Amortizing Garbled Circuits*

Yan Huang† Jonathan Katz‡ Vladimir Kolesnikov§ Ranjit Kumaresan¶
Alex J. Malozemoff‡

Abstract

We consider secure two-party computation in a *multiple-execution* setting, where two parties wish to securely evaluate the same circuit multiple times. We design efficient garbled-circuit-based two-party protocols secure against *malicious* adversaries. Recent works by Lindell (Crypto 2013) and Huang-Katz-Evans (Crypto 2013) have obtained optimal complexity for cut-and-choose performed over garbled circuits in the single execution setting. We show that it is possible to obtain much lower *amortized* overhead for cut-and-choose in the multiple-execution setting.

Our efficiency improvements result from a novel way to combine a recent technique of Lindell (Crypto 2013) with LEGO-based cut-and-choose techniques (TCC 2009, Eurocrypt 2013). In concrete terms, for 40-bit statistical security we obtain a 2× improvement (per execution) in communication and computation for as few as 7 executions, and require only 8 garbled circuits (i.e., a 5× improvement) per execution for as low as 3500 executions. Our results suggest the exciting possibility that secure two-party computation in the malicious setting can be less than an order of magnitude more expensive than in the semi-honest setting.

1 Introduction

Two-party secure computation (2PC) is a rapidly developing area of cryptography. While the basic approach for semi-honest security, *garbled circuits* (GC) [Yao86], is extensively studied and is largely settled, security against malicious players has seen recent significant improvements. The classical technique for lifting the GC approach to work in the malicious setting is *cut-and-choose* (C&C), formalized and proven secure by Lindell and Pinkas [LP07]. Until recently, this approach required significant overhead: to guarantee probability of cheating $< 2^{-s}$, approximately $3s$ garbled circuits needed to be generated and sent. However, in Crypto 2013 two works reduced the number of garbled circuits required in cut-and-choose to $s + O(\log s)$ [HKE13] and to s [Lin13].

Our contribution. We further significantly reduce the replication factor for C&C-based protocols in the *multiple execution* setting, where the same function (possibly with different inputs) is evaluated multiple times either in parallel or sequentially. To achieve this, we combine in a novel way the "fast C&C" technique of Lindell [Lin13] with the "LEGO C&C" technique [FJN⁺13, NO09].

---

*©IACR 2014. CRYPTO 2014. This article is a major revision of the version published by Springer-Verlag.

†Indiana University. **E-mail:** `yh33@indiana.edu`. This work was done while at the University of Maryland.

‡University of Maryland. **E-mail:** `{jkatz,amaloz}@cs.umd.edu`

§Bell Labs. **E-mail:** `kolesnikov@research.bell-labs.com`

¶Technion. **E-mail:** `ranjit@cs.technion.ac.il`

Our setting and motivation. We consider the *multiple execution* setting, where two parties compute the same function on possibly different inputs either in parallel or sequentially. Here we argue that multiple evaluations of the same function is indeed a natural and frequently-occurring important scenario.

Today, 2PC is only beginning to enter practical deployment. However, we can reasonably speculate on likely future use cases. In the commercial setting, 2PC is natural in both business-to-business and business-to-customer interactions. For example, a bank customer could perform financial transactions (e.g., payments or transfers), a cell phone customer could perform private location-based queries, two businesses or government agencies might query their joint databases of customers, etc. In all of these scenarios, many of the securely evaluated functions are the same, only differing on their inputs. In fact, we conjecture that single-execution functions may be *less likely* to be used in commercial settings. This is because, as a rule-of-thumb of security, externally-accessible interfaces need to be clean and standardized. Allowing a small number of predetermined customer actions allows for more manageable overall security.

Additionally, many complex protocols from the research literature include multiple executions of the same function evaluated on different inputs. For example, Gordon et al. [GKK+12] propose sublinear 2PC based on oblivious RAM (ORAM). In their protocol, each ORAM step is executed by evaluating the same function using 2PC. Another frequently used subroutine is oblivious PRF, used, e.g., in the previously mentioned sublinear 2PC work [GKK+12] as well as in private database searches [CJJ+13, JJK+13]. A recent such work [PVK+14] traverses the database search tree by evaluating the same match function at each tree node. Finally, any two universal circuits (of the same size) are implementing the same function.

1.1 Preliminaries

Let s denote the statistical security parameter; namely, an adversary can succeed in cheating with probability up to 2^{-s}. Let n denote the computational security parameter. We let t denote the total number of times the parties wish to evaluate a given circuit, and let $\rho = \rho(s,t)$ represent the number of circuits, per evaluation, that need to be generated to achieve an error probability of 2^{-s}. Before discussing our specific technical contribution, we recall the main ideas of our building blocks.

Fast cut-and-choose using cheating punishment [Lin13]. Cut-and-choose (C&C) protocols for GCs work by letting circuit constructor P_1 generate and send a number of GCs to the evaluator P_2, who then chooses a subset of circuits to open and check for correctness. If the checks pass, the remaining circuits are evaluated as in Yao's protocol [Yao86], and the final output is obtained by taking majority over the individual outputs. In concrete terms, prior works [LP07, sS11] required at least 125 circuits to be sent by P_1 to guarantee security 2^{-40}. Lindell's improved technique [Lin13] achieves 2^{-s} security while requiring P_1 to send only s circuits (i.e., 40 circuits for 2^{-40} security).

Lindell's protocol (which we call the "fast C&C" protocol) has two phases. In the first phase, P_1 with input x and P_2 with input y run a modified C&C which ensures that P_2 obtains a proof of cheating ϕ if it receives two inconsistent output values in any two evaluation circuits. Now, if all evaluation circuits produce the same output z, P_2 locally stores z as its output. Both parties *always* continue to the second *cheating-punishment* phase. In it, P_1 and P_2 securely evaluate a *smaller* circuit C', which takes as inputs P_1's input x and P_2's proof ϕ. (P_2 inputs random values if he does not have ϕ.) P_1 proves in zero-knowledge the consistency of its input x between the two phases. C' outputs x to P_2 if ϕ is a valid proof of cheating; otherwise P_2 receives nothing. The efficiency improvement is due to the fact that cheating is *punished* if there is any inconsistency in outputs.

LEGO cut-and-choose [FJN$^+$13, NO09]. These works take a different approach by implementing a two-stage C&C at the *gate* level. The evaluation circuit is then constructed from the unopened garbled gates. In the first stage, P_1 sends multiple garbled gates and P_2 performs a standard C&C with replication factor $\rho(s) = O(s/\log|C|)$. P_2 aborts if any opened gate is garbled incorrectly. In the next stage, P_2 partitions the $\rho(s)|C|$ garbled gates into *buckets* such that each bucket contains $O(\rho(s))$ garbled gates. This two-stage C&C ensures that, except with probability 2^{-s}, each bucket contains a *majority* of correctly constructed garbled gates.

To connect gates with one another, Nielsen and Orlandi [NO09] use homomorphic Pedersen commitments. The resulting computational efficiency is relatively poor as they perform several expensive public-key operations *per gate*. This is addressed in the miniLEGO work [FJN$^+$13], where the authors (among other things) construct homomorphic commitments from oblivious transfer (OT), whose cost can be amortized by OT extension [IKNP03]. However, the overall efficiency of this construction is still lacking in concrete terms due to large constants inside the big-O notation. In particular, the communication efficiency is adversely affected by the use of asymptotically constant-rate codes that are concretely inefficient.

1.1.1 Naïve Approaches to Combining Fast Cut-and-Choose with LEGO.

We now discuss two natural approaches for combining Lindell's fast C&C technique with LEGO-based C&C to achieve protocols secure in the multiple execution setting, which yields baseline benchmarks.

The obvious and uninteresting approach is to simply run a maliciously-secure protocol multiple times. Note that to keep the total failure probability in t *sequential* executions at 2^{-s}, we need to increase the replication factor from s to $s + \log t$. More interestingly, the following LEGO trick, implicit in the work of Nordholt et al. [NNOB12], can help. Consider a circuit $\widetilde{C}$ which consists of t copies of the original circuit C. We perform gate-level LEGO C&C directly on $\widetilde{C}$.[1] Doing this requires $\rho = O(s/\log|\widetilde{C}|) = O(s/(\log|C| + \log t))$. However, while this is a good asymptotic improvement, the concrete efficiency of LEGO protocols is weak due to both heavy public-key machinery per gate [NO09] and expensive communication [FJN$^+$13]. Further, LEGO requires a *majority* of gates in each bucket to be good.

This leads to the second natural approach: use fast C&C in LEGO and require that as long as each bucket contains at least one (as opposed to a majority) correctly constructed garbled gate, the protocol succeeds. Unfortunately, the circuit C' used in the corresponding cheating-punishment phase is no longer small. Indeed, C' has to deliver P_1's input x to P_2 if P_2 supplies a valid cheating proof ϕ. However, the number of possible proofs are now proportional to $|C|$, since such a proof could be generated from any of the $|C|$ buckets. This implies that C' is of size at least $|C|$.[2] Therefore, this approach cannot perform better than evaluating C from scratch using fast C&C.

---

[1] A similar approach (i.e., of directly securely evaluating $\widetilde{C}$) can be used to run Lindell's protocol [Lin13] t times *in parallel* without having to increase the replication factor.

[2] The size of C' is also proportional to computational security parameter n, as the proofs are of length at least $2n$.

1.2 Overview of Our Approach

Our main idea for the multiple execution setting is to run two-stage LEGO C&C at the *circuit* level, and then use fast C&C in the second stage (thereby requiring only a single correctly constructed circuit from each bucket). In particular, now the size of C' used in each execution depends only on the input and output lengths of C, and is no longer proportional to $|C|$. In this section, we focus only on the cut-and-choose aspect of the protocol; namely, on preventing P_1's cheating by submitting incorrect garbled circuits. More detailed protocol descriptions for both the parallel and sequential settings can be found in Section 2 and Section 3.

In the first-stage cut-and-choose, P_1 constructs and sends to P_2 a total of ρt GCs. Next, P_2 requests that P_1 open a random $\rho t/2$-sized subset of the garbled circuits. If P_2 discovers that any opened garbled circuit is incorrectly constructed, it aborts. Otherwise, P_2 proceeds to the second stage cut-and-choose, where it randomly assigns unopened circuits to t buckets such that each bucket contains $\rho/2$ circuits. Now, as in the fast C&C protocol [Lin13], each of the t evaluations are executed in two phases. In the first phase of the kth execution, party P_2 evaluates the $\rho/2$ evaluation circuits contained in the kth bucket. The circuits are designed such that if P_2 obtains different outputs from evaluating circuits in the kth bucket, then it obtains a proof of cheating ϕ_k. Next, both parties continue to the cheating-punishment phase, where P_1 and P_2 securely evaluate a smaller circuit that outputs P_1's input x_k if P_2 provides a valid proof ϕ_k.

Clearly, P_1 succeeds in cheating only if (1) it constructed $m \geq \rho/2$ bad circuits, (2) none of these m bad circuits were caught in the first cut-and-choose stage (i.e., $m \leq \rho t/2$), and (3) in the second stage, there exists a bucket that contains all bad circuits. It is easy to see that the probability with which m bad circuits escape detection in the first stage cut-and-choose is $\binom{\rho t-m}{\rho t/2}/\binom{\rho t}{\rho t/2}$. Conditioned on this event happening, the probability that a particular bucket contains all bad circuits is $\binom{m}{\rho/2}/\binom{\rho t/2}{\rho/2}$. Applying the union bound, we conclude that the probability that P_1 succeeds in cheating is bounded by

$$t\binom{\rho t - m}{\rho t/2}\binom{m}{\rho/2}\Big/\binom{\rho t}{\rho t/2}\binom{\rho t/2}{\rho/2}.$$

For any given t and s, the smallest ρ, hinging on the maximal probability of P_1's successful attack, can be determined by enumerating over all possible values of m (i.e., $\{\rho/2, \rho/2+1, \ldots, \rho t/2\}$).

As an example, for $t = 20$ in a parallel execution setting with $s = 40$, using our protocol the circuit generator needs to construct $16 \cdot t = 320$ garbled circuits, whereas using a naïve application of Lindell's protocol [Lin13] requires $40 \cdot t = 800$ garbled circuits.

Parallel vs. sequential executions. As will be evident, it is important to distinguish between the settings where multiple evaluations are carried out in parallel (e.g., when all inputs are available at the start of the protocol) and where these evaluations are carried out sequentially (e.g., when not all inputs are available as they, for example, depend on the outputs of previous executions). Below, we provide an overview of the main challenges of each setting, and an outline of our solutions.

Parallel executions. Under the DDH assumption, we apply our C&C technique in the parallel execution setting by modifying Lindell's protocol [Lin13] as follows. We construct a generalized *cut-and-choose oblivious transfer* (C&C OT) functionality that supports *multi-stage* cut-and-choose. We call this functionality $\mathcal{F}_{\text{mcot}}$. Asymptotically, we can realize $\mathcal{F}_{\text{mcot}}$ using general secure computation, since the circuit for $\mathcal{F}_{\text{mcot}}$ depends only on the length of P_2's input and is otherwise independent of the circuit. However, such a realization is extremely inefficient in practice (the size

# of Executions	**Replication**	**Replication for Fast C&C**	
	parallel/sequential	*parallel*	*sequential*
2	32	40	41
4	24	40	42
7	20	40	42
20	16	40	44
100	12	40	46
3500	8	40	51

Table 1: The number of garbled circuits required *per execution* in order to guarantee a security loss of $< 2^{-40}$. For comparison, the last two columns show the number of circuits required by the fast C&C protocol [Lin13] in the parallel and sequential settings. Note that when using the fast C&C protocol for sequential executions we need to increase the replication factor from s to $s + \log t$.

of the circuit for realizing $\mathcal{F}_{\text{mcot}}$ needs to accept inputs of length at least $n\rho t\ell$, where n is the computational security parameter and ℓ is the input length). Instead, we show an efficient realization that is only a factor $\rho t^2/s$ less efficient (per execution) than the modified C&C OT realization of Lindell [Lin13]. We elaborate more on this, and other important details, in Section 2.

Sequential executions. To prevent a malicious evaluator from choosing its inputs based on the garbled circuit, GC-based 2PC protocols perform OT *before* the constructor sends its GCs to the evaluator (i.e., before the cut-and-choose phase). This forces the parties, and in particular the evaluator, to "commit" to their inputs before performing the cut-and-choose. This, however, does not work in the sequential setting, where the parties may not know all their inputs at the beginning of the protocol. Standard solutions used in previous works [AIKW13, GGP10, MR13] include assuming the garbled-circuit construction is adaptively secure or using adaptively-secure garbling [BHR12] explicitly, assuming the programmable random-oracle model. Another issue is that since now we perform OTs for each execution separately, we can no longer use C&C OT or its variants; instead we rely on the "XOR-tree" approach of Lindell and Pinkas [LP07] to avoid selective failure attacks. We elaborate more on this, and other details, in Section 3.

Our solution for the sequential setting readily carries over to the parallel setting. In particular, adapting our protocol from the sequential to the parallel setting may address situations where the cost incurred by the use of $\mathcal{F}_{\text{mcot}}$ outweighs the cost of using both the XOR-tree approach and adaptively secure garbled circuits.

1.3 Related Work

Lindell and Pinkas [LP07] gave the first[3] rigorous 2PC protocol based on cut-and-choose. For $s = 40$, their protocol required at least $17s = 680$ garbled circuits. Subsequent work by the same authors [LP11] reduced the number of circuits to 128. This was later improved by shelat and Shen [sS11] to 125 using a more precise analysis of the C&C approach. In Crypto 2013, two works [HKE13, Lin13] proposed (among other things) dramatic improvements to the number of garbled circuits that need to be sent. In more detail, for achieving statistical security 2^{-s}, Huang

---

[3] C&C mechanisms were previously employed in works by Pinkas [Pin03] and Malkhi et al. [MNPS04] but these approaches were later shown to be flawed [KS06, MF06].

et al.'s protocol [HKE13] requires $2s + O(\log s)$ circuits, where each party generates half of them, and Lindell's protocol [Lin13] requires exactly s circuits.

While all of the above works perform cut-and-choose over circuits, applying cut-and-choose at the gate-level has also been considered [DO10, FJN$^+$13, NNOB12, NO09]. As discussed above, this approach naturally extends to the multiple execution setting, and furthermore is not inherently limited to considering settings where the same function is evaluated multiple times. Nielsen et al. [NNOB12] indeed show concrete efficiency improvements using gate-level cut-and-choose techniques. However, the number of rounds grows linearly with the depth of the evaluated circuit.

Finally, in independent and concurrent work, Lindell and Riva [LR14] also investigate the multiple execution setting, and obtain performance improvements similar to ours. An interesting difference between our works is that while we always let the evaluator pick half the circuits to check, they show that varying the number of check circuits can lead to an additional performance improvement.

1.4 Security Definition

We use the standard definition of security for two-party computation in the presence of malicious adversaries [Gol04, Chapter 7]. In this work, we consider the setting where a function is executed t times over different inputs, and explicitly describe the security definitions for such a setting in Appendix A.

2 The Parallel Execution Setting

Consider a setting where two parties wish to securely evaluate the same function multiple times in parallel (see Appendix A.1 for the formal security definition). Let f denote the function of interest, and let t denote the number of times the parties wish to evaluate f. Let P_1's (resp., P_2's) input in the kth execution be x_k (resp., y_k), and let $x = (x_1, \ldots, x_t)$ and $y = (y_1, \ldots, y_t)$. We define $f^{(t)}(x, y) = (f(x_1, y_1), \ldots, f(x_t, y_t))$.

We adapt Lindell's protocol [Lin13] to support our cut-and-choose technique in the parallel execution setting. The main difficulty is the design and construction of a generalization of cut-and-choose oblivious transfer [LP11] which we use to avoid the "selective failure attack" where a malicious P_1 constructs invalid keys for P_2's input wires to try and deduce P_2's inputs based on if P_2 aborts execution or not. We discuss this more in Section 2.1. We note that the naïve idea of using the XOR-tree approach [LP07] in our setting does not appear to work without using adaptively secure garbled circuits. Specifically, it is no longer clear how P_1, without any knowledge of which circuits will end up as evaluation circuits, can batch P_2's input keys together in a way that lets P_2 learn different sets of input keys corresponding to different evaluation circuits and yet within each evaluation bucket guaranteeing that P_2 can learn only input keys corresponding to the same set of inputs.

We give details of our protocol construction for the parallel executions setting in Section 2.2.

2.1 Generalizing Cut-and-Choose Oblivious Transfer

Cut-and-choose oblivious transfer (C&C OT) [LP11] is an extension of standard one-out-of-two oblivious transfer (OT). The sender inputs L pairs of strings, and the receiver inputs L selection bits to select one string out of each pair of sender strings. The receiver also inputs a set J of

Inputs:

- P_1 inputs ℓ vectors $\vec{x}_i$, each containing s pairs of values $x_0^{i,j}, x_1^{i,j} \in \{0,1\}^{n \times n}$, $i \in [\ell]$, $j \in [s]$. In addition, P_1 inputs s "check values" $\chi_1, \ldots, \chi_s \in (\{0,1\}^n)^s$.
- P_2 inputs $\sigma_1, \ldots, \sigma_\ell \in \{0,1\}$ and a set of indices $J \subseteq [s]$.

Outputs: P_1 receives no output. P_2 receives the following:

- For every $i \in [\ell]$ and $j \in J$, P_2 receives $(x_0^{i,j}, x_1^{i,j})$.
- For every $i \in [\ell]$, P_2 receives $\langle x_{\sigma_i}^{i,1}, \ldots, x_{\sigma_i}^{i,s} \rangle$.
- For every $k \notin J$, P_2 receives χ_k.

In other words, P_2 receives $\{\chi_j\}_{j \in [s] \setminus J}$ and $\{\{x_{\sigma_i}^{i,j}\}_{j \in [s] \setminus J}, \{(x_0^{i,j}, x_1^{i,j})\}_{j \in J}\}_{i \in [\ell]}$.

Figure 1: Modified batch single-choice cut-and-choose OT functionality $\mathcal{F}_{\text{ccot}}$ [Lin13].

size $L/2$ that consists of indices where it wants *both* the sender's inputs to be revealed. Note that for indices not contained in J, only those sender inputs that correspond to the receiver's selection bits are revealed. In applications to secure computation, and in particular when transferring input keys corresponding to a particular input wire across all evaluation circuits, one needs *single-choice* cut-and-choose oblivious transfer, where the receiver is restricted to inputting the *same* selection bit in all the $L/2$ instances where it receives exactly one out of two sender strings. Furthermore, when transferring input keys for multiple input wires, it is crucial that the subset J input by the receiver is the same across each instance of single-choice C&C OT executed for all input wires. This variant, called *batch single-choice* C&C OT, can be realized from the decisional Diffie-Hellman problem [LP11].

Lindell [Lin13] presented a variant of batch single-choice C&C OT [LP11] in order to address settings where the check set J input by the receiver may be of arbitrary size. We denote this variant by $\mathcal{F}_{\text{ccot}}$; see Figure 1 for the formal description. In this variant, in addition to obtaining one of the two sender inputs for pairs whose indices are not in J, the receiver also obtains a "check value" for each index not in J. These check values are used to confirm whether or not a circuit is an evaluation circuit.

For our purposes, we introduce a new variant of $\mathcal{F}_{\text{ccot}}$, which we call batch single-choice *multi-stage* C&C OT. We denote this primitive by $\mathcal{F}_{\text{mcot}}$ and present its formal description in Figure 2. At a high level, our variant differs from $\mathcal{F}_{\text{ccot}}$ in that receiver P_2 can now input multiple sets $J_1, \ldots, J_t$ (where J is now implicitly defined as $[\rho t] \setminus \cup_{k \in [t]} J_k$) and make independent selections for each of $J_1, \ldots, J_t$. Unlike in Lindell's scheme [Lin13], we only need to consider sets $J_1, \ldots, J_t$ whose sizes are pre-specified in order to provide the desired security guarantees. However, as in the $\mathcal{F}_{\text{ccot}}$ functionality, $\mathcal{F}_{\text{mcot}}$ (1) does not require sets $J_1, \ldots, J_t$ to be of a particular size, and (2) delivers "check values" for indices contained in each of $J_1, \ldots, J_t$. These check values are used to confirm whether a circuit is an evaluation circuit in the kth bucket for some $k \in [t]$.

Designing the $\mathcal{F}_{\text{mcot}}$ functionality. As in $\mathcal{F}_{\text{ccot}}$, the sender P_1 inputs ℓ vectors $\vec{x}_1, \ldots, \vec{x}_\ell$ each of length ρt, where each element in the vector is a pair of values (corresponding to the 0-key and the 1-key of a given garbled wire). In addition, P_1 inputs ρt^2 "check values". Receiver P_2 inputs t vectors $\vec{\sigma}_1, \ldots, \vec{\sigma}_t$ each of length ℓ and pairwise non-intersecting sets $J_1, \ldots, J_t$. Upon receiving these inputs from P_1 and P_2, the functionality computes $J = [\rho t] \setminus \cup_{k \in [t]} I_k$, and delivers, for each $j \in J$, the jth element (i.e., both values in the jth pair) in each of the ℓ vectors. Next, for every

Inputs:

- P_1 inputs ℓ vectors $\vec{x}_i$, each containing ρt^2 pairs $x_0^{i,j}, x_1^{i,j} \in \{0,1\}^n$. In addition, P_1 inputs ρt^2 "check values" $\chi_1^1, \ldots, \chi_{\rho t}^1; \ldots; \chi_1^t, \ldots, \chi_{\rho t}^t \in \{0,1\}^n$.
- P_2 inputs $\vec{\sigma}_1 = (\sigma_{1,1}, \ldots, \sigma_{1,\ell}), \ldots, \vec{\sigma}_t = (\sigma_{t,1}, \ldots, \sigma_{t,\ell}) \in \{0,1\}^\ell$ and sets $J_1, \ldots, J_t$ that are pairwise non-intersecting subsets of $[\rho t]$.

Outputs: Party P_1 receives no output. Party P_2 receives the following:

- For every $k \in [t]$ and for every $j \in J_k$, party P_2 receives χ_j^k.
- Let $J = [\rho t] \setminus \cup_{k \in [t]} J_k$. For every $i \in [\ell]$ and $j \in [\rho t]$:
  - If $j \in J$, then P_2 receives $(x_0^{i,j}, x_1^{i,j})$.
  - Otherwise, if there exists a (unique) $k \in [t]$ such that $j \in J_k$, then P_2 receives $x_{\sigma_{k,i}}^{i,j}$.

In other words, P_2 receives sets $\{\chi_j^1\}_{j \in J_1}, \ldots, \{\chi_j^t\}_{j \in J_t}$ and $\{\{x_{\sigma_{1,i}}^{i,j}\}_{j \in J_1}, \ldots, \{x_{\sigma_{t,i}}^{i,j}\}_{j \in J_t}, \{(x_0^{i,j}, x_1^{i,j})\}_{j \in J}\}_{i \in [\ell]}$.

Figure 2: Batch single-choice multi-stage cut-and-choose OT functionality $\mathcal{F}_{\text{mcot}}$.

$k \in [t]$ and for each $j \in J_k$, the functionality delivers to P_2 the $\sigma_{k,i}$ value in the jth pair of vector $\vec{x}_i$ for every $i \in [\ell]$ along with the check value χ_j^k.

Realizing $\mathcal{F}_{\text{mcot}}$ in the $\mathcal{F}_{\text{ccot}}$-hybrid model. We now proceed to construct a protocol for $\mathcal{F}_{\text{mcot}}$. Our goal is to provide an information-theoretic reduction from $\mathcal{F}_{\text{mcot}}$ to $\mathcal{F}_{\text{ccot}}$. We first consider a naïve approach which serves as a warm-up to our final construction and provides intuition behind our definition of $\mathcal{F}_{\text{mcot}}$.

The naïve approach. We propose the following natural approach to realizing $\mathcal{F}_{\text{mcot}}$ from $\mathcal{F}_{\text{ccot}}$: P_1 first performs a t-out-of-t additive secret sharing of all input keys corresponding to P_2's inputs. In addition, P_1 chooses ρt^2 check values. Next, P_1 and P_2 interact with the $\mathcal{F}_{\text{ccot}}$ functionality t times in parallel. In the kth interaction, P_1 provides the kth additive share of its input plus ρt check values $\chi_1^k, \ldots, \chi_{\rho t}^k$ (i.e., a check value for each circuit that could potentially be an evaluation circuit in the kth execution), while P_2 provides its inputs for the kth execution along with a set $[\rho t] \setminus J_k$, where J_k indicates the indices of the evaluation circuits to be used in the kth execution. Let $J = [\rho t] \setminus \cup_{k \in [t]} J_k$. At the end of the interaction, P_2 obtains (1) all t additive shares of input keys, and therefore all input keys, for circuits GC_j with $j \in J$, and (2) all t additive shares of input keys that *correspond to its actual input* in the kth execution, and therefore its input keys, along with check values for circuits GC_j with $j \notin J$.

Note, in particular, that for the check circuits, P_2 does not obtain the check values, and for the evaluation circuits, P_2 does not obtain both input keys. Thus, the above protocol seems to successfully fulfill our requirements from the $\mathcal{F}_{\text{mcot}}$ functionality. However, note that there is no mechanism in place to enforce that P_2 supplies non-intersecting sets $J_1, \ldots, J_k$. In the following we show that this prevents the above protocol from realizing $\mathcal{F}_{\text{mcot}}$.

Suppose $t = 2$. A malicious P_2 may input overlapping sets J_1, J_2 to $\mathcal{F}_{\text{ccot}}$. The consequence of this is that P_2 now possesses check values χ_j^1 and χ_j^2 for $j \in J_1 \cap J_2$. Clearly, the functionality $\mathcal{F}_{\text{mcot}}$ does not allow this. On the other hand, recall that the input keys are all additively shared, and as a result P_2 does not possess input keys corresponding to its input in circuit GC_j unless its input in both executions are identical. At the surface, there does not seem to be any attack due to this malicious strategy. Sure, P_2 can now equivocate on assigning GC_j to either the first evaluation bucket or the second evaluation. However, as observed earlier, it either has no corresponding keys, or it is going to evaluate both circuits on the same input, say y (in which case it seems immaterial whether j is revealed as part of J_1 or J_2). Unfortunately, we show that the above strategy for malicious P_2 is not simulatable. In particular, at the end of the interaction with $\mathcal{F}_{\text{ccot}}$, the simulator successfully extracts P_2's input in the first and second execution, but is now unable to decide on how to fake the garbled circuit GC_j. On the one hand, if $j \in J_1$, then the fake garbled circuit has to output $z_1 = f(x_1, y)$. On the other hand, if $j \in J_2$, then the fake garbled circuit has to output $z_2 = f(x_2, y)$. Therefore, the simulator has to choose on how to fake GC_j in the dark. Note that a simulation strategy for this specific case that decides to fake GC_j to output z_1 with probability $1/2$, and to output z_2 with probability $1/2$, does indeed succeed with probability $1/2$. However, this strategy does not extend well to the case when t is large.

The discussion above motivates our definition of $\mathcal{F}_{\text{mcot}}$; in particular, it reinforces why $\mathcal{F}_{\text{mcot}}$ must deliver at most one check value per circuit. In the following, we explain how to modify the naïve construction to enforce this.

Our approach. The high level idea behind our protocol is to let P_1 perform independent additive sharings of both the input values as well as the check values. Then P_1 and P_2 query the $\mathcal{F}_{\text{ccot}}$ functionality t times to transfer the values as required by $\mathcal{F}_{\text{mcot}}$. We detail this below, explaining it in the context of our secure computation protocol.

Let $(x_0^{i,j}, x_1^{i,j})$ be the input keys corresponding to P_2's ith input wire in GC_j. First, P_1 performs a t-out-of-t additive secret sharing of all input values corresponding to P_2's inputs; i.e., for each $i \in [\ell], j \in [\rho t]$, P_1 secret shares $x_0^{i,j}$ (resp., $x_1^{i,j}$) into $\{x_0^{i,j,k}\}_{k \in [t]}$ (resp., $\{x_1^{i,j,k}\}_{k \in [t]}$). P_1 then chooses ρt^2 check values $\{\chi_1^k, \ldots, \chi_{\rho t}^k\}_{k \in [t]}$. It then performs a $(2\ell(t-1)+1)$-out-of-$(2\ell(t-1)+1)$ additive sharing of each value χ_j^k to obtain shares denoted $\widetilde{\chi}_j^k$, $\{\chi_{0,k}^{i,j,k'}, \chi_{1,k}^{i,j,k'}\}_{k' \in [t] \setminus \{k\}, i \in [\ell]}$. Then, instead of creating inputs to $\mathcal{F}_{\text{ccot}}$ using $x_c^{i,j,k}$ shares alone, P_1 instead creates a "share block" $X_c^{i,j,k} = (x_c^{i,j,k}, \chi_{c,1}^{i,j,k}, \ldots, \chi_{c,t}^{i,j,k})$. That is, a share block $X_c^{i,j,k}$ contains, in addition to a share of the input key, a share of all check values corresponding to circuit GC_j.

Next, P_1 and P_2 run t instances of $\mathcal{F}_{\text{ccot}}$ in parallel. In the kth interaction, in addition to the ρt check value shares $\widetilde{\chi}_1^k, \ldots, \widetilde{\chi}_{\rho t}^k$, P_1 provides its kth share block while P_2 provides its inputs for the kth execution along with a set $[\rho t] \setminus J_k$, where J_k indicates the indices of the evaluation circuits to be used in the kth execution. Let $J = [\rho t] \setminus \cup_{k \in [t]} J_k$. At the end of the interaction, P_2 obtains (1) all t share blocks of input keys, and therefore all input keys, for circuits GC_j with $j \in J$, and (2) all t share blocks of input keys that *correspond to its actual input* in the kth execution, and therefore its input keys, along with a check value $\widetilde{\chi}_j^k$ for circuits GC_j with $j \in J_k$.

Note, in particular, that for each check circuit GC_j, P_2 does not obtain the check value χ_j^k for any k, because it always misses the check value share $\widetilde{\chi}_j^k$. For each evaluation circuit GC_j with $j \in J_k$, P_2 does not obtain both input keys, and more importantly can obtain at most one check value (which is χ_j^k). This is because share blocks contain shares of input keys as well as shares of check values. For an evaluation circuit, party P_2 always misses a share block, and consequently shares of all values $\chi_j^{k'}$ with $k' \neq k$. Furthermore, if P_2 wants to ensure it receives χ_j^k, then it should never input $J_{k''}$ such that $k'' \neq k$ and yet $j \in J_{k''}$. This is because for $j \in J_{k''}$, P_2 is guaranteed to miss a share block that contains an additive share of χ_j^k. Note that the above observations suffice to deal with a malicious P_2 that inputs overlapping sets since in this case P_2 fails to obtain any check values corresponding to indices in the intersection.

The formal description of the protocol in the $\mathcal{F}_{\text{ccot}}$-hybrid model can be found in Figure 3. We

Inputs:

- P_1 inputs ℓ vectors of pairs $\vec{x}_i = \langle (x_0^{i,1}, x_1^{i,1}), \ldots, (x_0^{i,\rho t}, x_1^{i,\rho t}) \rangle$ for $i \in [\ell]$. In addition, P_1 inputs ρt^2 "check values" $(\chi_1^1, \ldots, \chi_{\rho t}^1), \ldots, (\chi_1^t, \ldots, \chi_{\rho t}^t)$. All values are in $\{0,1\}^n$.
- P_2 inputs $\vec{\sigma}_1 = (\sigma_{1,1}, \ldots, \sigma_{1,\ell}), \ldots, \vec{\sigma}_t = (\sigma_{t,1}, \ldots, \sigma_{t,\ell}) \in \{0,1\}^\ell$ and sets $J_1, \ldots, J_t$.

Protocol:

- For all $i \in [\ell]$, P_1 performs a t-out-of-t additive secret sharing of $\vec{x}_i$ to obtain shares $\vec{x}_{i,1}, \ldots, \vec{x}_{i,t}$. For $k \in [t]$, let $\vec{x}_{i,k} = \langle (x_0^{i,1,k}, x_1^{i,1,k}), \ldots, (x_0^{i,\rho t,k}, x_1^{i,\rho t,k}) \rangle$. Let $X_0^{i,j,k} = (x_0^{i,j,k}, \chi_{0,1}^{i,j,k}, \ldots, \chi_{0,t}^{i,j,k})$ and $X_1^{i,j,k} = (x_1^{i,j,k}, \chi_{1,1}^{i,j,k}, \ldots, \chi_{1,t}^{i,j,k})$, where $\chi_{0,1}^{i,j,k}, \ldots, \chi_{0,t}^{i,j,k}$ and $\chi_{1,1}^{i,j,k}, \ldots, \chi_{1,t}^{i,j,k}$ are random independent values in $\{0,1\}^n$. Let $\vec{X}_{i,k} = \langle (X_0^{i,1,k}, X_1^{i,1,k}), \ldots, (X_0^{i,\rho t,k}, X_1^{i,\rho t,k}) \rangle$.
- For all $k \in [t]$ and $j \in [\rho t]$, set $\tilde{\chi}_j^k = \chi_j^k \oplus \bigoplus_{k' \in [t] \setminus \{k\}, i \in [\ell]} (\chi_{0,k}^{i,j,k'} \oplus \chi_{1,k}^{i,j,k'})$.
- P_1 and P_2 run t instances of $\mathcal{F}_{\mathrm{ccot}}$ in parallel as follows. In the kth instance:
  - P_1 inputs ℓ vectors of pairs $\vec{X}_{i,k}$ of length ρt for $i \in [\ell]$ and ρt "check values" $\widetilde{\chi}_1^k, \ldots, \widetilde{\chi}_{\rho t}^k$. P_2 inputs $\sigma_{k,1}, \ldots, \sigma_{k,\ell} \in \{0,1\}$ and the set $[\rho t] \setminus J_k$.
  - P_2 receives $\{\widetilde{\chi}_j^k\}_{j \in J_k}$ and $\{\{X_{\sigma_{k,i}}^{i,j,k}\}_{j \in J_k} \cup \{(X_0^{i,j,k}, X_1^{i,j,k})\}_{j \in [\rho t] \setminus J_k}\}_{i \in [\ell]}$.
- For all $k \in [t]$ and $j \in J_k$, P_2 reconstructs $\chi_j^k = \tilde{\chi}_j^k \oplus \bigoplus_{k' \in [t] \setminus \{k\}, i \in [\ell]} (\chi_{0,k}^{i,j,k'} \oplus \chi_{1,k}^{i,j,k'})$.
- Let $J = [\rho t] \setminus \cup_{k \in [t]} J_k$. For all $i \in [\ell]$ and $j \in [\rho t]$, P_2 does the following:
  - If $j \in J$: set $x_0^{i,j} = \bigoplus_{k \in [t]} x_0^{i,j,k}$, and $x_1^{i,j} = \bigoplus_{k \in [t]} x_1^{i,j,k}$.
  - If there exists (unique) $k \in [t]$ such that $j \in J_k$: set $x_{\sigma_{k,i}}^{i,j} = \bigoplus_{k \in [t]} x_{\sigma_{k,i}}^{i,j,k}$.
- P_2 outputs sets $\{\chi_j^1\}_{j \in J_1}, \ldots, \{\chi_j^t\}_{j \in J_t}$ and $\{\{(x_0^{i,j}, x_1^{i,j})\}_{j \in J}, \{x_{\sigma_{1,i}}^{i,j}\}_{j \in J_1}, \ldots, \{x_{\sigma_{t,i}}^{i,j}\}_{j \in J_t}\}_{i \in [\ell]}$.

Figure 3: Realizing $\mathcal{F}_{\mathrm{mcot}}$ in the $\mathcal{F}_{\mathrm{ccot}}$-hybrid model.

prove the following.

Theorem 1. *There exists a protocol perfectly realizing $\mathcal{F}_{\mathrm{mcot}}$ in the $\mathcal{F}_{\mathrm{ccot}}$-hybrid model.*

Proof (Sketch). Consider the protocol described in Figure 3. We prove that this protocol realizes $\mathcal{F}_{\mathrm{mcot}}$ in the $\mathcal{F}_{\mathrm{ccot}}$-hybrid model. We split the analysis into two cases depending on whether P_1 or P_2 is corrupted.

P_1 is corrupted. The simulation is straightforward since P_1 does not receive any output. We describe it below.

- For each $k \in [t]$, acting as $\mathcal{F}_{\mathrm{ccot}}$ simulator $\mathcal{S}$ obtains the following from P_1: (1) ℓ vectors of pairs $\vec{X}_{i,k} = \langle (X_0^{i,1,k}, X_1^{i,1,k}), \ldots, (X_0^{i,\rho t,k}, X_1^{i,\rho t,k}) \rangle$ of length ρt for $i \in [\ell]$ and (2) ρt "check values" $\tilde{\chi}_1^k, \ldots, \tilde{\chi}_{\rho t}^k$.
- For $c \in \{0,1\}$, $i \in [\ell]$, $j \in [\rho t]$, $k \in [t]$, simulator $\mathcal{S}$ parses $X_c^{i,j,k}$ as $(x_c^{i,j,k}, \chi_{c,1}^{i,j,k}, \ldots, \chi_{c,t}^{i,j,k})$.
- For each $i \in [\ell]$, simulator $\mathcal{S}$ constructs $\vec{x}_i = \langle (x_0^{i,1}, x_1^{i,1}), \ldots, (x_0^{i,\rho t}, x_1^{i,\rho t}) \rangle$, where for $c \in \{0,1\}$ and $j \in [\rho t]$, $x_c^{i,j} = \bigoplus_{k \in [t]} x_c^{i,j,k}$.
- For each $j \in [\rho t]$ and each $k \in [t]$, simulator $\mathcal{S}$ computes $\chi_j^k = \tilde{\chi}_j^k \oplus \bigoplus_{k' \in [t] \setminus \{k\}, i \in [\ell]} (\chi_{0,k}^{i,j,k'} \oplus \chi_{1,k}^{i,j,k'})$.

- $\mathcal{S}$ sends ℓ vectors of pairs $\vec{x}_i$ of length ρt, for $i \in [\ell]$, and ρt^2 "check values" $(\chi_1^1, \ldots, \chi_{\rho t}^1)$, …, $(\chi_1^t, \ldots, \chi_{\rho t}^t)$ to $\mathcal{F}_{\text{mcot}}$ and terminates outputting whatever P_1 outputs.

P_2 **is corrupted.** The simulation is slightly tricky since a malicious P_2 may input sets $J_1, \ldots, J_t$ that are intersecting to $\mathcal{F}_{\text{ccot}}$. For clarity, we denote the (effective) sets input by P_2 as $I_1, \ldots, I_t$. The key observation is that none of the input values or check values are determined until P_2 completes its *final* query to $\mathcal{F}_{\text{ccot}}$. Due to symmetry and hence without loss of generality, in the following, we assume P_2 last query to $\mathcal{F}_{\text{ccot}}$ is its tth query. We describe the simulation below.

- For each $1 \leq k < t$ acting as $\mathcal{F}_{\text{ccot}}$ simulator $\mathcal{S}$ interacts with P_2 for the kth query in the following way:
  - $\mathcal{S}$ obtains the following from P_2: (1) $\sigma_{k,1}, \ldots, \sigma_{k,\ell}$ and (2) the set $[\rho t] \setminus I_k$. Let $\vec{\sigma}_k = (\sigma_{k,1}, \ldots, \sigma_{k,\ell})$.
  - $\mathcal{S}$ chooses uniformly random and independent values $X_0^{i,j,k} = (x_0^{i,j,k}, \chi_{0,1}^{i,j,k}, \ldots, \chi_{0,t}^{i,j,k})$ and $X_1^{i,j,k} = (x_1^{i,j,k}, \chi_{1,1}^{i,j,k}, \ldots, \chi_{1,t}^{i,j,k})$ for each $i \in [\ell], j \in [\rho t]$. In addition, $\mathcal{S}$ chooses uniformly random and independent values $\tilde{\chi}_1^k, \ldots, \tilde{\chi}_{\rho t}^k$.
  - $\mathcal{S}$ sends $\{\tilde{\chi}_j^k\}_{j \in I_k}$, $\{\{X_{\sigma_{k,i}}^{i,j,k}\}_{j \in I_k} \cup \{(X_0^{i,j,k}, X_1^{i,j,k})\}_{j \in [\rho t] \setminus I_k}\}_{i \in [\ell]}$ to P_2.
- Acting as $\mathcal{F}_{\text{ccot}}$ simulator $\mathcal{S}$ first obtains the tth query from P_2 as (1) $\sigma_{t,1}, \ldots, \sigma_{t,\ell}$, and (2) the set $[\rho t] \setminus I_t$.
- For each $k \in [t]$, $\mathcal{S}$ sets $\vec{\sigma}_k = (\sigma_{k,1}, \ldots, \sigma_{k,\ell})$. For each $k \in [t]$, let $J_k = I_k \setminus \cup_{k' \neq k} I_{k'}$. Define $J = [\rho t] \setminus \cup_{k \in [t]} J_k$. $\mathcal{S}$ sends $\vec{\sigma}_1, \ldots, \vec{\sigma}_t$ and sets $J_1, \ldots, J_t$ to $\mathcal{F}_{\text{mcot}}$, and receives back $\{\chi_j^1\}_{j \in J_1}, \ldots, \{\chi_j^t\}_{j \in J_t}$, $\{\{(x_0^{i,j}, x_1^{i,j})\}_{j \in J}, \{x_{\sigma_{1,i}}^{i,j}\}_{j \in J_1}, \ldots, \{x_{\sigma_{t,i}}^{i,j}\}_{j \in J_t}\}_{i \in [\ell]}$.
- $\mathcal{S}$ chooses values $\{\tilde{\chi}_j^t\}_{j \in [\rho t]}$ as follows:
  - If $j \in J_t$, then set $\tilde{\chi}_j^t = \chi_j^t \oplus \bigoplus_{k \in [t-1], i \in [\ell]} (\chi_{0,t}^{i,j,k} \oplus \chi_{1,t}^{i,j,k})$.
  - Else, choose $\tilde{\chi}_j^t$ uniformly at random.
- $\mathcal{S}$ chooses values $\{x_0^{i,j,t}, x_1^{i,j,t}\}_{i \in [\ell], j \in [\rho t]}$ as follows:
  - If $j \in J$, then for all $i \in [\ell]$ set $x_0^{i,j,t} = x_0^{i,j} \oplus \bigoplus_{k \in [t-1]} x_0^{i,j,k}$ and $x_1^{i,j,t} = x_1^{i,j} \oplus \bigoplus_{k \in [t-1]} x_1^{i,j,k}$.
  - Else if $j \in J_k$ for some (unique) $k \in [t]$, then for all $i \in [\ell]$ set $x_{\sigma_{k,i}}^{i,j,t} = x_{\sigma_{k,i}}^{i,j} \oplus \bigoplus_{k' \in [t-1]} x_{\sigma_{k,i}}^{i,j,k'}$, and $x_{1-\sigma_{k,i}}^{i,j,t}$ to a random value.
- $\mathcal{S}$ chooses values $\{\chi_{0,k}^{i,j,t}, \chi_{1,k}^{i,j,t}\}_{i \in [\ell], j \in [\rho t], k \in [t]}$ as follows:
  - If $j \in J_k$ for some (unique) $k \in [t]$, then for all $i \in [\ell]$ pick $\chi_{0,k}^{i,j,t}, \chi_{1,k}^{i,j,t}$ uniformly at random subject to $\bigoplus_{i \in [\ell]} (\chi_{0,k}^{i,j,t} \oplus \chi_{1,k}^{i,j,t}) = \tilde{\chi}_j^k \oplus \chi_j^k \oplus \bigoplus_{k' \in [t-1], i \in [\ell]} (\chi_{0,k}^{i,j,k'} \oplus \chi_{1,k}^{i,j,k'})$.
  - Else, for all $i \in [\ell], k \in [t]$, pick $\chi_{0,k}^{i,j,t}, \chi_{1,k}^{i,j,t}$ uniformly at random.
- For all $i \in [\ell], j \in [\rho t]$, let $X_0^{i,j,t} = (x_0^{i,j,t}, \chi_{0,1}^{i,j,t}, \ldots, \chi_{0,t}^{i,j,t})$, and $X_1^{i,j,t} = (x_1^{i,j,t}, \chi_{1,1}^{i,j,t}, \ldots, \chi_{1,t}^{i,j,t})$. Then, acting as $\mathcal{F}_{\text{ccot}}$ simulator $\mathcal{S}$ sends $\{\tilde{\chi}_j^t\}_{j \in I_t}$, $\{\{X_{\sigma_{t,i}}^{i,j,t}\}_{j \in I_t} \cup \{(X_0^{i,j,t}, X_1^{i,j,t})\}_{j \in [\rho t] \setminus I_t}\}_{i \in [\ell]}$ to P_2, and terminates outputting whatever malicious P_2 outputs.

First we show that if malicious P_2 inputs $I_1, \dots, I_t$ such that these sets are pairwise non-intersecting, then its view in the above simulation is identically distributed to its view in the real execution. In this case, it is easy to see that for all $k \in [t]$ the extracted sets J_k in the simulation are identical to I_k input by P_2. Further, $J = [\rho t] \setminus \cup_{k\in[t]} I_k$ also holds. Observe that for $j \neq j'$ the randomness used by honest P_1 in the real execution to create values $\{X^{i,j,k}_0, X^{i,j,k}_1\}_{i,k}$ and the randomness used to create $\{X^{i,j',k}_0, X^{i,j',k}_1\}_{i,k}$ are independent of each other. Clearly, this is also the case in the simulated execution. This allows us to split the analysis depending on the value of j.

- For $j \in J_k$, the values $\{x^{i,j,k'}_{\sigma_{k,i}}\}_{k'\in[t]}$ are identically distributed in both executions (i.e., uniformly random and independent subject to $\oplus_{k'\in[t]} x^{i,j,k'}_{\sigma_{k,i}} = x^{i,j}_{\sigma_{k,i}}$). Furthermore, the view of P_2 is independent of the values $x^{i,j}_{1-\sigma_{k,i}}$ since these are information-theoretically hidden from the real execution (as is the case in the ideal execution). This is because in the kth query to $\mathcal{F}_{\text{ccot}}$ party P_2 did not receive one of the additive shares of $x^{i,j}_{1-\sigma_{k,i}}$, i.e., $x^{i,j,k}_{1-\sigma_{k,i}}$. Next, it is easy to verify that the check values χ^k_j and its additive shares $\tilde{\chi}^k_j$, $\{\chi^{i,j,k'}_{0,k}, \chi^{i,j,k'}_{1,k}\}_{k'\in[t]\setminus\{k\}, i\in[\ell]}$ are also identically distributed in both executions. Also, we claim that the view of P_2 in the real execution is independent of the values $\{\chi^{k'}_j\}_{k'\neq k}$. This is because in the kth query to $\mathcal{F}_{\text{ccot}}$ party P_2 did not receive, for every $k' \neq k$, at least one of the additive shares of $\chi^{k'}_j$, e.g., $\chi^{1,j,k}_{0,k'}$.

- For $j \in J$, the values $\{x^{i,j,k'}_0, x^{i,j,k'}_1\}_{i\in[\ell], k'\in[t]}$ are identically distributed in both executions (i.e., uniformly random and independent subject to $\oplus_{k'\in[t]} x^{i,j,k'}_0 = x^{i,j}_0$ and $\oplus_{k'\in[t]} x^{i,j,k'}_1 = x^{i,j}_1$). Furthermore, we claim that the view of P_2 in the real execution is independent of the values $\{\chi^k_j\}_{k\in[t]}$. This is because in the kth query to $\mathcal{F}_{\text{ccot}}$ party P_2 did not receive, for every $k \in [t]$, exactly one of the additive shares of χ^k_j, i.e., $\tilde{\chi}^k_j$.

Given the above, it follows that the view of malicious P_2 in the simulated execution is identically distributed to its view in the real execution.

Now we need to consider the case when malicious P_2 inputs sets $I_1, \dots, I_t$ but these are no longer pairwise non-intersecting. We define sets $J_k = I_k \setminus \cup_{k'\neq k} I_{k'}$ for each $k \in [t]$. Also, define $J_0 = [\rho t] \setminus \cup_{k\in[t]} I_k$, and $J = [\rho t] \setminus \cup_{k\in[t]} J_k$. As in the case when $I_1, \dots, I_k$ were pairwise non-intersecting, we will split the analysis depending on the value of j. It is easy to verify that the analysis in the cases when $j \in J_k$ is identical to its counterpart in the case when $I_1, \dots, I_k$ were pairwise non-intersecting. Likewise the analysis in the cases when $j \in J_0$ is identical to the analysis in $j \in J$ cases when $I_1, \dots, I_k$ were pairwise non-intersecting. We only need to analyse the case when $j \in J_0 \setminus J$. Such a j would exist only when there exists distinct $k, k' \in [t]$ such that $j \in I_k$ and $j \in I_{k'}$. In this case, note that by construction, the simulated values for $\{x^{i,j,k''}_0, x^{i,j,k''}_1\}_{i\in[\ell], k''\in[t]}$ are consistent with actual input values $\{x^{i,j}_0, x^{i,j}_1\}_{i\in[\ell]}$, and thus the shares obtained by P_2 corresponding to the $x^{i,j}_0, x^{i,j}_1$ values are identically distributed. It remains to show that as in the simulated execution, the view of P_2 in the real execution is independent of the values $\{\chi^{k''}_j\}_{k''\in[t]}$. Indeed, we claim that when $j \in I_k$ the value χ^k_j is independent of its view if there exists $k' \neq k$ such that $j \in I_{k'}$. This is because for $j \in I_k$, the value χ^k_j can be reconstructed only if all its additive shares $\tilde{\chi}^k_j$, $\{\chi^{i,j,k''}_{0,k}, \chi^{i,j,k''}_{1,k}\}_{k''\in[t]\setminus\{k\}, i\in[\ell]}$ are obtained. However, if $j \in I_{k'}$, then in the k'th query to $\mathcal{F}_{\text{ccot}}$ party P_2 loses its chance to receive at least one of the additive shares of χ^k_j, e.g., $\chi^{1,j,k'}_{0,k}$. Thus, we conclude that the claim holds. This completes the proof that the view of malicious P_2 in the simulated execution is identically distributed as in the real execution. ☐

2.1.1 Cost of Realizing $\mathcal{F}_{\text{mcot}}$ from DDH

As described, the cost of realizing $\mathcal{F}_{\text{mcot}}$ is t times the cost of realizing $\mathcal{F}_{\text{ccot}}$ for ℓ vectors of pairs of length ρt with each element of size $(t+1)n$. Thus if we use Lindell's existing $\mathcal{F}_{\text{ccot}}$ construction [Lin13] in order to implement $\mathcal{F}_{\text{mcot}}$ from DDH, then for each of the t executions we need to use $9\rho\ell t$ fixed-base exponentiations and $1.5\rho\ell t$ regular exponentiations, and need to send a total of $5\rho\ell t$ group elements. However, note we need to use a group of much larger size (in order to support elements of size $(t+1)n$). This has the adverse effect of drastically reducing the computational efficiency as now we need to perform modular exponentiations over much larger groups.

Fortunately, the situation can be remedied using "length extension" techniques for $\mathcal{F}_{\text{ccot}}$. Specifically, first we realize the protocol for $\mathcal{F}_{\text{ccot}}$ as above except we replace each actual element, say $X_0^{i,j,k}$, that needs to be transferred by a single group element, say $K_0^{i,j,k}$, that will be interpreted as a "key". Then, once the protocol for $\mathcal{F}_{\text{ccot}}$ is executed, the sender now sends encryptions of the each actual element under the corresponding key (e.g., $G(K_0^{i,j,k}) \oplus X_0^{i,j,k}$ where G is a PRG). As is the case with 1-out-of-2 OT length extension, this length extension transformation for $\mathcal{F}_{\text{ccot}}$ is also UC-secure. The proof is also identical to the case for 1-out-of-2 OT length extension and is omitted. In summary, by tolerating an additional cost of sending $2\rho\ell t^2$ symmetric elements (for each of the t executions), we can work over standard DDH groups as in Lindell's protocol [Lin13].

2.1.2 Alternative Approaches

As discussed before, $\mathcal{F}_{\text{mcot}}$ can be realized using general secure computation, but this results in extremely poor efficiency. In particular, the circuit computing $\mathcal{F}_{\text{mcot}}$ is of size at least $n\rho\ell t$, and realization by state-of-the-art secure protocols would further include a multiplicative ns overhead. We leave a more efficient realization of $\mathcal{F}_{\text{mcot}}$ from either $\mathcal{F}_{\text{ccot}}$ or directly from DDH as an open question.

In settings where the $\rho t^2/s$ multiplicative overhead of realizing $\mathcal{F}_{\text{mcot}}$ through our protocol is expensive relative to the size of the circuit, one may wonder whether it is possible to use XOR-tree approaches to obtain better efficiency. Unfortunately, we do not know if this approach can be made to work with standard Yao garbled circuits [LP07]. Specifically, it is no longer clear how P_1, without any knowledge of the evaluation sets, can batch P_2's input keys together in a way that lets P_2 learn different sets of input keys corresponding to different evaluation circuits and yet within each evaluation bucket guaranteeing that P_2 can learn only input keys corresponding to the same set of inputs. However, if we assume that the garbling scheme is adaptively secure, then this lets us perform the oblivious transfer step after P_1 commits to its garbled circuits. Now P_2 can reveal its evaluation buckets one-by-one thereby letting P_1 to successfully batch P_2's input keys in the right manner. (See our protocol for sequential executions in Section 3 for a full description on how to do this.)

Finally, we note that the overhead of implementing the XOR-tree along with the necessary commitments can be quite prohibitive for certain choices of parameters [LP11, LPS08], and a careful comparison with our $\mathcal{F}_{\text{mcot}}$ realization is advised before using XOR-tree type constructions.

2.2 Using $\mathcal{F}_{\text{mcot}}$ in the Parallel Execution Setting

The input vectors $\vec{x}_i$, for $i \in [\ell]$, contain the key pairs associated with the ith input wire for P_2 in each of the ρt circuits. The vector $\vec{\sigma}_k$ corresponds to the inputs used by P_2 in the kth execution. An honest P_2 chooses sets $J_1, \ldots, J_t$ such that they are pairwise non-intersecting and each set is of size exactly $\rho/2$. The main observation is that, for a given execution $k \in [t]$, P_2 obtains check values χ_j^k from $\mathcal{F}_{\text{mcot}}$ only for $j \in J_k$. Therefore, once the parties complete the interaction with $\mathcal{F}_{\text{mcot}}$ and P_1 sends all the garbled circuits, we let P_1 determine the evaluation circuits in each bucket based on whether P_2 sends the corresponding check values. At this point, P_1 checks that each bucket of evaluation circuits is well-defined and that these buckets are of equal size, i.e., $\rho/2$. If not, P_1 aborts. To overcome technical difficulties, we also require P_2 to provide "check values" for the check circuits as well. A check value for check circuit GC_j, denoted χ_j, may simply be the set of all input keys (i.e., both the 0-key and the 1-key) on all wires in circuit GC_j.

Applying the cheating-punishment technique. Inspired by Lindell's protocol [Lin13], we use the knowledge of two different garbled values for a single output wire as a "proof" that P_2 received inconsistent outputs in a given execution. P_2 can use this proof to obtain P_1's input in a cheating-punishment phase. This cheating-punishment phase is implemented via a secure computation protocol, and thus it is important that the second phase functionality has a small circuit. We employ several optimizations proposed by Lindell [Lin13] to keep the size of this circuit small. One important difference in our setting is that, unlike in Lindell's protocol [Lin13], we cannot have, for a given output wire w, the same output keys b_w^0, b_w^1 across all garbled circuits. This is because in our setting garbled circuits are assigned to different evaluation buckets, and the circuits in each bucket can be evaluated with different input values, and thus can produce different outputs. Thus (even an honest) P_2 could potentially learn, say, output key b_w^0 in one execution and output key b_w^1 in another. We address this by simply removing the requirement that the set of output keys across different garbled circuits are the same. Thus, the circuit for the cheating-punishment phase for the kth execution must now take as input from P_1 *all* of the output keys in *all* of the evaluation circuits in the kth bucket, and from P_2 a pair of output keys that serve as proof of cheating. Somewhat surprisingly, we show that the size of the circuit (measured as the number of non-XOR gates) for the cheating-punishment phase is essentially the same as the circuit in Lindell's protocol [Lin13].[4]

Another detail we wish to point out is that in our protocol we need to run separate cheating-punishment phases for each execution. This is a restriction imposed by the way in which P_1 proves consistency of its inputs [Lin13, LP11]. However, we can run all of the t cheating-punishment phases *in parallel*. For this reason we use the universally composable variant of Lindell and Pinkas's protocol [LP11] (which is essentially obtained by replacing oblivious transfers and zero-knowledge subprotocols with their universally composable variants) to implement each cheating-punishment phase.

Other details. We now describe other important details of our protocol.

- *Input consistency across multiple executions.* It is important to guarantee that P_1 provides consistent inputs across all circuits in the kth execution. Fortunately, existing mechanisms [Lin13, LP11] for ensuring input consistency in the single execution setting can be readily extended to the multiple execution setting as well.

---

[4] Of course, the cost of realizing our cheating-punishment phase is more than the corresponding cost in Lindell's protocol [Lin13], mainly due to P_1's input being larger (but only by a factor of $\rho/2$).

- *Encoded translation tables for garbled circuits.* As in Lindell's protocol [Lin13], we modify the output translation tables used in the garbled circuits. Specifically, for keys k_i^0, k_i^1 on output wire i, we create an *encoded* output table $[h(k_i^0), h(k_i^1)]$, where h is some one-way function. We require that the output keys (or more precisely, the output of h applied to the output keys) corresponding to 0 and 1 are distinct. This encoding gives us the following two properties: (1) P_2 after evaluating a garbled circuit can use the encoded translation tables to determine whether the output is 0 or 1, and (2) the encoded translation table does not reveal the other output key (since this is equivalent to inverting the one-way function) to P_2.

- *Optimizing the cheating-punishment circuit.* We can apply similar techniques as shown by Lindell [Lin13] to optimize the size of the cheating-punishment circuit to contain only ℓ non-XOR gates. See Section 2.2.1.

Formal description. We proceed to the formal description of our protocol.

Inputs: P_1 has input $x = (x_1, \ldots, x_t)$, where $x_k \in \{0,1\}^\ell$, and P_2 has input $y = (y_1, \ldots, y_t)$, where $y_k \in \{0,1\}^\ell$.

Auxiliary Inputs: A statistical security parameter s, a computational security parameter n, the description of a circuit C where $C(x,y) = f(x,y)$, the number of evaluations t of the function f, and $(\mathbb{G}, q, g)$ where $\mathbb{G}$ is a cyclic group with generator g and prime order q, where q is of length n. Let $\mathsf{Ext} : \mathbb{G} \to \{0,1\}^n$ be a function mapping group elements to bitstrings. In the following, $\rho = \rho(s,t)$ is the replication factor defined as being the smallest $u \in \mathbb{N}$ such that for all $m \in \{u/2, \ldots, ut/2\}$ it holds that $t \cdot \binom{ut-m}{ut/2}\binom{m}{u/2} / \binom{ut}{ut/2}\binom{ut/2}{u/2} \le 2^{-s}$. If no such u exists or if $\rho \ge s$, then parties abort this protocol, and instead run the fast C&C protocol [Lin13] for the function $f^{(t)}$.

Outputs: P_2 receives $f^{(t)}(x,y)$ and P_1 receives no output. Let ℓ' denote the length of the output of $f(x,y)$.

Protocol:

1. **Input key choice and circuit preparation:**
    - P_1 chooses random values $a_1^0, a_1^1, \ldots, a_\ell^0, a_\ell^1 \in_R \mathbb{Z}_q$, $r_1, \ldots, r_{\rho t} \in_R \mathbb{Z}_q$ and $(b_{1,1}^0, b_{1,1}^1, \ldots, b_{1,\ell'}^0, b_{1,\ell'}^1), \ldots, (b_{\rho t,1}^0, b_{\rho t,1}^1, \ldots, b_{\rho t,\ell'}^0, b_{\rho t,\ell'}^1) \in_R \{0,1\}^{n\ell'}$ such that for every $c_1, c_2 \in \{0,1\}, j_1, j_2 \in [\rho t], i_1, i_2 \in [\ell']$ it holds that $b_{j_1,i_1}^{c_1} = b_{j_2,i_2}^{c_2}$ iff $i_1 = i_2$ and $j_1 = j_2$ and $c_1 = c_2$.
    - Let $w_1, \ldots, w_\ell$ denote the input wires corresponding to P_1's input, let $w_{i,j}$ denote the ith input wire in the jth garbled circuit, and let $k_{i,j}^b$ denote the key associated with bit b on wire $w_{i,j}$. P_1 sets $k_{i,j}^b$ as follows:
    $$k_{i,j}^0 = \mathsf{Ext}(g^{a_i^0 \cdot r_j}) \quad \text{and} \quad k_{i,j}^1 = \mathsf{Ext}(g^{a_i^1 \cdot r_j}).$$
    - Let $w'_1, \ldots, w'_{\ell'}$ denote the output wires. The keys for wire w'_i in the jth garbled circuit are set to $b_{j,i}^0$ and $b_{j,i}^1$.
    - P_1 constructs ρt independent garblings, $GC_1, \ldots, GC_{\rho t}$, of circuit C, using random keys except for wires $w_1, \ldots, w_\ell$ and $w'_1, \ldots, w'_m$, where the keys are set as above.
2. **Oblivious transfers:** P_1 and P_2 run $\mathcal{F}_{\mathrm{mcot}}$ as follows:

- For $i \in [\ell]$, let $\vec{z}_i$ denote a vector containing the ρt pairs of keys associated with P_2's ith input bit in all the garbled circuits. P_1 inputs $\vec{z}_1, \ldots, \vec{z}_\ell$, as well as random values $\chi_1^1, \ldots, \chi_{\rho t}^1; \ldots; \chi_1^t, \ldots, \chi_{\rho t}^t$.
- P_2 inputs random sets $J_1, \ldots, J_t$ which are pairwise non-intersecting subsets of $[\rho t]$ such that for all $k \in [t]$ it holds that $|J_k| = \rho/2$. Let $J = [\rho t] \setminus \cup_{k \in [t]} J_k$. P_2 also inputs bits $(\sigma_{1,1}, \ldots, \sigma_{1,\ell}), \ldots, (\sigma_{t,1}, \ldots, \sigma_{t,\ell}) \in \{0,1\}^\ell$, where $\sigma_{k,i} = y_{k,i}$ for every $i \in [\ell]$ and $k \in [t]$.
- For $j \in J$, P_2 receives both input keys associated with its input wires in garbled circuit GC_j, and for each $k \in [t]$ and $j \in J_k$, P_2 receives the keys associated with its input y_k on its input wires in garbled circuit GC_j. Also, for every $k \in [t]$ and $j \in J_k$, P_2 receives χ_j^k.

3. **Send circuits and commitments:** P_1 sends P_2 the garbled circuits $GC_1, \ldots, GC_{\rho t}$, the "seed" for the randomness extractor Ext, the following commitment to the garbled values associated with P_1's input wires:
$$\{(i, 0, g^{a_i^0}), (i, 1, g^{a_i^1})\}_{i \in [\ell]} \quad \text{and} \quad \{(j, g^{r_j})\}_{j=1}^{\rho t}$$
and the encoded output translation tables:
$$\{[(h(b_{j,1}^0), h(b_{j,1}^1)), \ldots, (h(b_{j,\ell'}^0), h(b_{j,\ell'}^1))]\}_{j \in [\rho t]}.$$
If $h(b_{j,i}^0) = h(b_{j,i}^1)$ for any $1 \le i \le \ell', 1 \le j \le \rho t$, then P_2 aborts.
4. **Send cut-and-choose challenge:** P_2 sends P_1 the sets $J, J_1, \ldots, J_t$ along with values $\{\chi_j^1\}_{j \in J_1}, \ldots, \{\chi_j^t\}_{j \in J_t}$, and all the keys associated with its input wires in all circuits GC_j for $j \in J$. If the values received by P_1 are (1) incorrect, or (2) the sets $J_1, \ldots, J_t$ are not pairwise non-intersecting, or (3) the input keys associated with P_2's input wires in circuits GC_j are revealed incorrectly, or (4) there exists some $k \in [t]$ such that $|J_k| \ne \rho/2$, then it outputs $\perp$ and aborts. Circuits GC_j for $j \in J$ are called *check circuits* and circuits GC_j for $j \in J_k$ are called *evaluation circuits* in the *kth bucket*.
5. **Send garbled input values in the evaluation circuits:** For each $k \in [t]$: P_1 sends the input keys associated with input x_k for the evaluation circuits in the kth bucket: For each $j \in J_k$ and every wire $i \in [\ell]$, P_1 sends the value $k'_{i,j} = g^{a_i^{x_{k,i}} \cdot r_j}$ and P_2 sets $k_{i,j} = \mathsf{Ext}(k'_{i,j})$.
6. **Circuit evaluation:** For each $k \in [t]$, P_2 does the following:
   - For each $j \in J_k$ and every wire $i \in [\ell']$, P_2 computes $b'_{j,i}$ by evaluating GC_j. If P_2 receives exactly *one* valid output value per output wire, then let z_k denote this output. In this case, it chooses random values $b_0^k, b_1^k \in_R \{0,1\}^n$. If P_2 receives *two* valid outputs on any output wire then it sets $b_0^k = b'_{j_1,i}$ and $b_1^k = b'_{j_2,i}$, where $j_1, j_2 \in J_k$ denote the conflicting circuit indices. If P_2 receives *no* valid output values on any output wire, then P_2 aborts.
7. **Run secure computation to detect cheating:** For each $k \in [t]$, P_1 and P_2 do the following *in parallel*:

   P_1 defines a circuit with the values $\{b_{j,1}^0, b_{j,1}^1, \ldots, b_{j,\ell'}^0, b_{j,\ell'}^1\}_{j \in J_k}$ hardcoded. The circuit computes the following function:

   - P_1 inputs $x_k \in \{0,1\}^\ell$ and has no output.
   - P_2 inputs a pair of values b_0^k, b_1^k.
   - If there exists values $i \in [\ell']$ and $j_1, j_2 \in J_k$ such that $b_0^k = b_{j_1,i}^0$ and $b_1^k = b_{j_2,i}^1$, then P_2's output is x_k; otherwise it receives no output.

   P_1 and P_2 run the UC-secure protocol of Lindell and Pinkas [LP11] on this circuit (except for the proof of P_1's input values), as follows:

- P_1 inputs x_k; P_2 inputs b_0^k and b_1^k as computed in Step 6.
- The garbled circuits constructed by P_1 use the same a_i^0, a_i^1 values as were chosen in Step 1, and the parties use $3(s + \log t)$ copies of the circuit for the cut-and-choose.

If this computation results in an abort, then both parties halt.

8. **Check circuits for computing $f^{(t)}(x, y)$:**
    - For $j \in J$, P_1 sends r_j to P_2, and P_2 checks that these values are consistent with the pairs $\{(j, g^{r_j})\}_{j \in J}$ received in Step 3. If not, P_2 aborts.
    - For every $j \in J$, P_2 uses the $g^{a_i^0}, g^{a_i^1}$ values received in Step 3 and the r_j values received above to compute the keys for P_1's input wires as $k_{i,j}^0 = \mathsf{Ext}(g^{a_i^0 \cdot r_j}), k_{i,j}^1 = \mathsf{Ext}(g^{a_i^1 \cdot r_j})$. In addition, P_2 uses the keys obtained from $\mathcal{F}_{\text{mcot}}$ in Step 2 for its own input wires. P_2 verifies that GC_j is a correct garbling of C. If there exists a circuit for which this does not hold, then P_2 aborts.
9. **Verify consistency of P_1's input:** For each $k \in [t]$: Let $\widehat{J}_k$ be the set of check circuits used in the 2PC computation in Step 7 for the kth bucket, let $\widehat{r}_{j,k}$ be the value used in that computation, and let $\hat{k}_{i,j}$ be the analogous value of $k'_{i,j}$ in Step 5 received by P_2 in the computation in Step 7. For each $k \in [t]$, P_1 and P_2 do the following *in parallel*:
    - For every input wire $i \in [\ell']$, P_1 proves a zero-knowledge proof-of-knowledge that there exist some $\sigma_{k,i} \in \{0, 1\}$ such that for every $j \in J_k$ and every $j' \notin \hat{J}_k$, it holds that $k'_{i,j} = g^{a_i^{\sigma_{k,i}} \cdot r_j}$ and $\hat{k}_{i,j} = g^{a_i^{\sigma_{k,i}} \cdot \hat{r}_{j',k}}$. If any of the t proofs fail, then P_2 aborts.
10. **Output evaluation:** For each $k \in [t]$, P_2 does the following:
    - If P_2 received no inconsistent outputs in Step 6, then it uses the encoded translation tables to decode the outputs it received, and sets z_k to that value. If P_2 received inconsistent output, then let x_k be the output that P_2 received from the circuit in Step 7. Let $z_k = f(x_k, y_k)$ be the output in this case.

P_2 outputs $z = (z_1, \ldots, z_t)$ and terminates.

We prove the following theorem in Appendix B.

Theorem 2. *Let s (resp., n) be the statistical (resp., computational) security parameter. If the decisional Diffie-Hellman assumption holds in $(\mathbb{G}, g, q)$, h is a one-way function, and the underlying circuit garbling procedure is secure, then for all $t = \mathsf{poly}(n)$, the protocol described above securely computes $f^{(t)}$ in the presence of a malicious adversary with error at most $2^{-s} + \mu(n)$ for some negligible function $\mu(\cdot)$.*

2.2.1 Optimizing the Circuit in Step 7

We use an optimization inspired by Lindell [Lin13] to construct an alternate circuit that minimizes the number of non-XOR gates. Specifically, Lindell [Lin13] shows how to efficiently construct a garbled circuit that checks if a given n-bit string is contained in a set S of size m. The garbled circuit has the property that it only requires ℓ (where ℓ equals the length of each party's input) non-XOR gates, and thus can be essentially computed for free using, e.g., the free-XOR technique [KS08]. This optimization relies on the fact that to take an n-wise AND of two n-bit strings, it suffices to encrypt the output 1-key with the 1-keys on the input wires. Therefore, to compare two n bit strings, we first XOR the two strings bit-by-bit, take the NOT of these bits, and finally output the n-wise AND of the resulting bits using the trick described above. Next, to check that a n-bit string equals any of the m strings in S, we need to evaluate the m-wise OR of each of these comparisons. Instead of using $m-1$ OR gates, we can set the 1-key on all of the output wires from the n-wise ANDs above to be the 1-key on the output wire of the OR. Since the XOR and NOT gates can be evaluated for free [KS08], it follows that the above circuit can essentially be securely evaluated for free.[5]

We now adapt these optimizations to our setting, while still minimizing the number of non-XOR gates. For string b and set S, we use the notation $b \overset{?}{\in} S$ to denote a boolean expression that evaluates to 1 iff $b \in S$. In our protocol we require a circuit that takes, in addition to an ℓ-bit string u (representing P_1's actual input), a pair of n-bit strings, say b_0, b_1, and two sets S_0, S_1 of n-bit strings, each set of size $m = \rho\ell'/2$, and outputs u iff $((b_0 \overset{?}{\in} S_0) \wedge (b_1 \overset{?}{\in} S_1)) \vee ((b_0 \overset{?}{\in} S_1) \wedge (b_1 \overset{?}{\in} S_0)) = 1$, i.e., an additional cost of 3 non-XOR gates. Alternatively, we may instead evaluate the expression $b_0 \oplus b_1 \overset{?}{\in} S$, where $S = \{b \oplus b' \ : \ b \in S_0, b' \in S_1\}$. (Note that a cheating P_2 can guess a value in S only with negligible probability.) This has the additional advantage of reducing P_2's input length from $2n$ to n (and the resulting gains from performing a lesser number of cut-and-choose oblivious transfers). In summary, it is possible to design the circuit in Step 7 using *exactly* ℓ non-XOR gates (i.e., ℓ AND gates to select the length-ℓ P_1 input depending on whether the relevant conditions are satisfied). It follows from the protocol description that the total number of garbled gates sent in Step 7 is $3\ell(s + \log t)$ in each of the t executions.

3 The Sequential Execution Setting

We now consider the setting where the parties securely evaluate the same function f multiple times sequentially (see Appendix A.2 for the formal security definition). Let t denote the number of times the parties wish to evaluate f. Let P_1's (resp., P_2's) input in the kth execution be denoted by x_k (resp., y_k). Let $f^{[t]}$ denote the reactive functionality that computes f a total of t times sequentially.

The main difference between this setting and the parallel setting discussed in Section 2 is that in the sequential setting the parties may not know their inputs to all executions at the start of the protocol. In particular, inputs may depend on outputs from previous executions. Thus, the parallel execution protocol does not immediately carry over to the sequential setting. To see why, observe for instance that $\mathcal{F}_{\text{mcot}}$ requires P_2 to submit all of its inputs at once[6]. This is not possible since in the sequential setting we cannot assume that P_2 has all its inputs at the beginning of the protocol. Instead, we take a different route; namely, we use the "XOR-tree" approach [LP07, Woo07] to protect against the so-called "selective failure attack" [KS06, MF06, sS11]. (In the parallel execution setting, this attack was implicitly avoided due to the use of $\mathcal{F}_{\text{mcot}}$.) In this approach, the circuit C to be evaluated is first modified into an equivalent circuit C_{XT} (to include an "XOR-tree" for P_2's inputs). Then, P_1 sends commitments to input keys corresponding to P_2's input wires in C_{XT}. The corresponding decommitments are revealed to P_2 via a standard one-out-of-two

---

[5] Note that in order to prove security of the "free-XOR" technique in the standard model, one needs to make additional assumptions about the encryption used in Yao's garbled circuits [App13, CKKZ12, KS08].

[6] Standard oblivious transfer precomputation/"correction" techniques [Bea95] still apply to $\mathcal{F}_{\text{mcot}}$ as well; however, it is not clear how to "correct" $\mathcal{F}_{\text{mcot}}$ correlations in a way suitable for the sequential setting.

oblivious transfer. In order to prevent P_2 from using different inputs across evaluation circuits within the same bucket, P_1 batches together the decommitments corresponding to a particular input wire across all evaluation circuits in a given bucket. Note that herein lies an opportunity for a malicious P_1 to force P_2 to abort the protocol depending on its input. (This can be done for instance by sending incorrect decommitments for say only the 0-key on a particular wire.) However, the modified circuit C_{XT} is such that the success of any such selective OT attack is statistically independent of P_2's actual input value. Therefore, if an honest P_2 receives an invalid decommitment and is unable to decrypt the evaluation circuit, then it simply aborts knowing that its privacy is not compromised. Finally, we note that since we use one-out-of-two oblivious transfer (as opposed to $\mathcal{F}_{\mathrm{mcot}}$), we can leverage oblivious transfer extension techniques [IKNP03, IPS08, NNOB12] to obtain better efficiency.

We stress that the oblivious transfer step happens *after* P_1 sends all the GCs to P_2. This is because P_2's inputs to all t executions are not available at the beginning of the protocol. Further, P_2's inputs may depend on previous outputs, which can be obtained only by decrypting evaluation circuits, i.e., after the evaluation bucket for the current execution is fully determined. Note that our cut-and-choose technique guarantees that there is at least one good evaluation circuit in every bucket under the assumption that P_1 has already committed to all its (good and bad) garbled circuits before the check sets and the evaluation sets are determined. Unfortunately, the above ordering of the oblivious transfer step and the garbled circuit sending step now allows a malicious P_2 to choose its input as a function of the garbled circuits it receives. To counter this, we need to use *adaptively secure garbling schemes* [BHR12] instead of standard garbled circuits; adaptively secure garbling schemes can be constructed efficiently in the programmable random oracle model [BHR12]. Note that we do not need the use of adaptively secure garbling schemes for implementing the cheating-punishment phase. Indeed, all the inputs for that subprotocol are known before the phase begins, and therefore, the oblivious transfer step can be carried out before P_1 sends its garbled circuits for that phase.

Formal description. We now proceed to the formal description of the protocol.

Auxiliary Input: A statistical security parameter s, a computational security parameter n, the description of a circuit C where $C(x, y) = f(x, y)$, the number of evaluations t of the function f, and $(\mathbb{G}, q, g)$ where $\mathbb{G}$ is a cyclic group with generator g and prime order q, where q is of length n. Let $\mathsf{Ext} : \mathbb{G} \to \{0,1\}^n$ be a randomness extractor known to both parties. In the following, ρ is the replication factor implicitly defined by parameters t and s as being the smallest $u \in \mathbb{N}$ such that for all $m \in \{u/2, \ldots, ut/2\}$ it holds that $t \cdot \binom{ut-m}{ut/2}\binom{m}{u/2} \Big/ \binom{ut}{ut/2}\binom{ut/2}{u/2} \leq 2^{-s}$. If no such u exists or if $\rho \geq s + \log t$, then the parties abort this protocol and instead run the protocol of Lindell [Lin13] for function f a total of t times sequentially.

Additional Notation: Let ℓ, ℓ' denote the length of each input and the final output $f(x, y)$, respectively, and let C_{XT} denote the circuit C enhanced with the XOR-tree.

Offline Phase:

1. INPUT KEY CHOICE AND CIRCUIT PREPARATION:
    - P_1 chooses random values $a_1^0, a_1^1, \ldots, a_\ell^0, a_\ell^1, r_1, \ldots, r_{\rho t} \in_R \mathbb{Z}_q$ and $(b_{1,1}^0, b_{1,1}^1, \ldots, b_{1,\ell'}^0, b_{1,\ell'}^1), \ldots, (b_{\rho t,1}^0, b_{\rho t,1}^1, \ldots, b_{\rho t,\ell'}^0, b_{\rho t,\ell'}^1) \in_R \{0,1\}^{n\ell'}$ such that for every $c_1, c_2 \in \{0,1\}$, $j_1, j_2 \in [\rho t]$, $i_1, i_2 \in [\ell']$ it holds that $b_{j_1,i_1}^{c_1} = b_{j_2,i_2}^{c_2}$ iff $i_1 = i_2$ and $j_1 = j_2$ and $c_1 = c_2$.
    - Let $w_1, \ldots, w_\ell$ be the input wires corresponding to P_1's input in C_{XT}, let $w_{i,j}$ denote the ith input wire in the jth garbled circuit, and let $k_{i,j}^b$ denote the key associated with bit b on wire $w_{i,j}$. P_1 sets $k_{i,j}^b$ as follows:

$$k^0_{i,j} = \mathsf{Ext}(g^{a^0_i \cdot r_j}) \quad \text{and} \quad k^1_{i,j} = \mathsf{Ext}(g^{a^1_i \cdot r_j}).$$

- Let $\widetilde{w}_1, \ldots, \widetilde{w}_{\ell_{\mathrm{XT}}}$ be the input wires corresponding to P_2's input in C_{XT}, and denote by $\widetilde{w}_{i,j}$ the instance of wire $\widetilde{w}_i$ in the jth garbled circuit, and $\widetilde{k}^b_{i,j}$ the key associated with bit b on wire $\widetilde{w}_{i,j}$. Then, P_1 picks the keys for P_2's input wires uniformly at random, and computes (standard) commitments
$$e^0_{i,j} = \mathsf{com}(\widetilde{k}^0_{i,j}) \quad \text{and} \quad e^1_{i,j} = \mathsf{com}(\widetilde{k}^1_{i,j}).$$
Let $d^0_{i,j}$ and $d^1_{i,j}$ denote the corresponding decommitments.
- Let $w'_1, \ldots, w'_{\ell'}$ denote the output wires in C_{XT}. The keys for wire w'_i in the jth garbled circuit are set as $b^0_{j,i}$, $b^1_{j,i}$.
- P_1 constructs ρt independent *adaptively secure* garblings of circuit C_{XT}, denoted $GC_1, \ldots, GC_{\rho t}$, using random keys except for wires $w_1, \ldots, w_\ell$ and $w'_1, \ldots, w'_m$, where the keys are as above.

2. Send circuits and commitments: P_1 sends P_2 the garbled circuits, the "seed" for the randomness extractor Ext, the commitments to the garbled values associated with P_1's input wires:
$$\{(i, 0, g^{a^0_i}), (i, 1, g^{a^1_i})\}_{i \in [\ell]} \quad \text{and} \quad \{(j, g^{r_j})\}_{j=1}^{\rho t},$$
the encoded output translation tables:
$$\{[(h(b^0_{j,1}), h(b^1_{j,1})), \ldots, (h(b^0_{j,\ell'}), h(b^1_{j,\ell'}))]\}_{j \in [\rho t]},$$
and the commitments to the garbled values associated with P_2's input wires:
$$\{e^0_{i,j}, e^1_{i,j}\}_{i \in [\ell_{\mathrm{XT}}], j \in [\rho t]}.$$
If $h(b^0_{j,i}) = h(b^1_{j,i})$ for any $1 \leq i \leq \ell', 1 \leq j \leq \rho t$, then P_2 aborts.
3. Cut-and-choose challenge: P_1 and P_2 run a secure coin-tossing protocol to compute a set $J \subseteq [\rho t]$ such that $|J| = \rho t/2$. Circuits GC_j for $j \in J$ are called *check-circuits.*
4. Check circuits for computing $f^{[t]}(x, y)$:
   - 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 $\{(j, g^{r_j})\}_{j \in J}$ received in Step 3. If not, P_2 aborts.
   - Send all decommitments for P_2's input wires in check circuits: For every check-circuit GC_j, party P_1 sends the decommitments $\{d^0_{i,j}, d^1_{i,j}\}_{i \in [\ell_{\mathrm{XT}}]}$ for commitments $\{e^0_{i,j}, e^1_{i,j}\}_{i \in [\ell_{\mathrm{XT}}]}$, and P_2 checks that these are valid decommitments, and computes the corresponding keys $\{\widetilde{k}^0_{i,j}, \widetilde{k}^1_{i,j}\}_{i \in [\ell_{\mathrm{XT}}]}$. If not, P_2 aborts.
   - Correctness of check circuits: For every $j \in J$, P_2 uses the $g^{a^0_i}, g^{a^1_i}$ values received in Step 3 and the r_j values received in Step 4 to compute the values $k^0_{i,j} = \mathsf{Ext}(g^{a^0_i \cdot r_j}), k^1_{i,j} = \mathsf{Ext}(g^{a^1_i \cdot r_j})$ associated with P_1's input. Given all the garbled values for all input wires in GC_j, i.e., $\{k^0_{i,j}, k^1_{i,j}\}_{i \in [\ell_{\mathrm{XT}}]}$ and $\{\widetilde{k}^0_{i,j}, \widetilde{k}^1_{i,j}\}_{i \in [\ell_{\mathrm{XT}}]}$, party P_2 decrypts the circuit and verifies that it is a garbling of C_{XT}, using the encoded translation tables for the output values. If there exists a circuit for which this does not hold, then P_2 aborts.

On-line Phase: For each $1 \leq k \leq t$ execute the following sequentially:

1. Receive inputs: P_1 and P_2 obtain inputs x_k and y_k, respectively. Using additional randomness, P_2 transforms its input y_k for circuit C into an equivalent input $\widetilde{y}_k$ for circuit C_{XT}.
2. Second-stage cut-and-choose challenge: P_2 picks $J_k \subseteq [\rho t] \setminus J$ of size $\rho/2$ such that $J_1, \ldots, J_k$ are pairwise non-intersecting. P_2 sends J_k to P_1, who aborts the protocol if $|J_k| \neq \rho/2$ or J_k intersects with a previously sent subset. We call J_k the *kth evaluation bucket.*
3. Oblivious transfers: For each $i \in [\ell_{\mathrm{XT}}]$, party P_1 prepares $D^0_{i,k} = \{d^0_{i,j}\}_{j \in J_k}$, and $D^1_{i,k} = \{d^1_{i,j}\}_{j \in J_k}$. P_1 and P_2 then engage in ℓ_{XT} *parallel* invocations of $\mathcal{F}_{\mathrm{OT}}$ where in the ith invocation:
   - Acting as sender, P_1 inputs $(D^0_{i,k}, D^1_{i,k})$.
   - Acting as receiver, P_2 inputs $\widetilde{y}_{k,i}$, and receives $D^{\widetilde{y}_{k,i}}_{i,k} = \{d^{\widetilde{y}_{k,i}}_{i,j}\}_{j \in J_k}$.

If there exists $j \in J_k$ and $i \in [\ell_{\mathrm{XT}}]$ such that $d_{i,j}^{\widetilde{y_{k,i}}}$ is not a valid decommitment to $e_{i,j}^{\widetilde{y_{k,i}}}$, then P_2 aborts and outputs $\perp$. Else, P_2 computes the keys $\{\widetilde{k}_{i,j}^{\widetilde{y_{k,i}}}\}_{i \in [\ell_{\mathrm{XT}}], j \in J_k}$ corresponding to the decommitments it received. Let $\widetilde{k}_{i,j} = \widetilde{k}_{i,j}^{\widetilde{y_{k,i}}}$

4. P_1 SENDS ITS GARBLED INPUT VALUES IN THE EVALUATION CIRCUITS: P_1 sends the input keys associated with input x_k for the evaluation circuits in the kth bucket: For each $j \in J_k$ and every wire $i \in [\ell]$, P_1 sends the value $k'_{i,j} = g^{a_i^{x_{k,i}} \cdot r_j}$ and P_2 sets $k_{i,j} = \mathsf{Ext}(k'_{i,j})$.
5. CIRCUIT EVALUATION: P_2 uses the keys $\{k_{i,j}\}_{i \in [\ell]}$ associated with P_1's input and the keys $\{\widetilde{k}_{i,j}\}_{i \in [\ell_{\mathrm{XT}}]}$ associated with its own input to evaluate the circuits GC_j for $j \in J_k$ as follows:
   - For every wire $i \in [\ell']$, P_2 computes $b'_{j,i}$ by evaluating GC_j. If P_2 receives exactly *one* valid output value per output wire, then let z_k denote this output. In this case, it chooses $b_0^k, b_1^k \in_R \{0,1\}^n$. If P_2 receives *two* valid outputs on any output wire, then it sets $b_0^k = b'_{j_1,i}$ and $b_1^k = b'_{j_2,i}$, where $j_1, j_2 \in J_k$ are the conflicting circuit indices and $i \in [\ell']$ is the conflicting wire. If P_2 receives *no* valid output values on any output wire, then P_2 aborts.
6. RUN SECURE COMPUTATION TO DETECT CHEATING: P_1 defines a circuit C' as follows:
   - The circuits has the values $\{b_{j,1}^0, b_{j,1}^1, \ldots, b_{j,\ell'}^0, b_{j,\ell'}^1\}_{j \in J_k}$ hardcoded.
   - P_1 inputs $x_k \in \{0,1\}^\ell$ and has no output.
   - P_2 inputs a pair of values b_0^k, b_1^k.
   - If there exists values $i \in [\ell']$ and $j_1, j_2 \in J_k$ such that $b_0^k = b_{j_1,i}^0$ and $b_1^k = b_{j_2,i}^1$, then P_2's output is x_k; otherwise it receives no output.

   P_1 and P_2 run the protocol of [LP11] (except for the proof of P_1's input values) on C' as follows:
   - P_1 inputs x_k; P_2 inputs b_0^k and b_1^k as computed in Step 5.
   - The garbled circuits constructed by P_1 use the same a_i^0, a_i^1 values as were chosen in Step 1, and the parties use $3(s + \log t)$ copies of the garbled circuit for the cut-and-choose.

   If this computation results in an abort, then both parties halt at this point.
7. VERIFY CONSISTENCY OF P_1'S INPUT: Let $\widehat{J_k}$ be the set of check circuits in the 2PC computation in Step 6, and likewise, let $\widehat{r}_{j,k}$ be the value used in that computation as the equivalent of the r_j values in Step 1 of the offline-phase. Let $\widehat{k}_{i,j}$ be the analogous value of $k'_{i,j}$ in Step 4 received by P_2 in the 2PC computation in Step 6.

   P_1 and P_2 do the following *in parallel*: For every input wire $i \in [\ell]$, P_1 proves a zero-knowledge proof-of-knowledge that there exists some $\sigma_{k,i} \in \{0,1\}$ such that for every $j \in J_k$ and every $j' \notin \widehat{J_k}$, it holds that $k'_{i,j} = g^{a_i^{\sigma_{k,i}} \cdot r_j}$ and $\hat{k}_{i,j} = g^{a_i^{\sigma_{k,i}} \cdot \hat{r}_{j',k}}$. If any of the proofs fail, then P_2 aborts.
8. OUTPUT EVALUATION: If P_2 received no inconsistent outputs in Step 6, then it uses the encoded translation tables to decode the outputs it received, and sets z_k to that value. If P_2 *did* receive inconsistent output, then let x_k be the output that P_2 received from the 2PC computation in Step 7; P_2 sets $z_k = f(x_k, y_k)$. Finally, P_2 outputs z_k.

We prove the following theorem in Appendix C.

Theorem 3. *Let s (resp., n) be the statistical (resp., computational) security parameter. If the decisional Diffie-Hellman assumption holds in $(\mathbb{G}, g, q)$, h is a one-way function, and the circuit is garbled using an adaptively secure garbling scheme, then for all polynomial values of t, the protocol described above securely computes $f^{[t]}$ in the presence of a malicious adversary with error at most $2^{-s} + \mu(n)$ for some negligible function $\mu(\cdot)$.*

Acknowledgments

Work of Yan Huang and Jonathan Katz supported in part by NSF award #1111599. Work of Vladimir Kolesnikov supported in part by the Intelligence Advanced Research Project Activity (IARPA) via Department of Interior National Business Center (DoI/NBC) contract Number D11PC20194. Work of Ranjit Kumaresan supported by funding from the European Community's Seventh Framework Programme (FP7/2007–2013) under grant agreement number 259426. Work of Alex J. Malozemoff conducted with Government support through the National Defense Science and Engineering Graduate (NDSEG) Fellowship, 32 CFG 168a, awarded by DoD, Air Force Office of Scientific Research.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoI/NBC, or the U.S. Government.

References

[AIKW13] Benny Applebaum, Yuval Ishai, Eyal Kushilevitz, and Brent Waters. Encoding functions with constant online rate or how to compress garbled circuits keys. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 166–184, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Berlin, Germany. Full version available at `https://eprint.iacr.org/2012/693`.

[App13] Benny Applebaum. Garbling XOR gates "for free" in the standard model. In Amit Sahai, editor, *TCC 2013: 10th Theory of Cryptography Conference*, volume 7785 of *Lecture Notes in Computer Science*, pages 162–181, Tokyo, Japan, March 3–6, 2013. Springer, Berlin, Germany. Full version available at `https://eprint.iacr.org/2012/516`.

[Bea95] Donald Beaver. Precomputing oblivious transfer. In Don Coppersmith, editor, *Advances in Cryptology – CRYPTO'95*, volume 963 of *Lecture Notes in Computer Science*, pages 97–109, Santa Barbara, CA, USA, August 27–31, 1995. Springer, Berlin, Germany.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology – ASIACRYPT 2012*, volume 7658 of *Lecture Notes in Computer Science*, pages 134–153, Beijing, China, December 2–6, 2012. Springer, Berlin, Germany. Full version available at `https://eprint.iacr.org/2012/564`.

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

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, Nevada, USA, October 14–17, 2001. IEEE Computer Society Press. Full version available at `https://eprint.iacr.org/2000/067`.

[CJJ+13] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 353–373, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Berlin, Germany. Full version available at https://eprint.iacr.org/2013/169.

[CKKZ12] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. On the security of the "free-XOR" technique. In Ronald Cramer, editor, *TCC 2012: 9th Theory of Cryptography Conference*, volume 7194 of *Lecture Notes in Computer Science*, pages 39–53, Taormina, Sicily, Italy, March 19–21, 2012. Springer, Berlin, Germany. Full version available at https://eprint.iacr.org/2011/510.

[DO10] Ivan Damgård and Claudio Orlandi. Multiparty computation for dishonest majority: From passive to active security at low cost. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 558–576, Santa Barbara, CA, USA, August 15–19, 2010. Springer, Berlin, Germany. Full version available at https://eprint.iacr.org/2010/318.

[FJN+13] Tore Kasper Frederiksen, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Peter Sebastian Nordholt, and Claudio Orlandi. MiniLEGO: Efficient secure two-party computation from general assumptions. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 537–556, Athens, Greece, May 26–30, 2013. Springer, Berlin, Germany. Full version available at https://eprint.iacr.org/2013/155.

[GGP10] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 465–482, Santa Barbara, CA, USA, August 15–19, 2010. Springer, Berlin, Germany. Full version available at https://eprint.iacr.org/2009/547.

[GKK+12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12: 19th Conference on Computer and Communications Security*, pages 513–524, Raleigh, NC, USA, October 16–18, 2012. ACM Press. Full version available at https://eprint.iacr.org/2011/482.

[Gol04] Oded Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge University Press, Cambridge, UK, 2004.

[HKE13] Yan Huang, Jonathan Katz, and David Evans. Efficient secure two-party computation using symmetric cut-and-choose. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 18–35, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Berlin, Germany. Full version available at https://eprint.iacr.org/2013/081.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 145–161, Santa Barbara, CA, USA, August 17–21, 2003. Springer, Berlin, Germany.

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In David Wagner, editor, *Advances in Cryptology – CRYPTO 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 572–591, Santa Barbara, CA, USA, August 17–21, 2008. Springer, Berlin, Germany.

[JJK$^+$13] Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Outsourced symmetric private information retrieval. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13: 20th Conference on Computer and Communications Security*, pages 875–888, Berlin, Germany, November 4–8, 2013. ACM Press. Full version available at https://eprint.iacr.org/2013/720.

[KS06] Mehmet Kiraz and Berry Schoenmakers. A protocol issue for the malicious case of Yao's garbled-circuit construction. In *27th Symposium on Information Theory in the Benelux*, pages 283–290, June 2006.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008: 35th International Colloquium on Automata, Languages and Programming, Part II*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498, Reykjavik, Iceland, July 7–11, 2008. Springer, Berlin, Germany.

[Lin13] Yehuda Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 1–17, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Berlin, Germany. Full version available at https://eprint.iacr.org/2013/079.

[LP07] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In Moni Naor, editor, *Advances in Cryptology – EUROCRYPT 2007*, volume 4515 of *Lecture Notes in Computer Science*, pages 52–78, Barcelona, Spain, May 20–24, 2007. Springer, Berlin, Germany. Full version available at https://eprint.iacr.org/2008/049.

[LP09] Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, April 2009.

[LP11] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In Yuval Ishai, editor, *TCC 2011: 8th Theory of Cryptography Conference*, volume 6597 of *Lecture Notes in Computer Science*, pages 329–346, Providence, RI, USA, March 28–30, 2011. Springer, Berlin, Germany. Full version available at https://eprint.iacr.org/2010/284.

[LPS08] Yehuda Lindell, Benny Pinkas, and Nigel P. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In Rafail Ostrovsky, Roberto De Prisco, and Ivan Visconti, editors, *SCN 08: 6th International Conference on Security in Communication Networks*, volume 5229 of *Lecture Notes in Computer Science*, pages 2–20, Amalfi, Italy, September 10–12, 2008. Springer, Berlin, Germany.

[LR14] Yehuda Lindell and Ben Riva. Cut-and-choose Yao-based secure computation in the online/offline and batch settings. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part II*, volume 8617 of *Lecture Notes in Computer Science*, pages 476–494, Santa Barbara, CA, USA, August 17–21, 2014. Springer, Berlin, Germany. Full version available at `https://eprint.iacr.org/2014/667`.

[MF06] Payman Mohassel and Matthew Franklin. Efficiency tradeoffs for malicious two-party computation. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006: 9th International Conference on Theory and Practice of Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 458–473, New York, NY, USA, April 24–26, 2006. Springer, Berlin, Germany.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay — a secure two-party computation system. In Matt Blaze, editor, *13th USENIX Security Symposium*, San Diego, California, USA, August 9–13, 2004. USENIX Association.

[MR13] Payman Mohassel and Ben Riva. Garbled circuits checking garbled circuits: More efficient and secure two-party computation. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 36–53, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Berlin, Germany. Full version available at `https://eprint.iacr.org/2013/051`.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 681–700, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Berlin, Germany. Full version available at `https://eprint.iacr.org/2011/091`.

[NO09] Jesper Buus Nielsen and Claudio Orlandi. LEGO for two-party secure computation. In Omer Reingold, editor, *TCC 2009: 6th Theory of Cryptography Conference*, volume 5444 of *Lecture Notes in Computer Science*, pages 368–386. Springer, Berlin, Germany, March 15–17, 2009. Full version available at `https://eprint.iacr.org/2008/427`.

[Pin03] Benny Pinkas. Fair secure two-party computation. In Eli Biham, editor, *Advances in Cryptology – EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 87–105, Warsaw, Poland, May 4–8, 2003. Springer, Berlin, Germany.

[PVK$^+$14] Vasilis Pappas, Binh Vo, Fernando Krell, Seung Geol Choi, Vladimir Kolesnikov, Steven Bellovin, Angelos Keromytis, and Tal Malkin. Blind seer: A scalable private DBMS. In *2014 IEEE Symposium on Security and Privacy*, San Jose, California, USA, May 18–21, 2014. IEEE Computer Society Press.

[sS11] abhi shelat and Chih-Hao Shen. Two-output secure computation with malicious adversaries. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 386–405, Tallinn, Estonia, May 15–19, 2011. Springer, Berlin, Germany. Full version available at `https://eprint.iacr.org/2011/533`.

[Woo07] David P. Woodruff. Revisiting the efficiency of malicious two-party computation. In Moni Naor, editor, *Advances in Cryptology – EUROCRYPT 2007*, volume 4515 of *Lecture Notes in Computer Science*, pages 79–96, Barcelona, Spain, May 20–24, 2007. Springer, Berlin, Germany. Full version available at `https://eprint.iacr.org/2006/397`.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167, Toronto, Ontario, Canada, October 27–29, 1986. IEEE Computer Society Press.

A Security Definitions

Our security definitions allows one of the two participating parties to be corrupted by $\mathcal{A}$. We assume that there is an environment $\mathcal{Z}$ which interacts with $\mathcal{A}$ and the honest party in the way specified below. At the end of the execution, $\mathcal{Z}$ needs to distinguish between the case where $\mathcal{A}$ runs a protocol with the real honest party, and the case where $\mathcal{A}$ and the honest party invoke a trusted entity TP that computes the function $f = (f_1, f_2)$ on their behalf and returns their respective outputs. We assume, without loss of generality [LP07, LP09], that only the circuit evaluation receives output, and thus that $f_1(\cdot,\cdot) = \perp$. Loosely speaking, the protocol is secure if $\mathcal{Z}$'s advantage in distinguishing the two cases is negligible.

A.1 Definition of Security for Parallel Executions

Execution in the ideal model. In the ideal model, we have parties P_1 and P_2, and an adversary $\mathcal{A}$ who can corrupt one of the two parties. Let P_j for $j \in \{1,2\}$ denote the corrupted party and P_i for $i = \{1,2\}\backslash\{j\}$ denote the honest party. An ideal execution for the computation of the function f multiple times in parallel proceeds as follows.

Auxiliary Input: P_1 and P_2 hold 1^n, where n is the security parameter, and $\mathcal{Z}$ holds auxiliary input z. In addition, $\mathcal{Z}$ provides P_1 and P_2 a parameter t which denotes the number of times f is executed.

Inputs: P_1 and P_2 obtain inputs $\vec{x} = (x_1, \ldots, x_t) \in (\{0,1\}^\ell)^t$ and $\vec{y} = (y_1, \ldots, y_t) \in (\{0,1\}^\ell)^t$, respectively, from $\mathcal{Z}$.

Send inputs to TP: The honest party sends its input to the trusted party, TP. The corrupted party may send any value of its choice. Denote the pair of inputs sent to the trusted party by $(\vec{x'}, \vec{y'})$.

TP **sends outputs to $\mathcal{A}$:** If $\vec{x'}$ (resp., $\vec{y'}$) is not a valid input, TP sets $\vec{x'}$ (resp., $\vec{y'}$) to some default value. TP sends $f_j(\vec{x'}, \vec{y'})$ to $\mathcal{A}$ (recall that j denotes the index of the malicious party).

TP **sends outputs to honest party:** The adversary chooses whether to continue or abort; this is formalized by having $\mathcal{A}$ send either a `continue` or `abort` message to TP. In the former case, the trusted party sends $f_i(\vec{x'}, \vec{y'})$ to the honest party (recall that i denotes the index of the honest party). In the latter case, the trusted party sends the special symbol $\perp$ to the honest party.

Outputs: The corrupted party outputs nothing, and $\mathcal{A}$ outputs an arbitrary function of its view to $\mathcal{Z}$. The honest party outputs whatever it was sent by the trusted party to $\mathcal{Z}$. In the end, $\mathcal{Z}$ outputs a bit. We let $\textsc{Ideal}_{f,\mathcal{A},\mathcal{Z}(z)}(1^n)$ denote the output of $\mathcal{Z}$.

Execution in the real model. In the real model, we have parties P_1 and P_2 who execute a two-party protocol Π_f. The protocol Π_f has a parameter t initialized by $\mathcal{Z}$, which specifies the number of times f is evaluated in parallel. P_1 and P_2 obtain their inputs $\vec{x}$ and $\vec{y}$, respectively, from $\mathcal{Z}$, and obtain their outputs $\vec{z^1}$ and $\vec{z^2}$, respectively, by executing Π_f using their respective inputs. The honest party sends its output to $\mathcal{Z}$ and the adversary sends an arbitrary function of its view to $\mathcal{Z}$. Throughout the protocol execution, $\mathcal{A}$ obtains the inputs of the corrupted party and sends all messages on its behalf, whereas the honest party follows the instructions of Π_f. In the end, $\mathcal{Z}$ outputs a bit. We let $\textsc{Real}_{\Pi_f,\mathcal{A},\mathcal{Z}(z)}(1^n)$ denote the output of $\mathcal{Z}$.

Security as emulation of an ideal execution in the real model. Having defined the ideal and real models, we can now define security of a protocol. Loosely speaking, the definition asserts that a secure protocol (in the real model) emulates the ideal model (in which a trusted party exists). This is formulated as follows:

Definition 1. *Protocol Π_f is said to* securely compute $f^{(t)}$ *if for every* PPT *adversary $\mathcal{A}$ in the real model, there exists a* PPT *adversary SS in the ideal model such that for every non-uniform probabilistic polynomial-time environment $\mathcal{Z}$ that specifies the number of executions as t, it holds that*

$$\{\textsc{Ideal}_{f,\mathcal{S},\mathcal{Z}(z)}(1^n)\} \overset{c}{\approx} \{\textsc{Real}_{\Pi_f,\mathcal{A},\mathcal{Z}(z)}(1^n)\}$$

Remarks. The definition above is somewhat similar to security definitions in the Universal Composability (UC) framework [Can01] in the way we define security as the success probability of an environment $\mathcal{Z}$ that attempts to distinguish between the ideal world and the real world. In spite of this we stress that our definition is *not* as strong as the UC definition, as the latter allows $\mathcal{Z}$ to interact arbitrarily with $\mathcal{A}$ during the protocol execution. On the other hand, our security definition is somewhat closer to the security definition for parallel composition of protocols [Can00].

A.2 Definition of Security for Sequential Executions

Execution in the ideal model. In the ideal model, we have parties P_1 and P_2, and an adversary $\mathcal{A}$ who can corrupt one of the two parties. Let P_j for $j \in \{1, 2\}$ denote the corrupted party and P_i for $i = \{1, 2\}\backslash\{j\}$ denote the honest party. An ideal execution for the computation of the function f multiple times sequentially proceeds as follows.

Auxiliary Input: P_1 and P_2 hold 1^n, where n is the security parameter, and $\mathcal{Z}$ holds auxiliary input z. In addition, $\mathcal{Z}$ provides P_1 and P_2 a parameter t which denotes the number of times f is executed.

For $1 \leq k \leq t$:

Inputs: P_1 and P_2 obtain inputs $x_k \in \{0, 1\}^\ell$ and $y_k \in \{0, 1\}^\ell$, respectively, from $\mathcal{Z}$.

Send inputs to TP**:** The honest party sends its input to the trusted party, TP. The corrupted party, P_j, may send any value of its choice. Denote the pair of inputs sent to the trusted party by (x'_k, y'_k).

TP **sends outputs to** $\mathcal{A}$**:** If x'_k (resp., y'_k) is not a valid ℓ-bit input, TP sets x'_k (resp., y'_k) to some default value. TP sends $f_j(x'_k, y'_k)$ to $\mathcal{A}$.

TP **sends outputs to honest party:** The adversary chooses whether to continue or abort; this is formalized by having $\mathcal{A}$ send either a $\mathsf{continue}$ or abort message to TP. In the former case, the trusted party sends $f_i(x'_k, y'_k)$ to the honest party. In the latter case, the trusted party sends the special symbol $\perp$ to the honest party.

Outputs: The corrupted party outputs nothing, and $\mathcal{A}$ outputs an arbitrary function of its view to $\mathcal{Z}$. The honest party outputs whatever it was sent by the trusted party to $\mathcal{Z}$. If the honest party receives $\perp$ as output from the trusted party in iteration k, then it aborts the rest of the protocol.

At the end of t iterations, $\mathcal{Z}$ outputs a bit. We let $\textsc{Ideal}_{f,\mathcal{A},\mathcal{Z}(z)}(1^n)$ denote the output of $\mathcal{Z}$.

Execution in the real model. In the real model, we have parties P_1 and P_2 who execute a two-party protocol Π_f. The protocol Π_f has a parameter t initialized by $\mathcal{Z}$, which specifies the number of times f is evaluated. Protocol Π_f is *stateful* across its execution spanning t stages. In each stage, P_1 and P_2 obtain their inputs x_k respectively y_k from $\mathcal{Z}$, and obtain their outputs z^1_k respectively z^2_k by executing Π_f using their respective inputs. At the end of each stage, the honest party sends its output to $\mathcal{Z}$ and the adversary sends an arbitrary function of its view to $\mathcal{Z}$. Throughout the protocol execution, $\mathcal{A}$ obtains the inputs of the corrupted party and sends all messages on its behalf, whereas the honest party follows the instructions of Π_f. At the end of t stages of Π_f, $\mathcal{Z}$ outputs a bit. We let $\textsc{Real}_{\Pi_f,\mathcal{A},\mathcal{Z}(z)}(1^n)$ denote the output of $\mathcal{Z}$.

Security as emulation of an ideal execution in the real model. Having defined the ideal and real models, we can now define security of a protocol. Loosely speaking, the definition asserts that a secure protocol (in the real model) emulates the ideal model (in which a trusted party exists). This is formulated as follows.

Definition 2. *Protocol* Π_f *is said to* securely compute $f^{[t]}$ *if for every* PPT *adversary* $\mathcal{A}$ *in the real model, there exists a* PPT *adversary SS in the ideal model such that for every non-uniform probabilistic polynomial-time environment* $\mathcal{Z}$ *that specifies the number of executions as* t*, it holds that*

$$\{\textsc{Ideal}_{f,\mathcal{S},\mathcal{Z}(z)}(1^n)\} \overset{c}{\approx} \{\textsc{Real}_{\Pi_f,\mathcal{A},\mathcal{Z}(z)}(1^n)\}$$

Remarks. As in the parallel executions case, this definition differs from the definitions in the Universal Composability (UC) framework [Can01], since in our setting we restrict $\mathcal{Z}$ to interact with $\mathcal{A}$ only between stages of the protocol Π_f, but never within a stage.

B Proof in the Parallel Execution Setting

Theorem. *Let* s *(resp.,* n*) be the statistical (resp., computational) security parameter. If the decisional Diffie-Hellman assumption holds in* $(\mathbb{G}, g, q)$*,* h *is a one-way function, and the underlying circuit garbling procedure is secure [LP07], then for all* $t = \mathsf{poly}(n)$*, the protocol described in*

Section 2 securely computes $f^{(t)}$ in the presence of a malicious adversary with error at most $2^{-s} + \mu(n)$ for some negligible function $\mu(\cdot)$.

Proof. We prove security in a hybrid model where a trusted third party computes the batch single-choice multi-stage cut-and-choose oblivious transfer functionality in Step 2, the zero-knowledge proof-of-knowledge functionality in Step 9. We split the analysis into two cases depending on whether P_1 or P_2 is corrupted.

P_1 **is corrupted.** The intuition is that P_1 can cheat only if it can construct incorrect circuits. To do this, P_1 needs to construct a *small* enough number of incorrect circuits such that it will not get caught in the first cut-and-choose stage; however, it need also construct a *large* enough number such that one of the buckets contains all incorrect circuits. This is due to the fact that P_2 aborts if it finds an invalid check circuit, and learns P_1's input (and thus the correct output) if a given bucket contains at least one correctly constructed circuit. This implies that the number of corrupt circuits m constructed by a malicious P_1 must be such that $\rho/2 \leq m \leq \rho t/2$. We stress that m is fixed once P_1 sends the circuits in Step 3; that is, P_1 cannot further "corrupt" circuits after this step. Now observe that the probability with which m bad circuits escape detection in the first stage cut-and-choose is $\binom{\rho t - m}{\rho t/2} / \binom{\rho t}{\rho t/2}$. Conditioned on this event happening, the probability that a particular bucket contains all bad circuits is $\binom{m}{\rho/2} / \binom{\rho t/2}{\rho/2}$. Applying the union bound, we conclude that the probability that P_1 succeeds in cheating is bounded by

$$t \binom{\rho t - m}{\rho t/2} \binom{m}{\rho/2} \bigg/ \binom{\rho t}{\rho t/2} \binom{\rho t/2}{\rho/2}.$$

Since it is given that the maximum value of this expression is less than 2^{-s} for parameter ρ chosen in the protocol, we have that the probability of cheating is at most 2^{-s}. We now proceed to the formal proof.

Let $\mathcal{A}$ be an adversary controlling P_1 with input $x = (x_1, \ldots, x_t)$. Since P_1 receives no output, we need only show that the difference in probability that P_2 aborts in the real world versus the ideal world in negligible. We construct a simulator $\mathcal{S}$ with access to a trusted party computing $f^{(t)}$ as follows:

1. $\mathcal{S}$ acts as an honest P_2 would for the entire protocol execution, using input $y = (0^\ell, \ldots, 0^\ell)$ throughout.
2. For each $k \in [t]$, let $x_k = \sigma_{k,1}, \ldots, \sigma_{k,\ell}$ be P_1's witness to the zero-knowledge proof-of-knowledge in Step 9. $\mathcal{S}$ extracts these values through the ideal functionality interface.
3. If would P_2 abort at any point in the protocol, then $\mathcal{S}$ sends $\perp$ to the ideal functionality computing $f^{(t)}$. Otherwise, it sends $x = (x_1, \ldots, x_t)$ and receives back $z = f^{(t)}(x, y)$.

We now claim that the distributions from $\mathcal{A}$ interacting with P_2 in the real world versus $\mathcal{A}$ interacting with $\mathcal{S}$ in the ideal world are indistinguishable. To do so, we construct a set of hybrids, starting from the real execution and ending at the ideal execution, and show that each hybrid is indistinguishable from its neighbors. Here, we provide a sketch of these hybrids.

The first hybrid is simply the real world execution of the protocol. The next hybrid is equivalent, except that we extract P_1's input $x' = (x'_1, \ldots, x'_t)$ from the zero-knowledge proof-of-knowledge ideal functionality in Step 9. Instead of outputting P_2's output from the execution of the protocol, we instead pass x' to the trusted third party, and output $f^{(t)}(x', y)$.

These two hybrids differ if the output of P_2 differs from the output computed by the trusted third party. This happens if one of the evaluation buckets contains all maliciously constructed circuits. As was shown above, this happens with probability $< 2^{-s}$, and thus we conclude that these hybrids are indistinguishable.

The next hybrid is the same as the prior one, except that P_2 uses input $y = (0^\ell, \ldots, 0^\ell)$ throughout. Noting that P_2 only uses y as input to the $\mathcal{F}_{\text{mcot}}$ functionality in Step 2, we conclude that the two hybrids are perfectly indistinguishable.

As this last hybrid is the same as the simulator $\mathcal{S}$ given above, we conclude that the protocol is secure.

P_2 is corrupted. The intuition for security in the case that P_2 is corrupt is standard [LP11, Lin13], and thus we jump right to the proof.

Let $\mathcal{A}$ be an adversary controlling P_2 with input $y = (y_1, \ldots, y_t)$. We assume the existence of a simulator $\mathcal{S}'$ which constructs garbled circuits with fixed outputs which are indistinguishable from correctly garbled circuits. Such a simulator is known to exist [LP09, LP07]. Also, we use the simulator from [LP11], which we denote as $\mathcal{S}''$.

We construct a simulator $\mathcal{S}$ with access to a trusted party computing $f^{(t)}$ as follows:

1. $\mathcal{S}$ extracts P_2's input y and sets $J, J_1, \ldots, J_t$ from the call to $\mathcal{F}_{\text{mcot}}$.

2. $\mathcal{S}$ sends y to the trusted party, receiving back $z = (z_1, \ldots, z_t) = f^{(t)}(x, y)$.

3. For every $j \in J$, $\mathcal{S}$ constructs a valid garbled circuit. For every $k \in [t]$ and for every $j \in J_k$, $\mathcal{S}$ uses $\mathcal{S}'$ to construct a garbled circuit that outputs the fixed string z_k irrespective of the input.

4. $\mathcal{S}$ uses $\mathcal{S}''$ to simulate the 2PC protocol in Step 7.

5. Otherwise, $\mathcal{S}$ runs the protocol as an honest P_1 would.

6. Upon protocol termination, $\mathcal{S}$outputs whatever $\mathcal{A}$ outputs and halts.

We now claim that the distributions from $\mathcal{A}$ interacting with P_2 in the real world versus $\mathcal{A}$ interacting with $\mathcal{S}$ in the ideal world are indistinguishable. To do so, we again construct a set of hybrids, starting from the real world and ending at the ideal world, and show that each hybrid is indistinguishable from its neighbors. Here, we provide a sketch of these hybrids.

The first hybrid is simply the real world execution of the protocol. The following hybrid is equivalent, except as follows. We extract $\mathcal{A}$'s input $y' = (y'_1, \ldots, y'_t)$ and the sets $J_1, \ldots, J_t$ from the call to $\mathcal{F}_{\text{mcot}}$ in Step 2. Let $z = (z_1, \ldots, z_t) = f^{(t)}(x, y')$ be the output of the trusted third party. Let $J = [\rho t] \setminus \cup_k J_k$. For $j \in J$, we construct correctly garbled circuits, and for $j \in J_k$ for all k, we use $\mathcal{S}'$ to construct a circuit which always outputs z_k.

We claim that these two hybrids are indistinguishable. Note that $\mathcal{A}$ can distinguish if either he can open one of the simulated garbled circuits, or he can evaluate a simulated garbled circuit in bucket k on something other than y'_k. The only way for one of the above situations to occur is if $\mathcal{A}$ can guess input keys for garbled circuits in Step 4. Clearly, this happens with negligible probability.

The following hybrid is the same as the above hybrid, except we replace the real 2PC execution in Step 7 with a simulated execution using $\mathcal{S}''$. Due to the security of $\mathcal{S}''$ (as was shown in [LP11]), we conclude that these two hybrids are indistinguishable.

Finally, the last hybrid is the same as the above one, except that we use $x = (0^\ell, \ldots, 0^\ell)$ as P_1's input throughout. Note that this affects Step 5, where $\mathcal{A}$ receives P_1's inputs $g^{a_i^{x_{k,i}} \cdot r_j}$; however, by the decisional Diffie-Hellman assumption, $\mathcal{A}$ cannot extract $a_i^{x_{k,i}}$ from this expression, and thus cannot deduce that P_1's input is x as defined above. Thus, the two hybrids are indistinguishable.

As this last hybrid is the same the simulator $\mathcal{S}$ given above, we conclude that the protocol is secure. □

C Proof in the Sequential Execution Setting

Theorem. *Let s (resp., n) be the statistical (resp., computational) security parameter. If the decisional Diffie-Hellman assumption holds in $(\mathbb{G}, g, q)$, h is a one-way function, and the circuit is garbled using an adaptively secure garbling scheme, then for all polynomial values of t, the protocol described in Section 3 securely computes $f^{[t]}$ in the presence of a malicious adversary with error at most $2^{-s} + \mu(n)$ for some negligible function $\mu(\cdot)$.*

Proof. The proof is similar to the parallel case. The two major changes are the use of the XOR-tree to avoid the selective failure attack (in place of cut-and-choose oblivious transfer), and the use of adaptively secure garbling.

P_1 is corrupted. The intuition here is the same as for the parallel execution setting, and thus we jump straight to the simulator. Let $\mathcal{A}$ be an adversary controlling P_1. We construct a simulator $\mathcal{S}$ with access to a trusted party computing f as follows:

1. $\mathcal{S}$ acts exactly as an honest P_2 would for the entire off-line phase of the protocol.

2. Likewise, $\mathcal{S}$ acts exactly as an honest P_2 would for each of the t executions of the on-line phase, using $y = 0^\ell$ as its input for each iteration, except as follows:
   - $\mathcal{S}$ extracts P_1's input x_k from the zero-knowledge proof-of-knowledge in Step 7 of the on-line phase, and sends it to the trusted third party, receiving back the output z_k.
   - If an honest P_2 would abort at any step in the protocol, $\mathcal{S}$ sends $\perp$ to the trusted third party; otherwise, $\mathcal{S}$ sends x and receives $f(x, y)$ in return.

Now, as was shown in [Lin13] and in Appendix B, the probability that the output in the ideal and real world are identical across all t executions is exactly one minus the probability that a given bucket contains all maliciously constructed circuits. As was shown in Appendix B, this probability is $< 2^{-s}$.

P_2 is corrupted. Let $\mathcal{A}$ be an adversary controlling P_2 . Again, the intuition here is similar to the parallel execution setting. However, we cannot use the standard simulator for garbled circuits anymore, as we need adaptively secure garbled circuits. Instead, we make use of an adaptively secure garbling simulator [BHR12]. In particular, we need to use a simulator for the $\mathsf{all2}$ definition of security, which provides fine-grained adaptive security in terms of privacy, obliviousness, and authenticity. Bellare, Hoang, and Rogaway [BHR12] show the existence of such a simulator, which we denote by $\mathcal{S}' = (\mathcal{S}'_1, \mathcal{S}'_2)$, in the Random Oracle Model. This simulator has two "stages": $\mathcal{S}'_1$

constructs a simulated garbled circuit, and $\mathcal{S}'_2$, given input y, constructs simulated input keys for input y. We also utilize the simulator for [LP11], which we denote by $\mathcal{S}''$. We construct a simulator $\mathcal{S}$ with access to a trusted party computing f as follows:

1. $\mathcal{S}$ the acts exactly as an honest P_1 would for the entire off-line phase of the protocol, except for the following:
   - Prior to Step 1, $\mathcal{S}$ chooses a random string r of size ρt such that half of the bits in r are set to one. Then, for each $i \in [\rho t]$, if $r_i = 1$ then $\mathcal{S}$ constructs a correctly garbled circuit; if $r_1 = 0$ then $\mathcal{S}$ uses $\mathcal{S}'_1$ to construct a simulated *adaptively-secure* garbled circuit. In Step 1(c), for those circuits with $r_i = 1$, $\mathcal{S}$ uses the inputs keys generated by $\mathcal{S}'_1$, otherwise $\mathcal{S}$ constructs the input keys as specified in the protocol.
   - In Step 3, $\mathcal{S}$ sets the secure coin-tossing protocol to output r.
2. In the on-line phase, $\mathcal{S}$ acts as follows:
   - $\mathcal{S}$ uses $x = 0^\ell$ as its input for each iteration.
   - In Step 3, $\mathcal{S}$ receives P_2's input $\widetilde{y}_k$ from the OT functionality. It then runs $\mathcal{S}'_2$ on $\widetilde{y}_k$, receiving back encoded values $(D^0_{i,k}, D^1_{i,k})$, and sends $D^{\widetilde{y}_{k,i}}_{i,k}$ to P_2 through the OT functionality interface.
   - In Step 6, $\mathcal{S}$ uses $\mathcal{S}''$ to simulate the execution of circuit C'.
   - Otherwise, $\mathcal{S}$ runs exactly as an honest P_1 would.

We now claim that the distributions from $\mathcal{A}$ interacting with P_2 in the real world versus $\mathcal{A}$ interacting with $\mathcal{S}$ in the ideal world are indistinguishable by $\mathcal{Z}$. The argument is nearly equivalent to that shown in the parallel execution case, and is thus omitted. $\square$

Changelog

- Version 1.0 (February 3, 2015): First release. This is the full version of the proceedings version published at CRYPTO 2014.