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

Yehuda Lindell* and Benny Pinkas**

Dept. of Computer Science, Bar-Ilan University, Israel.

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.

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 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 [13, 14, 26, 2, 3]. An important parameter when considering this

* lindell@cs.biu.ac.il.Research supported by the European Research Council as part of the ERC project LAST, and by the ISRAEL SCIENCE FOUNDATION (781/07).

** benny@pinkas.net Research supported by the European Research Council as part of the ERC project SFEROT and by the EU project CACE.

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 functionality can be securely computed in the presence of semi-honest adversaries [33] and in the presence of malicious adversaries [13]. 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 [33] 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 [20, 24, 28, 18]. 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 Section 1.3.

We remark that the protocol of [24] 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" [31]. A Boolean circuit for computing this function was designed with approximately 33,000 gates, and the protocol of [24] 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 [24] and significantly simplify and improve the efficiency of their construction. The approach of [24] is to carry out a basic cut-and-choose on the garbled circuit construction of Yao [33] (we assume familiarity with Yao's protocol). That is, party P_1 constructs s copies of a garbled circuit and sends 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 a statistical security parameter s. 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 [24], 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 [24]. 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 [24] required constructing and sending $s^2\ell$ commitments. In the implementation by [31], they set $s = 160$ and $\ell = 128$. Thus, the overhead due to these consistency proofs is the computation and transmission of $3,276,800$ commitments! Another problem that arises in the protocol of [24] 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. The solution to this problem in [24] 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 [24] 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 [31] 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 [24]. In addition, we reduce the error probability to $2^{-0.311s}$ and thus for an error of 2^{-40} it suffices to send only 128 circuits. This is an important improvement because the experiments of [31] 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 [24] 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 [27]. 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 is using the same input in all circuits when the keys have this nice structure.
2. As we have described, the reason that the inputs and circuits were needed to be made larger in [24] 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 with the sender inputting many pairs $(x_1^0, x_1^1), \ldots, (x_s^0, x_s^1)$, and the receiver inputting 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 $j \in \mathcal{J}$. 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 Section 1.3 for a summary of the exact complexity of our protocol, and Section 3 for a complete analysis, with optimizations in Section 3.3.

Variants. Another advantage of our protocol over that of [24] 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. Finally, we also 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 [20]):** The secure two-party protocol of [20] 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 ($23,760,000$ for AES), 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. Furthermore, for a large circuit we believe it unlikely that this method will yield a highly efficient protocol.)
- **LEGO (Nielsen-Orlandi [28]):** The LEGO protocol [28] 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 [29]. Thus, for an error of 2^{-40} we have 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 have 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 [20], as predicted in [28]. 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 [20] who work in $\mathbb{Z}_N^*$.)
- **Virtual multiparty method (Ishai et al. [18, 19]):** 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 the multiplication 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 [24]):** 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 $s^2\ell$ commitments. In addition, the proven error of the protocol is $2^{-s/17}$ and its conjectured error is $2^{-s/4}$. The actual 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 $\mathbf{5.66}s\ell$ **full exponentiations**, and if preprocessing can be used then only $s\ell/2$ full exponentiations need to be computed after the inputs become known. In addition, the protocol requires the exchange of $\mathbf{7}s\ell + \mathbf{22}\ell + \mathbf{7}s + \mathbf{5}$ **group elements**, and has **12 rounds of communication**. Finally, there are $\mathbf{6.5}|C|s$ **symmetric encryptions** for constructing and decrypting the garbled circuits and $\mathbf{4}|C|s$ ciphertexts sent for transmitting these circuits. An important factor here is the value of s needed. The error of our protocol is $\mathbf{2^{-0.311}}s$ and so for an error of 2^{-40} it suffices to set $s = \mathbf{128}$. (The overhead of computing an AES circuit, after preprocessing, with $|C| = 33,000$, and $s = \ell = 128$, is therefore about $93,000$ exponentiations, $27,500,000$ symmetric encryptions, and communicating 28.6 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.

Full version. In this extended abstract we do not have space to present the proofs of security of our protocols. A full version of this paper appears in the *Cryptology ePrint Archive* (report 2010/284).

2 Cut-and-Choose Oblivious Transfer

2.1 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 $\mathcal{F}_{\mathbf{ccot}}$, with parameter s, is formally defined in Figure 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 $\mathcal{F}^{\boldsymbol{S}}_{\mathbf{ccot}}$.

FIGURE 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 $\mathcal{F}^{\boldsymbol{S}}_{\mathbf{ccot}}$:

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

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 with its input while keeping its input secret. It is crucial that P_2 obtain only a single key for each wire, since this is what ensures that it can only obtain a single output. 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, enabling it to decrypt all garbled gates and check that they were correctly constructed. 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 Section 3.1 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 (equivalently, every wire on the circuit), and P_2 should receive the keys associated with this fixed bit in all of the circuits. In order to ensure that P_2 uses the same input in all circuits, we devised a *single-choice* variant of cut-and-choose OT. In the full version, we separately present the basic variant since it is of independent interest and may be useful in other applications.

2.2 Constructing a Single-Choice Cut-and-Choose OT Protocol

The starting point for our construction of cut-and-choose OT is the universally composable protocol of Peikert et al. [30]; 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 [30] 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 $(g_0, g_1, g_0, \frac{h_1}{g_1})$ is a Diffie-Hellman tuple (note that the security of [30] 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 [30] in the stand-alone model.

Next, observe that the protocol of [30] 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 [30] to prove universal composability; in their case the simulator can choose the CRS so that $a = b$ and then learn 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 pairs $\{(h_0^j, h_1^j)\}_{j=1}^s$ such that for $s/2$ of the pairs (h_0^j, h_1^j) it holds that $a \neq b$ (ensuring that it learns only one input) and for $s/2$ of the pairs (h_0^j, h_1^j) it holds that $a = b$ (enabling it to learn both inputs by actually running the UC simulator). 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 $a \neq b$. This proof too can be computed at low cost using the technique of Cramer et al. [7]; see the full version of the paper for a full description and efficiency analysis of the zero-knowledge protocol.

In the oblivious transfer protocol, the receiver with an input σ chooses a random r and sends $(g_\sigma)^r$ and $(h_\sigma^1)^r, \ldots, (h_\sigma^s)^r$ to the sender, using the $g_0, g_1, (h_0^j, h_1^j)$ values sent previously. In order to ensure that the single-choice property holds, the receiver R must prove that it used the same σ in every computation of $(h_\sigma^j)^r$. The protocol has been carefully designed so that the way that R chooses the values enables this proof to be carried out efficiently. Specifically, 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 $1 \leq j \leq s$, or $g' = (g_1)^r$ and $h_j = (h_1^j)^r$ for every $1 \leq j \leq s$, which is just a proof that one of two sets of tuples are all of the Diffie-Hellman type; see the protocol specification below and the full version for details. The cost of this proof is $s + 18$ exponentiations and the exchange of 10 group elements.

PROTOCOL 2 (Single-Choice 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 a single bit $\sigma \in \{0,1\}$ 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 (and through this, that the tuples (g_0, g_1, h_0^j, h_1^j) are* not *DH tuples). If S rejects the proof then it outputs $\perp$ and halts.*
- **Transfer phase (for every j):**
  1. *The receiver chooses a 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.*
  2. *The receiver proves in zero knowledge that either all $\{(g_0, h_0^j, g', h_j)\}_{j=1}^s$ or all $\{(g_1, h_1^j, g', h_j)\}_{j=1}^s$ are Diffie-Hellman tuples.*
  3. *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 the pairs $(u_0^j, v_0^j) = RAND(g_0, g_j, h_0^j, h_j)$ and $(u_1^j, v_1^j) = RAND(g_1, g_j, h_1^j, h_j)$.*
     - *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 $w_1^j = v_1^j \cdot x_1^j$, for every j.*
- **Output:**
  1. *For every $j \in \{1, \ldots, s\}$, the receiver computes $x_\sigma^j = w_\sigma^j / (u_\sigma^j)^r$.*
  2. *For every $j \in \mathcal{J}$, the receiver also computes $x_{1-\sigma}^j = w_{1-\sigma}^j / (u_{1-\sigma}^j)^{r \cdot z}$ where $z = y^{-1} \bmod q$ if $\sigma = 0$, and $z = y$ if $\sigma = 1$.*

The fact that the output obtained is correct follows from the correctness (and simulation strategy) of the protocol of [30]. We prove the following:

Proposition 3 *If the Decisional Diffie-Hellman assumption holds in the group $\mathbb{G}$, then Protocol 2 securely realizes the $\mathcal{F}_{\text{ccot}}^S$ functionality in the presence of malicious adversaries.*

The proof of Proposition 3 is based on the following ideas. Suppose first that the adversary controls the receiver; we briefly describe an ideal-model simulator "interacting" with the adversary. The simulator receives from the adversary

the values (g_0, g_1) and $(h_0^1, h_1^1, \ldots, h_s^0, h_s^1)$ of the setup phase, and then runs the extractor of the zero-knowledge proof of knowledge to learn the witness set. The extracted witnesses identify exactly the set $\mathcal{J}$ of $s/2$ indices for which (g_0, g_1, h_0^j, h_1^j) is a Diffie-Hellman tuple, and in addition provides the simulator with the set of values α_j for all $j \notin \mathcal{J}$. Next, in the transfer phase, given g' and h_j, the simulator can extract the value of σ by simply checking if $(g')^{\alpha_j} = h_j$; alternatively, it can run the extractor for the zero-knowledge proof of the transfer phase. The simulator then sends the set $\mathcal{J}$ and bit σ to the trusted party and obtains the corresponding outputs. Now, for every $j \in \mathcal{J}$, the simulator received both values x_0^j and x_1^j and so can compute (u_0^j, w_0^j) and (u_1^j, w_1^j) just like the honest sender. In contrast, for every $j \notin \mathcal{J}$, the simulator received only x_σ^j. In this case, it can still compute (u_σ^j, w_σ^j) like the honest sender. Regarding $(u_{1-\sigma}^j, w_{1-\sigma}^j)$, these can just be taken as uniformly distributed values in $\mathbb{G}$, by the property of the $RAND$ transformation when applied to non-Diffie-Hellman tuples.

Next consider the case that the adversary controls the sender. In this case, the simulator chooses the (h_0^j, h_1^j) values such that (g_0, g_1, h_0^j, h_1^j) is a Diffie-Hellman tuple for *all* j. The simulator then "cheats" in the zero-knowledge proof by generating a *simulated* zero-knowledge proof with the adversary. Given that now all (g_0, g_1, h_0^j, h_1^j) are Diffie-Hellman tuples, the simulator can extract both inputs (x_0^j, x_1^j) for *every* $j = 1, \ldots, s$, using the same computation as an honest receiver would for $j \in \mathcal{J}$. Finally, the simulator sends the values $\{(x_0^j, x_1^j)\}_{j=1}^s$ to the trusted party, outputs whatever the adversary outputs, and halts. The fact that this is indistinguishable from a real execution follows directly from the indistinguishability of simulated zero-knowledge proofs from real proofs, and from the DDH assumption.

Exact efficiency. In the full version, we present an exact analysis of this protocol, including the cost of all zero-knowledge proofs. The result is that the protocol requires **$20.5s + 24$ exponentiations**, the exchange of **$11s + 15$ group elements**, and **6 rounds of communication**.

2.3 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 (where for each wire the input used is P_2's input bit). However, the subset of indices for which the receiver obtains both pairs must be the same in all transfers. This is due to the fact that this determines which of the circuits are checked and which are evaluated, and it is crucial that in all of the evaluated circuits P_2 only receives one value on every wire. We call a functionality that achieves this "batch single-choice cut-and-choose OT" and denote it $\mathcal{F}_{\text{ccot}}^{S,B}$.

In order to realize this functionality it suffices to run the setup phase of Protocol 2 once (this ensures that the set $\mathcal{J}$ is the same in all executions). Then, the transfer phase of the single-choice protocol is run ℓ times *in parallel* (with the single choice in the ith execution being some σ_i). We remark that parallel composition holds here because the simulation only rewinds in the transfer

phase for the zero-knowledge protocol, which is zero-knowledge under parallel composition. We have the following:

Proposition 4 *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 proof it 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 $6s + 10$ group elements. We conclude that there are $\mathbf{11.5s\ell + 19\ell + 9s + 5}$ **exponentiations**, $\mathbf{6s\ell + 10\ell + 5s + 5}$ **group elements** sent and **6 rounds of communication**. In Section 3.3 we observe that $10.5s\ell$ of the exponentiations are "fixed-base" and thus the overall effective cost is about $\mathbf{4.5s\ell}$ **exponentiations**.

3 The Protocol for Secure Two-Party Computation

3.1 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.[1] (The keys are actually a pseudorandom synthesizer [27], 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).

[1] 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, 16]. 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 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. Observe that 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 was indeed its choice by sending, for each check circuit, *both* values on the wire associated with P_2's first input bit. 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, then P_1 sends r_j to P_2. Given all of the $g^{a_i^0}, g^{a_i^1}$ values and r_j, party 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 assured that a majority of the evaluation circuits are correct.

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 Section 1.2).

We remark on one type of attack discussed in [21, 24]. 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 [24] 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. We now proceed to the full protocol description.

PROTOCOL 5 (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: $k_{i,j}^0 = H(g^{a_i^0 \cdot r_j})$ and $k_{i,j}^1 = H(g^{a_i^1 \cdot r_j})$, where H is a suitable randomness extractor [6, 16]; see also [10].*
   (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.*
2. Oblivious transfers: *P_1 and P_2 run batch single-choice cut-and-choose oblivious transfer with parameters ℓ (the number of parallel executions) and s (the number of pairs in each execution):*
   (a) *P_1 defines vectors $\boldsymbol{z}_1, \ldots \boldsymbol{z}_\ell$ so that $\boldsymbol{z}_i$ 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: $\left\{(i, 0, g^{a_i^0}), (i, 1, g^{a_i^1})\right\}_{i=1}^{\ell}$ and $\left\{(j, g^{r_j})\right\}_{j=1}^{s}$.*

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.*
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 proves* in parallel *that there exists a value $\sigma_i \in \{0, 1\}$ such that for every $j \notin J$, $k'_{i,j} = g^{a_i^{\sigma_i} \cdot r_j}$. 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 7a and the keys associated with its own input obtained in Step 2c 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.*

3.2 Properties

The security of the protocol is expressed in the following theorem, which is proved in the full version of the paper:

Theorem 6 *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 5 securely computes the function f in the presence of malicious adversaries.*

We remark here on one aspect of the proof that is crucial to the concrete efficiency of the protocol. Party P_1 can successfully cheat if it manages to pass the cut-and-choose test with a majority of the evaluation circuits being incorrect. To do this, at least $s/4$ circuits must be incorrect. In the proof of security

we show that the probability that at least this many circuits are incorrect without P_2 catching P_1 is approximately $2^{-0.311s}$ where the approximation is due to Stirling's formula. Based on this, it suffices to use 128 garbled circuits in order to obtain an error of 2^{-40}. (We also compared the exact bound to this approximation on the concrete value of $s = 128$, to verify that the approximation is good for s of this size.)

Exact efficiency. An exact analysis of the protocol yields that there are $\mathbf{15s\ell + 39\ell + 10s + 5}$ **exponentiations** (of which $11.5s\ell$ are for Step 2, performing the OTs), $\mathbf{7s\ell + 22\ell + 7s + 5}$ **group elements** sent and **12 rounds of communication**. In addition, there are $\mathbf{6.5|C|s}$ **symmetric encryptions**, of which $4|C|s$ encryptions for constructing all s garbled circuits, and $2|C|s$ encryptions for P_2 to check $s/2$ of them. Finally, there are $\mathbf{4|C|s}$ **ciphertexts** sent for transmitting these circuits. The overhead of the protocol can be improved by different optimizations, as shown in Section 3.3 below.

3.3 Optimizations

Fixed-base exponentiations. Exponentiations are commonly computed by repeated squaring, which for a group of order q of length m bits requires on average $1.5m$ 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 $0.5m$ multiplications. All but $s\ell$ of the exponentiations in our protocol are fixed-based, and thus taking this into account the effective overhead of the exponentiations is equivalent to that of just $\mathbf{5.66s\ell}$ **full exponentiations.**

Reducing the computation of P_2 in Step 6. In Step 6 of Protocol 5, 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). 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).

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

Claim 7 *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 $\frac{s}{2} \cdot 2^{-L}$.*

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 to 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 $\boldsymbol{s\ell/2}$ **full exponentiations**.

4 Universal Composability and Covert Adversaries

4.1 Universally Composable Two-Party Computation

The simulators in the proof of Theorem 6 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 5. In order to obtain this property, we simply need to apply a transformation from Sigma protocols to universally-composable zero knowledge, which can be achieved efficiently using universally-composable commitments. Details of how this can be achieved are given in the full version of the paper. We have the following:

Theorem 8 *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 commitment-hybrid model in the presence of malicious adversaries, with 8 rounds of computation and $O(s\ell + s^2)$ exponentiations.*

4.2 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 of successfully cheating (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 $2^{-\frac{s}{4}+1}$. This immediately yields a protocol that is secure in the presence of covert adversaries, as stated in the following theorem.

Theorem 9 *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 5 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 [15] when values of ϵ that are greater than $1/2$ are desired. For example, in order to obtain an ϵ-deterrent of 0.98, the protocol of [1] requires using 50 garbled circuits. However, taking $s = 50$ in our protocol here yields an ϵ-deterrent of $1 - 2^{-11.5}$ which is much much larger.

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.

References

1. Y. Aumann and Y. Lindell. Security Against Covert Adversaries: Efficient Protocols for Realistic Adversaries. *Journal of Cryptology*, 23(2):281–343, 2010.
2. D. Beaver. Foundations of Secure Interactive Computing. *CRYPTO'91*, Springer (LNCS 576), pages 377–391, 1991.
3. R. Canetti. Security and Composition of Multi-party Cryptographic Protocols. *Journal of Cryptology*, 13(1):143–202, 2000.
4. R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. 42*nd FOCS*, pages 136–145, 2001. Full version available at `http://eprint.iacr.org/2000/067`.
5. R. Canetti and M. Fischlin. Universally Composable Commitments. *CRYPTO 2001*, Springer (LNCS 2139), pages 19–40, 2001.
6. L. Carter and M.N. Wegman. Universal Classes of Hash Functions. *JCSS*, 18(2):143–154, 1979.
7. R. Cramer, I. Damgård and B. Schoenmakers. Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols. *CRYPTO'94*, Springer (LNCS 839), pages 174-187, 1994.
8. I. Damgård. On Σ Protocols. `http://www.daimi.au.dk/~ivan/Sigma.pdf`.
9. Y. Dodis, V. Shoup and S. Walfish. Efficient Constructions of Composable Commitments and Zero-Knowledge Proofs. *CRYPTO 2008*, Springer (LNCS 5157), pages 515–535, 2008.
10. Y. Dodis, R. Gennaro, J. Hastad, H. Krawczyk and T. Rabin. Randomness Extraction and Key Derivation Using the CBC, Cascade and HMAC Modes. *CRYPTO 2004*, Springer (LNCS 3152), pages 494–510, 2004.
11. J. Garay, P. MacKenzie and K. Yang. Strengthening Zero-Knowledge Protocols Using Signatures. *EUROCRYPT 2003*, Springer (LNCS 2656), 177–194, 2003.

12. O. Goldreich. *Foundations of Cryptography: Volume 2 – Basic Applications*. Cambridge University Press, 2004.
13. O. Goldreich, S. Micali and A. Wigderson. How to Play any Mental Game – A Completeness Theorem for Protocols with Honest Majority. *19th STOC,* pages 218–229, 1987. For details see [12, Chapter 7].
14. S. Goldwasser and L. Levin. Fair Computation of General Functions in Presence of Immoral Majority. *CRYPTO'90*, Spring-Verlag (LNCS 537), 77–93, 1990.
15. V. Goyal, P. Mohassel and A. Smith. Efficient Two Party and Multi Party Computation Against Covert Adversaries. *EUROCRYPT 2008*, Springer (LNCS 4965), pages 289–306, 2008.
16. J. Hastad, R. Impagliazzo, L. Levin and M. Luby. Construction of a Pseudo-random Generator from any One-way Function. *SIAM Journal on Computing*, 28(4);1364–1396, 1999.
17. C. Hazay and K. Nissim. Efficient Set Operations in the Presence of Malicious Adversaries. *PKC 2010*, Springer (LNCS 6056), pages 312-331, 2010.
18. Y. Ishai, M. Prabhakaran and A. Sahai. Founding Cryptography on Oblivious Transfer – Efficiently. *CRYPTO 2008*, Springer (LNCS 5157), 572–591, 2008.
19. Y. Ishai, M. Prabhakaran and A. Sahai. Secure Arithmetic Computation with No Honest Majority. *TCC 2009*, Springer (LNCS 5444), pages 294–314, 2009.
20. S. Jarecki and V. Shmatikov. Efficient Two-Party Secure Computation on Committed Inputs. *EUROCRYPT 2007*, Springer (LNCS 4515), pages 97–114, 2007.
21. M. Kiraz and B. Schoenmakers. A Protocol Issue for the Malicious Case of Yao's Garbled Circuit Construction. *Proceedings of 27th Symposium on Information Theory in the Benelux,* pages 283–290, 2006.
22. Y. Lindell. Parallel Coin-Tossing and Constant-Round Secure Two-Party Computation. *Journal of Cryptology*, 16(3):143–184, 2003.
23. Y. Lindell and B. Pinkas. A Proof of Yao's Protocol for Secure Two-Party Computation. the *Journal of Cryptology*, 22(2):161–188, 2009.
24. Y. Lindell and B. Pinkas. An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries. *EUROCRYPT 2007*, Springer (LNCS 4515), pages 52-78, 2007.
25. P. MacKenzie and K. Yang. On Simulation-Sound Trapdoor Commitments. *EUROCRYPT 2004*, Springer (LNCS 3027), pages 382-400, 2004.
26. S. Micali and P. Rogaway. Secure Computation. Unpublished manuscript, 1992. Preliminary version in *CRYPTO'91*, Springer (LNCS 576), pages 392–404, 1991.
27. M. Naor and O. Reingold. Synthesizers and Their Application to the Parallel Construction of Psuedo-Random Functions. 36*th FOCS*, pages 170–181, 1995.
28. J.B. Nielsen and C. Orlandi. LEGO for Two-Party Secure Computation. *TCC 2009*, Springer (LNCS 5444), pages 368–386, 2009.
29. C. Orlandi. Personal communication, 2010.
30. C. Peikert, V. Vaikuntanathan and B. Waters. A Framework for Efficient and Composable Oblivious Transfer. *CRYPTO'08*, Springer (LNCS 5157), pages 554–571, 2008.
31. B. Pinkas, T. Schneider, N.P. Smart and S.C. Williams. Secure Two-Party Computation Is Practical. *ASIACRYPT 2009*, Springer (LNCS 5912), 250–267, 2009.
32. A. Shamir. How to Share a Secret. *Communications of the ACM*, 22(11):612–613, 1979.
33. A.C. Yao. How to Generate and Exchange Secrets. *27th FOCS*, 162–167, 1986.