Σ protocol plus $57L$ group exponentiations. Thus, with $L = s/4$ we find that the zero-knowledge protocols costs $7sL + 57L = \frac{7}{4} \cdot s^2 + \frac{57}{4} \cdot s$ exponentiations.

Our protocol for two-party computation also uses a 1-out-of-2 variant of Protocol B.2. In this case, the cost is 14 group exponentiations for a single execution of the Σ protocol. Setting $L = s/4$ we have $14L + 57L = 71L = \frac{71s}{4} < 18s$ group exponentiations. In Protocol 4.1 the $s/2$ out of s variant is run once, while the 1-out-of-2 variant is run ℓ times. We therefore conclude that the universally composable version of Protocol 4.1 has an *additional* $\frac{7}{4} \cdot s^2 + \frac{57}{4} \cdot s + 18s\ell$ *group exponentiations*, beyond the cost of the basic version that is only secure in the stand-alone model. Although this is certainly not "for free" it is not overly prohibitive. We remark that the universally composable zero-knowledge proofs require three rounds of communication, rather than five rounds for the stand-alone versions. Since two of these are run at separate phases, we have see that this takes four rounds off the round complexity of Protocol 4.1, yielding a total of eight rounds.

5.2. *Covert Security*

In the model of security in the presence of covert adversaries [1], the requirement is that any cheating by an adversary will be caught with some probability ϵ. The value of ϵ taken depends on the application, the ramifications to an adversary being caught, the value to an adversary when it cheats successfully (if not caught) and so on. The analysis

of our protocol shows that for *every* value of s (even if s is very small) the probability that an adversary can cheat without being caught is at most $2^{-\frac{s}{4}+1}$; see Claim 4.3 and a discussion below. This immediately yields a protocol that is secure in the presence of covert adversaries, as stated in the following theorem.

Theorem 5.2. *Assume that the decisional Diffie–Hellman assumption is hard in* $\mathbb{G}$, *that the protocol used in Step* 2 *securely computes the batch single-choice cut-and-choose oblivious transfer functionality*, *that the protocol used in Step* 7b *is a zero-knowledge proof of knowledge*, *and that the symmetric encryption scheme used to generate the garbled circuit is secure. Then*, *for any integer* $s > 4$, *Protocol* 4.1 *securely computes the function* f *in the presence of covert adversaries with* ϵ*-deterrent* (*under the strong explicit cheat formulation*), *for* $\epsilon = 1 - 2^{-\frac{s}{4}+1}$.

We stress that our protocol is significantly more efficient than the protocols of [1] and [16] when values of ϵ that are greater than $1/2$ are desired. For example, in order to obtain an ϵ-deterrent of 0.99, the protocol of [1] requires using 100 garbled circuits. However, taking $s = 30$ in our protocol here yields an ϵ-deterrent of about 0.99, and so it suffices to send 30 circuits and not 100, reducing the cost by more than a factor of 3.

A Tighter Analysis As we saw in the proof of Claim 4.3, the probability of cheating is actually significantly lower than $2^{-s/4+1}$. For small values of s the exact probability of cheating, given in Claim 4.3, is $\binom{3s/4+1}{s/2+1}/\binom{s}{s/2}$. Based on this formula, with $s = 24$ we obtain an ϵ-deterrent of 0.99, and so the cost is actually less than $1/4$ of the protocol of [1], which is very significant.

Acknowledgements

We thank Bo Zhang for pointing out an error in the single-choice cut-and-choose oblivious transfer protocol in an earlier version of this work.

Research of first author was generously supported by the European Research Council as part of the ERC project LAST, and by the ISRAEL SCIENCE FOUNDATION (grant No. 781/07). Research of second author was generously supported by the European Research Council as part of the ERC project SFEROT, and by the CACE project funded by the European Union.

Appendix A. Yao's Protocol—Semi-honest Adversaries

We describe here the construction of secure two-party computation (for semi-honest adversaries) which is described in [41]. This construction is based on Yao construction. It is proved in [29] to be secure against semi-honest adversaries.

Let C be a Boolean circuit that receives two inputs $x, y \in \{0,1\}^\ell$ and outputs $C(x, y) \in \{0,1\}^\ell$ (for simplicity, we assume that the input length, output length and the security parameter are all of the same length ℓ). We also assume that C has the property that if a circuit output wire comes from a gate g, then gate g has no wires that

are input to other gates.[3] (Likewise, if a circuit input wire is itself also a circuit output, then it is not input into any gate.)

We begin by describing the construction of a single garbled gate g in C. The circuit C is Boolean, and therefore any gate is represented by a function $g : \{0,1\} \times \{0,1\} \to \{0,1\}$. Now, let the two input wires to g be labeled w_1 and w_2, and let the output wire from g be labeled w_3. Furthermore, let $k_1^0, k_1^1, k_2^0, k_2^1, k_3^0, k_3^1$ be six keys obtained by independently invoking the key-generation algorithm $G(1^\ell)$; for simplicity, assume that these keys are also of length ℓ. Intuitively, we wish to be able to compute $k_3^{g(\alpha,\beta)}$ from k_1^α and k_2^β, without revealing any of the other three values $k_3^{g(1-\alpha,\beta)}, k_3^{g(\alpha,1-\beta)}, k_3^{g(1-\alpha,1-\beta)}$. The gate g is defined by the following four values:

$$c_{0,0} = E_{k_1^0}\big(E_{k_2^0}\big(k_3^{g(0,0)}\big)\big), \qquad c_{0,1} = E_{k_1^0}\big(E_{k_2^1}\big(k_3^{g(0,1)}\big)\big),$$
$$c_{1,0} = E_{k_1^1}\big(E_{k_2^0}\big(k_3^{g(1,0)}\big)\big), \qquad c_{1,1} = E_{k_1^1}\big(E_{k_2^1}\big(k_3^{g(1,1)}\big)\big)$$

where E is from a private key encryption scheme (G, E, D) that has indistinguishable encryptions for multiple messages, and has an elusive efficiently verifiable range. Informally, this means (1) that for any two (known) messages x and y, no polynomial-time adversary can distinguish between the encryptions of x and y, and (2) that there is a negligible probability that an encryption under one key falls into the range of encryptions under another key, and given a key k it is easy to verify whether a certain ciphertext is in the range of encryptions with k. See [29] for a detailed discussion of these properties, and for examples of easy implementations satisfying them. For example, the encryption scheme could be $E_k(m) = \langle r, f_k(r) \oplus m0^n \rangle$, where f_k is a pseudo-random function keyed by k whose output is $|m| + n$ bits long, and r is a randomly chosen value.

Now, the actual gate is defined by a *random permutation* of the above values, denoted as c_0, c_1, c_2, c_3; from here on we call them the garbled table of gate g. Notice that given k_1^α and k_2^β, and the values c_0, c_1, c_2, c_3, it is possible to compute the output of the gate $k_3^{g(\alpha,\beta)}$ as follows. For every i, compute $D_{k_2^\beta}(D_{k_1^\alpha}(c_i))$. If more than one decryption returns a non-$\perp$ value, then output abort. Otherwise, define k_3^γ to be the only non-$\perp$ value that is obtained. (Notice that if only a single non-$\perp$ value is obtained, then this will be $k_3^{g(\alpha,\beta)}$ because it is encrypted under the given keys k_1^α and k_2^β. Later we will show that except with negligible probability, only one non-$\perp$ value is indeed obtained.)

We are now ready to show how to construct the entire garbled circuit. Let m be the number of *wires* in the circuit C, and let $w_1, \ldots, w_m$ be labels of these wires. These labels are all chosen uniquely with the following exception: if w_i and w_j are both output wires from the same gate g, then $w_i = w_j$ (this occurs if the fan-out of g is greater than one). Likewise, if an input bit enters more than one gate, then all circuit input wires associated with this bit will have the same label. Next, for every label w_i, choose two independent keys $k_i^0, k_i^1 \leftarrow G(1^\ell)$; we stress that all of these keys are chosen independently of the others. Now, given these keys, the four garbled values of each gate

[3] This requirement is due to our labeling of gates described below, which does not provide a unique label to each wire (see [29] for more discussion). We note that this assumption on C increases the number of gates by at most ℓ.

are computed as described above and the results are permuted randomly. Finally, the output or decryption tables of the garbled circuit are computed. These tables simply consist of the values $(0, k_i^0)$ and $(1, k_i^1)$ where w_i is a *circuit output wire*. (Alternatively, output gates can just compute 0 or 1 directly. That is, in an output gate, one can define $c_{\alpha,\beta} = E_{k_1^\alpha}(E_{k_2^\beta}(g(\alpha, \beta)))$ for every $\alpha, \beta \in \{0, 1\}$.)

The entire garbled circuit of C, denoted GC, consists of the garbled table for each gate and the output tables. We note that the structure of C is given, and the garbled version of C is simply defined by specifying the output tables and the garbled table that belongs to each gate. This completes the description of the garbled circuit.

Let $x = x_1 \cdots x_\ell$ and $y = y_1 \cdots y_\ell$ be two ℓ-bit inputs for C. Furthermore, let $w_1, \ldots, w_\ell$ be the input labels corresponding to x, and let $w_{\ell+1}, \ldots, w_{2\ell}$ be the input labels corresponding to y. It is shown in [29] that given the garbled circuit GC and the strings $k_1^{x_1}, \ldots, k_\ell^{x_\ell}, k_{\ell+1}^{y_1}, \ldots, k_{2\ell}^{y_\ell}$, it is possible to compute $C(x, y)$, except with negligible probability. The way that these values are obtained is as follows. For every bit of P_1's input, party P_1 just sends P_2 the appropriate keys $k_1^{x_1}, \ldots, k_\ell^{x_\ell}$. Furthermore, for every bit y_i of P_2's input, the parties run an oblivious transfer protocol where P_1 inputs $(k_{\ell+i}^0, k_{\ell+i}^1)$ and P_2 inputs y_i. The result is that P_2 obtains the keys $k_{\ell+1}^{y_1}, \ldots, k_{2\ell}^{y_\ell}$ and can therefore compute the garbled circuit. The crucial observation is that P_1 learns nothing of P_2's input by the security of the oblivious transfer, and P_2 learns nothing but the output because the circuit is encrypted and the keys $k_1^{x_1}, \ldots, k_\ell^{x_\ell}$ keep the identity of the bits $x_1, \ldots, x_\ell$ secret.

A Useful Lemma In [28], the following lemma is proven (that is actually implicit already in [29]). The lemma states that it is possible to build a fake garbled circuit that outputs a fixed value $z = f(x, y)$, and this is indistinguishable to an adversary who has only a single set of keys that correspond to the inputs x and y. The simulator that we construct in order to prove the security of our protocol constructs such "fake" circuits, and we therefore rely on this lemma in our proof.

Lemma A.1. *Given a circuit C and an output value z (of the same length as the output of C) it is possible to construct a garbled circuit $\widetilde{GC}$ such that we have the following.*

1. *The output of $\widetilde{GC}$ is always z, regardless of the garbled values that are provided for P_1 and P_2's input wires, and*
2. *If $z = f(x, y)$, then no non-uniform prob. polynomial-time adversary $\mathcal{A}$ can distinguish between the distribution ensemble consisting of $\widetilde{GC}$ and a single arbitrary garbled value for every input wire, and the distribution ensemble consisting of a real garbled version of C, together with garbled values that correspond to x for P_1's input wires and to y for P_2's input wires.*

Appendix B. Zero-Knowledge Proofs of Knowledge

In this section we present zero-knowledge proofs that we need in our protocols. The reason for presenting the protocols in full detail is to facilitate an exact efficiency analysis.

B.1. *Zero-Knowledge Proof of Knowledge—Diffie–Hellman Tuples*

A zero-knowledge proof of knowledge of Diffie–Hellman tuples is presented in Figure B.1. The fact that this is a proof of knowledge (and not just a zero-knowledge proof system) has been proven in [19, Chap. 6].

PROTOCOL B.1 ZK Proof of Knowledge of Diffie–Hellman Tuples.

- **Joint statement:** The values $(\mathbb{G}, g_0, g_1, u, v)$ that are elements of a group $\mathbb{G}$ of known order q, and g is the generator.
- **Auxiliary input for the prover:** A witness w such that $u = (g_0)^w$ and $v = (g_1)^w$.
- **The protocol:**
  1. P chooses a random $a \in \{1, \ldots, q\}$, computes $\alpha = (g_0)^a$ and sends α to V.
  2. The verifier V chooses random $s, t \in \{1, \ldots, q\}$, computes $c = (g_0)^s \cdot \alpha^t$ and sends c to P (this is a perfectly hiding commitment to s).
  3. P chooses a random $r \in \{1, \ldots, q\}$ and computes $A = (g_0)^r$ and $B = (g_1)^r$. It then sends (A, B) to V.
  4. V sends s and t as above to P.
  5. P verifies that $c = (g_0)^s \cdot \alpha^t$. If no, it aborts. Otherwise, it sends $z = s \cdot w + r$ to V. In addition, it sends a as chosen above.
  6. V accepts if and only if $\alpha = (g_0)^a$, $A = (g_0)^z / u^s$ and $B = (g_1)^z / v^s$.

Exact Efficiency The overall number of exponentiations in the protocol is 12. However, eight of these are of the form $x^a \cdot y^b$ (observe that the verification of A and B by V in the last step is actually also of this form). Each of these double exponentiations costs only 1.25 the cost of a standard exponentiation [31, Alg. 14.88], and thus we conclude that the protocol requires nine exponentiations overall. In addition, it takes five rounds of communication and involves the exchange of 8 group elements (where we count s, t, z, a as group elements even though they are actually smaller).

B.2. *Zero-Knowledge for Subset DH*

The protocol below uses the technique of [7] in order to prove that half of a given set of values are of the Diffie–Hellman form. We assume familiarity with the technique of [7] here. We remark that in Protocol B.2, it is possible to use *any* perfect secret sharing with the properties that are defined in [7]. The secret-sharing scheme of Shamir [40] works for example, but [7] have a more efficient scheme based on matrix multiplication. The

following proposition is common knowledge and follows from [7], and the proof is therefore omitted.

PROTOCOL B.2 ZK Proof of Knowledge of Subset of Diffie–Hellman Tuples.

[*The set I that is used here is the complement of the set $\mathcal{J}$ in Protocol 3.3.*]

- **Joint statement:** The description $\mathbb{G}$ of a group of order q with generators g_0, g_1. In addition, s pairs $(h_0^1, h_1^1), \ldots, (h_0^s, h_1^s)$
- **Auxiliary input for the prover:** A witness set of $s/2$ values $W = \{(i_j, w_{i_j})\}$ where the set of indices i_j, denoted I, are such that for every $i_j \in I$ it holds that $h_0^{i_j} = (g_0)^{w_{i_j}}$ and $h_1^{i_j} = (g_1)^{w_{i_j}}$.
- **The protocol:**
  1. **Set up commitment to verifier challenge:**
     (a) The prover P chooses a random $a \leftarrow \mathbb{Z}_q$, computes $\alpha = (g_0)^a$ and sends α to V.
     (b) The verifier V chooses random $t, c \leftarrow \mathbb{Z}_q$, computes $C = (g_0)^c \cdot \alpha^t$ and sends C to P (this is a perfectly hiding commitment to c).
  2. **Prover message 1:**
     (a) For every $i \notin I$, the prover P chooses random values $c_i \leftarrow \mathbb{Z}_q$ and $z_i \leftarrow \mathbb{Z}_q$ and sets $A_i = \frac{(g_0)^{z_i}}{(h_0^i)^{c_i}}$ and $B_i = \frac{(g_1)^{z_i}}{(h_1^i)^{c_i}}$.
     (b) For every $i \in I$, the prover chooses random $\rho_i \leftarrow \mathbb{Z}_q$ and sets $A_i = (g_0)^{\rho_i}$ and $B_i = (g_1)^{\rho_i}$.
     (c) P sends $(A_1, B_1), \ldots, (A_s, B_s)$ to V.
  3. **Verifier query:** V sends t, c as above.
  4. **Prover message 2:**
     (a) P checks that $C = (g_0)^c \cdot \alpha^t$ and aborts if not. Otherwise, it takes c as the verifier query.
     (b) The values $\{c_i\}_{i \notin I}$ and the verifier query c are interpreted as $s/2$ shares $\{c_i\}$ and a secret c in a secret sharing scheme with s participants and threshold $s/2+1$. Thus, these values fully define all shares for all participants (when an appropriate secret sharing scheme is used). The prover computes these shares $c_1, \ldots, c_s$ and sends them to the verifier V.
     (c) For every $i \notin I$, P sends z_i as chosen above.
     (d) For every $i \in I$, P sends $z_i = c_i \cdot w_i + \rho_i$.
     (e) Finally, P sends a as chosen in the first step.
  5. **Verifier validation:** V accepts if and only if all the following hold:
     (a) $\alpha = g^a$
     (b) The shares $c_1, \ldots, c_s$ define the secret c
     (c) For *all* $i = 1, \ldots, s$ it holds that $A_i = \frac{(g_0)^{z_i}}{(h_0^i)^{c_i}}$ and $B_i = \frac{(g_1)^{z_i}}{(h_1^i)^{c_i}}$.

Proposition B.3. *Protocol B.2 is a zero-knowledge proof of knowledge that is secure under parallel composition, for the relation*

$$R = \left\{\left(\mathbb{G}, q, g_0, g_1, \left(h_0^1, h_1^1\right), \ldots, \left(h_0^s, h_1^s\right)\right)\right\}$$

where $\mathbb{G}$ is a group of order q with generators g_0, g_1, and there exists a set of at least $s/2$ values $w_{i_1}, \ldots, w_{i_{s/2}}$ such that for every $j = 1, \ldots, s/2$ it holds that $h_0^{i_j} = (g_0)^{w_{i_j}}$ and $h_1^{i_j} = (g_1)^{w_{i_j}}$.

A One-Out-of-Two Variant We also need to prove that one out of two tuples is a Diffie–Hellman tuple. The idea is basically as in Protocol B.2 with $s = 2$. This case is simpler since instead of using a secret sharing scheme the parties work in the following way:

1. The prover chooses c_i as it wishes for the statement i whose proof it wants to forge;
2. The verifier sends the receiver a random c;
3. The prover sets $c_{2-i} = c_i + c$.

Exact Efficiency The overall cost of the protocol is $7s + 4$ exponentiations and the exchange of $3s + 4$ group elements. In addition, it takes five rounds of communication (note that the first and last round of communication can be combined with messages from the calling protocol). In the one-out-of-two variant, the count is identical by setting $s = 2$.

B.3. *ZK Proof for Extended Diffie–Hellman Tuples*

A zero-knowledge proof of an extended Diffie–Hellman tuple is given in Protocol B.4. The input is a tuple $(g_0, g_1, h_0, h_1, u_1, v_1, \ldots, u_\eta, v_\eta)$ such that either all $\{(g_0, h_0, u_i, v_i)\}_{i=1}^{\eta}$ are Diffie–Hellman tuples, or all $\{(g_1, h_1, u_i, v_i)\}_{i=1}^{\eta}$ are Diffie–Hellman tuples. This zero-knowledge proof is used twice; once in the single-choice cut-and-choice oblivious transfer protocol in Sect. 3.4, and once in the main protocol itself (Protocol 4.1 in Sect. 4.1). In the latter use (in Protocol 4.1), the same g is used in all tuples. That is, one sets $g = g_0 = g_1$ in Protocol B.4 below.

PROTOCOL B.4 ZK Proof of Knowledge of Extended Diffie–Hellman Tuple.

- **Common input:** $(g_0, g_1, h_0, h_1, u_1, v_1, \ldots, u_\eta, v_\eta)$ where g_0 and g_1 are generators of a group of *prime* order q.
- **Prover witness:** a such that either $h_0 = (g_0)^a$ and $v_i = (u_i)^a$ for all i, or $h_1 = (g_1)^a$ and $v_i = (u_i)^a$ for all i.
- **The protocol:**
  1. The verifier V chooses $\gamma_1, \ldots, \gamma_\eta \in_R \{0, 1\}^L$ where $2^L < q$, and sends the values to the prover.
  2. The prover and verifier locally compute
  $$u = \prod_{i=1}^{\eta} (u_i)^{\gamma_i} \quad \text{and} \quad v = \prod_{i=1}^{\eta} (v_i)^{\gamma_i}$$
  3. The prover proves in zero-knowledge that either (g_0, h_0, u, v) or (g_1, h_1, u, v) is a Diffie–Hellman tuple, and V accepts if and only it accepts in this subproof. (This subproof is the 1-out-of-2 variant of Protocol B.2.)

The zero-knowledge property of the protocol follows directly from the zero-knowledge property of the subprotocol for proving Diffie–Hellman. The soundness follows from the following claim.

Claim B.5. *If there exists an index* $1 \leq j \leq \eta$ *such that* (g, h, u_j, v_j) *is not a Diffie–Hellman tuple, then for every choice of* $\{\gamma_i\}_{i \neq j}$ *there exists at most one value* γ_j *such that* (g, h, u, v) *is a Diffie–Hellman tuple.*

The proof of this claim is almost identical to the proof of Claim 4.4 and is therefore omitted.

Corollary B.6. *The soundness error of Protocol B.4 is* $2^{-L} + \mu(n)$, *where* $\mu(n)$ *is the soundness error of the Diffie–Hellman tuple subproof.*

Exact Efficiency The cost of this protocol is exactly the cost of the 1-out-of-2 variant of Protocol B.2 (which costs 18 exponentiations and the exchange of 10 group elements) together with an additional 2η "short" 40-bit exponentiations by each party. Under the assumption that we work in an elliptic curve group with order q of length 160 bits, we find that this is equivalent to $\eta/2$ exponentiations by each party. Thus, the result is η additional exponentiations overall. We remark that no additional rounds are needed because the γ values can be sent together with the verifier's first message (this is because the prover's first message is independent of the input statement). We conclude that there are $\eta + 18$ *exponentiations*, the exchange of 10 *group elements* and *five rounds of communication*. When this zero-knowledge protocol is used in Protocol 4.1, we see that $\eta = s/2$ and so there are $s/2 + 18$ exponentiations; when it is used in Sect. 3.4 then $\eta = s$ and so there are $s + 18$ exponentiations.

Remark Recall that this zero-knowledge proof is called ℓ times in Protocol 4.1, each time proving an extended Diffie–Hellman tuple of $\eta = s/2$ pairs. Specifically, for every $i = 1, \ldots, \ell$, party P_1 proves that $(g, g^{r_j}, g^{a_i^0}, k'_{i,j})$ is a Diffie–Hellman tuple for all $j \notin \mathcal{J}$. The cost of this is therefore $s\ell + 18\ell$ exponentiations. However, an alternative and equivalent way of carrying out these proofs is for P_1 to prove that for every $j = 1, \ldots, s$ it holds that $(g, g^{a_i^0}, g^{r_j}, k'_{i,j})$ is a Diffie–Hellman tuple for all $1 \leq i \leq \ell$. In this case, each proof costs $2\ell + 18$ exponentiations and so the overall cost is $2s\ell + 18s$ exponentiations. Thus, the way these proofs should be carried out depends on the values of ℓ and s (recall that ℓ is the length of P_2's input and s is the number of circuits sent).

Appendix C. Simple UC Zero-Knowledge from any Σ Protocol

For the sake of completeness, we present the simple general transformation from Σ protocols [8] to universally composable zero-knowledge arguments [4], as appearing in [20]. This transformation is applied to the Σ protocol underlying Protocol B.2 (as described, Protocol B.2 is a stand-alone zero-knowledge proof of knowledge derived by applying a general transformation to a Σ protocol). Loosely speaking, a Σ protocol π for a relation R is a 3-round public-coin proof with the following two properties:

FIGURE C.1 The $\mathcal{F}_{\mathsf{com}}$ Multiple-Commitment Functionality.

$\mathcal{F}_{\mathsf{com}}$ proceeds as follows, running with parties P_1 and P_2, and an adversary $\mathcal{S}$:

1. Upon receiving a value $(\mathsf{Commit}, sid, cid, P_i, P_j, w)$ from P_i $(i \in \{1, 2\})$ where $w \in \{0, 1\}^n$, record the tuple (cid, P_i, P_j, w) and send the message $(\mathsf{Receipt}, sid, cid)$ to P_j and $\mathcal{S}$ (where $j = 3 - i$). Ignore subsequent $(\mathsf{Commit}, sid, cid, P_i, P_j, \ldots)$ values.
2. Upon receiving a value $(\mathsf{Open}, sid, cid, P_i, P_j)$ from P_i, proceed as follows. If the tuple (cid, P_i, P_j, b) is recorded then send the message $(\mathsf{Open}, sid, cid, P_i, P_j, w)$ to P_j and $\mathcal{S}$. Otherwise, do nothing.

1. For any x in the language defined by R, and any pair of accepting conversations (α, β, γ), $(\alpha, \beta', \gamma')$ with the same first prover message α, it is possible to efficiently compute w such that $(x, w) \in R$.
2. There exists a simulator $\mathcal{S}_\pi$ who upon input x and β generates a transcript (α, β, γ) that is indistinguishable from a real proof with a verifier who replies with β upon receiving α.

A universally composable protocol is one that remains secure under arbitrary composition. Essentially, it suffices here to present a straight-line simulator and extractor in order to demonstrate universal composability. We omit a formal definition of Σ protocols and universal composability and refer to [8] and [4], respectively, for details. Our transformation uses universally composable commitment schemes [5] which can be constructed efficiently. Using the scheme of [27] that is secure under the DDH assumption and in the *common reference string model*, the cost of committing to n bits is five Diffie–Hellman group exponentiations and the cost of decommitting is 21 such exponentiations (where the size of the group is at least 2^n). The universally composable (multiple) commitment functionality $\mathcal{F}_{\mathsf{com}}$ is formally defined in Figure C.1.

The Idea Behind the Transformation Let π be a Σ-protocol for a relation R. Then, in order to achieve simulation in the case of a corrupted verifier, we need to know the verifier query β before sending α. Although this may seem impossible it is easily achieved by having the verifier first commit to β using the universally composable commitment functionality $\mathcal{F}_{\mathsf{com}}$. Then, the prover sends α, the verifier decommits to β, and the prover answers with γ. Now, in the simulation, the simulator is able to learn β before the commitment is opened and thus before it sends α. Thus, it is possible to use $\mathcal{S}_\pi$ to simulate the proof after β has been committed to. This takes care of the problem of simulating in the case of a corrupted verifier. However, in the case of a corrupted prover, we need to extract the witness used by the prover without rewinding. This is difficult because the Σ protocol only provides a method for extraction when two different accepting conversations with the same first-prover message are observed. This seems to require rewinding. We solve this problem as follows. Assume for now that β is a single bit. Then, for any α, define γ_0 to be an accepting prover response when $\beta = 0$, and define γ_1 analogously for $\beta = 1$. Then, after receiving the verifier's commitment to β, the prover sends α together with a commitment to γ_0 and a commitment to γ_1. After the verifier reveals β,

the prover reveals γ_β. Now, if the prover indeed sent commitments to two accepting responses γ_0 and γ_1 it follows that the simulator can extract the witness without any rewinding (recall that the simulator can open all commitments). This yields soundness of one half; in order to improve this, we repeat L times and obtain a soundness error of 2^{-L}. Observe that if the original Σ protocol had soundness of $1/2$, then the transformation is extraordinarily efficient. However, if it had negligible soundness—which is often the case—then the transformation costs L times the original protocol. Nevertheless, since the error is 2^{-L}, the value of L can be set to be quite small. We remark that if β comes from a large domain, then we arbitrarily fix two values from the domain to be used in the protocol. Finally, we note that once the prover commits first to its γ values, there is no need to have the verifier commit to β ahead of time. See Protocol C.2 for a formal description.

PROTOCOL C.2 Transformation from Σ-Protocols to UC-ZK.

Let π be a 3-round Σ protocol for a relation R. Let β^0, β^1 be two arbitrary distinct values in the domain of verifier queries. Denote by Π the protocol as follows:

- **Inputs:** The prover P and verifier V both hold a common statement x; the prover P has a witness w such that $(x, w) \in R$.
- **Session identifier:** Both parties have the same *sid*
- **The protocol:**

1. *First prover message:*
   (a) P runs the prover instructions L times for the Σ protocol π in order to obtain L first prover message $\alpha_1, \ldots, \alpha_L$. The prover P commits to all of these values by sending $(\mathsf{Commit}, sid, i, P, V, \alpha_i)$ to $\mathcal{F}_{\mathsf{com}}$, for every i.
   (b) For every $i = 1, \ldots, L$, the prover P computes the second prover message in the case that the verifier query after α_i is β_0 and in the case that it is β_1; we denote these messages by γ_i^0 and γ_i^1, respectively.
   (c) P commits to all of the γ_i^0, γ_i^1 values by sending $(\mathsf{Commit}, sid, i\|0, P, V, \gamma_i^0)$ and $(\mathsf{Commit}, sid, i\|1, P, V, \gamma_i^1)$ to $\mathcal{F}_{\mathsf{com}}$, for every $i = 1, \ldots, L$.
2. *Verifier message:* Upon receiving $(\mathsf{Receipt}, sid, i)$, $(\mathsf{Receipt}, sid, i\|0)$ and $(\mathsf{Receipt}, sid, i\|1)$ from $\mathcal{F}_{\mathsf{com}}$ for every $i = 1, \ldots, L$, the verifier V chooses L independent random values $\beta_1, \ldots, \beta_L \in \{\beta^0, \beta^1\}$ and sends them to P.
3. *Second prover message:* Upon receiving $\beta_1, \ldots, \beta_L$ from V, the prover P decommits to all α_i values, as well as the appropriate $\gamma_i^{\beta_i}$ values, for every i. Formally, P sends $(\mathsf{Open}, sid, i, P, V)$ to $\mathcal{F}_{\mathsf{com}}$ for every i. In addition, let σ_i be such that $\beta_i = \beta^{\sigma_i}$. Then P sends $(\mathsf{Open}, sid, i\|\sigma_i, P, V)$ to $\mathcal{F}_{\mathsf{com}}$ for every $i = 1, \ldots, L$.
4. *Accept/reject decision:* Upon receiving $(\mathsf{Open}, sid, i, P, V, \alpha_i)$ and $(\mathsf{Open}, sid, i\|\sigma_i, P, V, \gamma_i)$ for every $i = 1, \ldots, L$, the verifier V checks that for every i the value σ_i is such that $\beta_i = \beta^{\sigma_i}$ and that $(\alpha_i, \beta_i, \gamma_i)$ is an accepting transcript according to the Σ protocol π. V accepts if and only if this holds for every i.

We have the following theorem.

Theorem C.3. *If π is a Σ protocol for a relation R, then the protocol Π obtained by applying the transformation of Protocol C.2 to π is a universally composable zero-knowledge proof of knowledge for R with soundness error 2^{-L}.*

Proof Sketch. In the case that the adversary $\mathcal{A}$ controls the prover P, the simulator $\mathcal{S}$ receives $(\mathsf{Commit}, sid, i, \alpha_i)$, $(\mathsf{Commit}, sid, i\|0, \gamma_i^0)$, $(\mathsf{Commit}, sid, i\|1, \gamma_i^1)$ from $\mathcal{A}$. Then, $\mathcal{S}$ chooses random $\beta_1, \ldots, \beta_L$, hands them to $\mathcal{A}$ as if coming from V, and verifies that for every i the transcript $(\alpha_i, \beta_i, \gamma_i^{\beta_i})$ is accepting; if not, it sends an invalid witness w' to $\mathcal{F}_{\mathsf{zk}}$ (i.e., w' such that $(x, w') \notin R$). Otherwise, if for every i, at most one of $(\alpha_i, 0, \gamma_i^0)$ and $(\alpha_i, 1, \gamma_i^1)$ is an accepting transcript in the Σ protocol, then $\mathcal{S}$ outputs fail and halts. Otherwise, it uses the extractor of the Σ protocol to obtain a witness w such that $(x, w) \in R$ and sends this to $\mathcal{F}_{\mathsf{zk}}$. In addition, $\mathcal{S}$ sends all messages that it receives from the environment $\mathcal{Z}$ to $\mathcal{A}$, and vice versa. The fact that this simulation is indistinguishable follows from the fact that the only difference between the real and ideal executions is when $\mathcal{S}$ outputs fail, which occurs in the case that all $(\alpha_i, \beta_i, \gamma_i^{\beta_i})$ are accepting (and so V would accept), and yet at most one of all $(\alpha_i, 0, \gamma_i^0)$ and $(\alpha_i, 1, \gamma_i^1)$ is accepting (and so $\mathcal{S}$ does not obtain a valid witness). However, this event occurs with probability at most 2^{-L}, which is negligible.

In the case that the adversary $\mathcal{A}$ controls the verifier V, the simulator $\mathcal{S}$ works by handing the appropriate $\mathsf{Receipt}$ messages to $\mathcal{A}$. Then, $\mathcal{S}$ receives from $\mathcal{A}$ values $\beta_1, \ldots, \beta_L$. $\mathcal{S}$ then hands the Σ protocol simulator all of the β_i messages and obtains accepting transcripts $(\alpha_i, \beta_i, \gamma_i)$. Finally, $\mathcal{S}$ hands $\mathcal{A}$ all of the commitment openings to be to the α_i, γ_i values generated by the simulator. As above, it also sends all messages that it receives from the environment $\mathcal{Z}$ to $\mathcal{A}$, and vice versa. The indistinguishability here follows immediately from the indistinguishability of the Σ protocol simulated transcripts. □

Exact Efficiency of the Transformation The complexity of the universally composable zero-knowledge protocol is the cost of the initial Σ protocol plus 3L universally composable commitments, 2L of which are opened. As we have mentioned, the commitment scheme of [27] is such that a commitment to n bits requires five exponentiations and a decommitment requires 21. Thus, the cost of Protocol C.2 is L times the initial Σ protocol plus $(3 \times 5 + 2 \times 21) \cdot L = 57L$ regular Diffie–Hellman group exponentiations.

References

[1] Y. Aumann, Y. Lindell, Security against covert adversaries: Efficient protocols for realistic adversaries. *J. Cryptol.* **23**(2), 281–343 (2010)

[2] D. Beaver, Foundations of secure interactive computing, in *CRYPTO'91*. LNCS, vol. 576 (Springer, Berlin, 1991), pp. 377–391

[3] R. Canetti, Security and composition of multi-party cryptographic protocols. *J. Cryptol.* **13**(1), 143–202 (2000)

[4] R. Canetti, Universally composable security: A new paradigm for cryptographic protocols, in 42*nd FOCS* (2001), pp. 136–145. Full version available at http://eprint.iacr.org/2000/067

[5] R. Canetti, M. Fischlin, Universally composable commitments, in *CRYPTO 2001*. LNCS, vol. 2139 (Springer, Berlin, 2001), pp. 19–40

[6] L. Carter, M.N. Wegman, Universal classes of Hash functions. *J. Comput. Syst. Sci.* **18**(2), 143–154 (1979)
[7] R. Cramer, I. Damgård, B. Schoenmakers, Proofs of partial knowledge and simplified design of Witness hiding protocols, in *CRYPTO'94*. LNCS, vol. 839 (Springer, Berlin, 1994), pp. 174–187
[8] I. Damgård, On Σ protocols. http://www.daimi.au.dk/~ivan/Sigma.pdf
[9] Y. Dodis, R. Gennaro, J. Hastad, H. Krawczyk, T. Rabin, Randomness extraction and key derivation using the CBC, cascade and HMAC modes, in *CRYPTO 2004*. LNCS, vol. 3152 (Springer, Berlin, 2004), pp. 494–510
[10] Y. Dodis, V. Shoup, S. Walfish, Efficient constructions of composable commitments and zero-knowledge proofs, in *CRYPTO 2008*. LNCS, vol. 5157 (Springer, Berlin, 2008), pp. 515–535
[11] S. Even, O. Goldreich, A. Lempel, A randomized protocol for signing contracts. *Commun. ACM* **28**(6), 637–647 (1985)
[12] J. Garay, P. MacKenzie, K. Yang. Strengthening zero-knowledge protocols using signatures, in *EUROCRYPT 2003*. LNCS, vol. 2656 (Springer, Berlin, 2003), pp. 177–194
[13] O. Goldreich, *Foundations of Cryptography: Volume 2—Basic Applications* (Cambridge University Press, Cambridge, 2004)
[14] O. Goldreich, S. Micali, A. Wigderson, How to play any mental game—A completeness theorem for protocols with honest majority, in *19th STOC* (1987), pp. 218–229. For details see [13, Chap. 7]
[15] S. Goldwasser, L. Levin, Fair computation of general functions in presence of immoral majority, in *CRYPTO'90*. LNCS, vol. 537 (Spring, Berlin, 1990), pp. 77–93
[16] V. Goyal, P. Mohassel, A. Smith. Efficient two party and multi party computation against covert adversaries, in *EUROCRYPT 2008*. LNCS, vol. 4965 (Springer, Berlin, 2008), pp. 289–306
[17] R.L. Graham, D.E. Knuth, O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, 2nd edn. (Addison-Wesley, Reading, 1998)
[18] J. Hastad, R. Impagliazzo, L. Levin, M. Luby, Construction of a pseudo-random generator from any one-way function. *SIAM J. Comput.* **28**(4), 1364–1396 (1999)
[19] C. Hazay, Y. Lindell, *Efficient Secure Two-Party Protocols: Techniques and Constructions* (Springer, Berlin, 2010)
[20] C. Hazay, K. Nissim, Efficient set operations in the presence of malicious adversaries, in *PKC 2010*. LNCS, vol. 6056 (Springer, Berlin, 2010), pp. 312–331
[21] Y. Ishai, M. Prabhakaran, A. Sahai, Founding cryptography on oblivious transfer—Efficiently, in *CRYPTO 2008*. LNCS, vol. 5157 (Springer, Berlin, 2008), pp. 572–591
[22] Y. Ishai, M. Prabhakaran, A. Sahai, Secure arithmetic computation with no honest majority, in *TCC 2009*. LNCS, vol. 5444 (Springer, Berlin, 2009), pp. 294–314
[23] S. Jarecki, V. Shmatikov, Efficient two-party secure computation on committed inputs, in *EUROCRYPT 2007*. LNCS, vol. 4515 (Springer, Berlin, 2007), pp. 97–114
[24] M. Kiraz, B. Schoenmakers, A protocol issue for the malicious case of Yao's garbled circuit construction, in *Proceedings of the 27th Symposium on Information Theory in the Benelux* (2006), pp. 283–290
[25] V. Kolesnikov, T. Schneider, Improved garbled circuit: Free XOR gates and applications, in 35*th ICALP*. LNCS, vol. 5126 (Springer, Berlin, 2008), pp. 486–498
[26] Y. Lindell, Parallel coin-tossing and constant-round secure two-party computation. *J. Cryptol.* **16**(3), 143–184 (2003)
[27] Y. Lindell, Highly-efficient universally-composable commitments, in *EUROCRYPT 2011*. LNCS, vol. 6632 (Springer, Berlin, 2011), pp. 446–466
[28] Y. Lindell, B. Pinkas, An efficient protocol for secure two-party computation in the presence of malicious adversaries, in *EUROCRYPT 2007*. LNCS, vol. 4515 (Springer, Berlin, 2007), pp. 52–78
[29] Y. Lindell, B. Pinkas, A proof of Yao's protocol for secure two-party computation. *J. Cryptol.* **22**(2), 161–188 (2009)
[30] P. MacKenzie, K. Yang, On simulation-sound trapdoor commitments, in *EUROCRYPT 2004*. LNCS, vol. 3027 (Springer, Berlin, 2004), pp. 382–400
[31] A.J. Menezes, P.C. van Oorschot, S.A. Vanstone, *Handbook of Applied Cryptography* (CRC Press, Boca Raton, 2001)
[32] S. Micali, P. Rogaway, Secure computation. Unpublished manuscript, 1992. Preliminary version in *CRYPTO'91*, LNCS, vol. 576 (Springer, Berlin, 1991), pp. 392–404
[33] P. Mohassel, M.K. Franklin, Efficiency tradeoffs for malicious two-party computation, in *9th PKC Conference*. LNCS, vol. 3958 (Springer, Berlin, 2006), pp. 458–473

[34] M. Naor, O. Reingold, Synthesizers and their application to the parallel construction of pseudo-random functions, in *36th FOCS* (1995), pp. 170–181
[35] J.B. Nielsen, C. Orlandi, LEGO for two-party secure computation, in *TCC 2009*. LNCS, vol. 5444 (Springer, Berlin, 2009), pp. 368–386
[36] C. Orlandi, Personal communication, 2010
[37] C. Peikert, V. Vaikuntanathan, B. Waters, A framework for efficient and composable oblivious transfer, in *CRYPTO'08*. LNCS, vol. 5157 (Springer, Berlin, 2008), pp. 554–571
[38] B. Pinkas, T. Schneider, N.P. Smart, S.C. Williams. Secure two-party computation is practical, in *ASIACRYPT 2009*. LNCS, vol. 5912 (Springer, Berlin, 2009), pp. 250–267
[39] M. Rabin, How to exchange secrets by oblivious transfer. Tech. Memo TR-81, Aiken Computation Laboratory, Harvard U., 1981
[40] A. Shamir, How to share a secret. *Commun. ACM* **22**(11), 612–613 (1979)
[41] A.C. Yao, How to generate and exchange secrets, in *27th FOCS* (1986), pp. 162–167