Cut-and-Choose Based Two-Party Computation in the Online/Offline and Batch Settings[*]

Yehuda Lindell[†] Ben Riva[†]

August 26, 2014

Abstract

Protocols for secure two-party computation enable a pair of mistrusting parties to compute a joint function of their private inputs without revealing anything but the output. One of the fundamental techniques for obtaining secure computation is that of Yao's garbled circuits. In the setting of malicious adversaries, where the corrupted party can follow any arbitrary (polynomial-time) strategy in an attempt to breach security, the cut-and-choose technique is used to ensure that the garbled circuit is constructed correctly. The cost of this technique is the construction and transmission of multiple circuits; specifically, s garbled circuits are used in order to obtain a maximum cheating probability of 2^{-s}.

In this paper, we show how to reduce the amortized cost of cut-and-choose based secure two-party computation to $\mathcal{O}\left(\frac{s}{\log N}\right)$ garbled circuits when N secure computations are run. We use this method to construct a secure protocol in the batch setting. Next, we show how the cut-and-choose method on garbled circuits can be used in an online/offline setting in order to obtain a very fast online phase with very few exponentiations, and we apply our amortization method to this setting as well. Our online/offline protocols are competitive with the TinyOT and SPDZ protocols due to the minimal interaction in the online phase (previous protocols require only information-theoretic operations in the online phase and are therefore very efficient; however, they also require many rounds of communication which increases latency). Although $\mathcal{O}(\frac{s}{\log N})$ may seem to be a mild efficiency improvement asymptotically, it is a *dramatic improvement* for concrete parameters since s is a statistical security parameter and so is typically small. Specifically, instead of 40 circuits to obtain an error of 2^{-40}, when running 2^{10} executions we need only 7.06 circuits on average per secure computation, and when running 2^{20} executions this is reduced to an average of just 4.08. In addition, in the online/offline setting, the online phase per secure computation consists of evaluating only 6 garbled circuits for 2^{10} executions and 4 garbled circuits for 2^{20} executions (plus some small additional overhead). In practice, when using fast implementations (like the JustGarble framework of Bellare et al.), the resulting protocol is remarkably fast.

We present a number of variants of our protocols with different assumptions and efficiency levels. Our basic protocols rely on the DDH assumption alone, while our most efficient variants are proven secure in the random-oracle model. Interestingly, the variant in the random-oracle model of our protocol for the online/offline setting has online communication that is independent of the size of the circuit in use. None of the previous protocols in the online/offline setting achieves this property, which is very significant since communication is usually a dominant cost in practice.

[*]An extended abstract of this work appeared at *CRYPTO 2014*. This work was funded by the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement n. 239868 (LAST), and by the European Union's Seventh Framework Program (FP7/2007-2013) under grant agreement n. 609611 (PRACTICE).

[†]Department of Computer Science, Bar-Ilan University, Israel. `lindell@biu.ac.il, benr.mail@gmail.com`.

Contents

1 Introduction

1.1 Background

In the setting of secure two-party computation, a pair of parties with private inputs wish to compute a joint function of their inputs. The computation should maintain privacy (meaning that the legitimate output but nothing else is revealed), correctness (meaning that the output is correctly computed), and more. These properties should be maintained even if one of the parties is corrupted. The feasibility of secure computation was demonstrated in the 1980s, where it was shown that any probabilistic polynomial-time functionality can be securely computed [Yao86, GMW87].

The two main adversary models that have been considered in the literature are *semi-honest* and *malicious*. A semi-honest adversary follows the protocol specification but attempts to learn more than allowed by inspecting the transcript. In contrast, a malicious adversary can follow any arbitrary (probabilistic polynomial-time) strategy in an attempt to break the security guarantees of the protocol. On the one hand, the security guarantees in the semi-honest case are rather weak, but there exist extraordinarily efficient protocols [HEKM11, BHR12b, ALSZ13]. On the other hand, the security guarantees in the malicious case are very strong, but they come at a significant computational cost.

The goal of constructing efficient secure two-party (2PC) computation protocols in the presence of malicious adversaries has been an active area of research in the recent years. [JS07, NO09] construct 2PC protocols with a small number of exponentiations per gate of the circuit, which is quite inefficient in practice. [IPS08, IKO$^+$11] construct 2PC protocols based on the MPC-in-the-head approach which (asymptotically) requires only a small number of symmetric-key operations per gate of the circuit, though no implementation has been presented yet to clarify the concrete complexity of this approach in practice. [NNOB12, FJN$^+$13] construct 2PC protocols in the random-oracle model with (amortized) $\mathcal{O}(s/\log(|C|))$ symmetric-key operations per gate of the circuit, where s is a security parameter and $C(\cdot)$ is a boolean circuit that computes the function of interest. [DPSZ12, DKL$^+$13] construct secure multi-party computation protocols with security against *all-but-one* corrupted parties, and thus, could be used in the two-party setting as well. These protocols use somewhat homomorphic encryption. The protocols of [NNOB12, DPSZ12, DKL$^+$13] all require a number of rounds of communication that is in the order of the depth of the circuit being computed.[1] Thus, their performance is limited in the case of deep circuits, and when parties are geographically far and so communication latency is significant.

A different approach that has received a lot of attention is based on applying the *cut-and-choose* technique to Yao's garbled-circuit protocol. In this technique, one of the parties prepares many garbled circuits, and the other asks to open a random subset of them in order to verify that they are correct; the parties then evaluate the remaining, unchecked circuits. This forces the party generating the garbled circuits to make most of them correct, or it will be caught cheating (solving perhaps the biggest problem in applying Yao's protocol to the malicious setting, which is that an incorrect garbled circuit that computes the wrong function cannot be distinguished from a correct garbled circuit). [MF06, LP07, LP11, SS11, Lin13, MR13, SS13] present different 2PC protocols based on this approach, and several implementations have been presented to study the concrete efficiency of it in practice (e.g.[PSSW09, SS11, KSS12, SS13]). *In this work we focus on the cut-and-choose approach.*

[1]The protocol of [FJN$^+$13] *is* constant round. However, its concrete efficiency has not been established.

Is it possible to go below s garbled circuits with 2^{-s} error? Until the recent work of [Lin13], protocols that use the cut-and-choose technique required approximately $3s$ garbled circuits to obtain a bound of 2^{-s} on the cheating probability by the adversary. Recently, [Lin13] showed that by executing another light 2PC, the number of garbled circuits can be reduced to s, which seems optimal given that 2^{-s} is the probability that a "cut" is as bad as possible (meaning that all the check circuits are good and all the unchecked circuits are bad). The number of garbled circuits affects both computation time and communication. In most applications, when $|C|$ is large, sending s garbled circuits becomes the dominant overhead. (For example, [HMSG13] showed a prototype for garbling a circuit on GPUs, which generates more than 30 million gates per second. The communication size of this number of gates is about 15GB, and transferring 15GB of data most likely takes much more than a second.) Thus, further reducing the number of circuits is an important goal. *This goal is the focus of this paper.*

2PC with offline and online stages. In the online/offline setting, the parties try to push as much work as possible to an offline stage in which they do not know their inputs. Later, in the online stage, when they have their inputs, they use the results of the offline stage to run a very efficient online phase, possibly with much lower latency than their standard counterparts.

The protocols of [NNOB12, DPSZ12, DKL+13] are especially well suited to the online/offline setting, and have extremely efficient online stages.[2] However, these protocols require many rounds of interaction in the online stage (i.e., $\mathcal{O}(\text{depth}(C))$ rounds). They therefore become considerably slower for deep circuits and over high-latency networks.

Previous cut-and-choose based protocols work only in the regular setting, in which both parties run the protocol from beginning to its end. Note that cut-and-choose based 2PC protocols are constant-round, which is another reason for trying to apply them in the online/offline setting.

1.2 Our Contributions

As we have mentioned, the goal of this paper is to reduce the number of circuits in cut-and-choose on Yao's garbled circuits. We achieve this goal in the multiple-execution setting, where a pair of parties run many executions of the protocol. As we will see, this enables the parties to *amortize* the cost of the check-circuits over many executions.

Amortizing checks over multiple executions. In the single-execution setting, party P_1 constructs s circuits and party P_2 asks to open a random subset of them. If P_1 makes some of them incorrect and some correct, then it can always succeed in cheating if P_2 opens all of the good circuits and the remaining are all bad. Since this bad event can happen with probability 2^{-s}, this approach to cut-and-choose seems to have a limitation of s circuits for 2^{-s} error. However, consider now the case that the parties wish to run N executions. One possibility is to simply prepare $N \cdot s$ circuits and work as in the single execution case. Alternatively, P_1 can prepare $c \cdot N$ circuits (for some constant c); then P_1 can ask to open a subset of the circuits; finally, P_2 randomly assigns the remaining circuits to N small buckets of size B (where one bucket is used for every execution). The protocol that we use, which is based on [Lin13], has the property that P_1 can cheat only if there

[2] In fact, the protocols of [NNOB12, DPSZ12, DKL+13] allow the parties to choose the function also in the online stage. In this work we assume that the function is known in the offline stage, and it is only the inputs that are obtained later.

is a bucket in which *all* of the circuits are bad. The probability of this happening when not too many bad circuits are constructed by P_1 is very small, but if P_1 does construct many bad circuits then it will be caught even if a relatively small subset of circuits is checked.

This idea is very powerful and it enables us to obtain an extraordinary speedup over the single-execution case. Asymptotically, only $\mathcal{O}(\frac{s}{\log N})$ garbled circuits are needed per execution (on average). Concretely, if the parties wish to run $N = 1024$ executions and maintain an error of 2^{-40}, then it suffices to construct 7229 circuits, check 15% of them, and randomly map the remaining into buckets of size 6. The number of circuits per execution is thus reduced from 40 to 7.06, which is a considerable improvement. As the number of executions grows, the improvement is more significant. Specifically, for $N = 1,048,576$ and an error of 2^{-40}, it suffices to construct 4,279,903 circuits, check 2% of them, and randomly map the remaining into buckets of size 4. The number of circuits per execution is thus reduced to just 4.08, which is almost a tenfold improvement! Finally, we note that improvements are obtained even for small numbers of N; e.g., for $N = 10$, the number of circuits per execution is reduced to 20, which is half the cost.

The batch setting – parallel executions. In this setting, the parties run N executions in parallel. Formally, they compute the functionality $F(\vec{x}, \vec{y}) = (f(x_1, y_1), \ldots, f(x_N, y_N))$ where $\vec{x} = (x_1, \ldots, x_N)$ and $\vec{y} = (y_1, \ldots, y_N)$. We start with the protocol of [Lin13] and apply our amortized checking technique in order to use only $\mathcal{O}\left(\frac{s}{\log N}\right)$ garbled circuits per execution. However, the protocol of [Lin13] does not work in a setting where the circuits are constructed without knowing which circuits will be placed together in a single bucket. In Section 2.2 we describe the problems that arise and how we overcome them.

The online/offline setting. Next, we turn to the online/offline setting, with the aim of constructing an efficient 2PC protocol with a constant-round online stage and low latency. In order to achieve this, we show how to adapt the protocol of [Lin13] to the online/offline setting, and then use the amortized checking technique described above to significantly reduce the number of circuits needed. There are many issues that arise when trying to run cut-and-choose based protocols in the online/offline setting, mainly due to the fact that many of the techniques used to prevent cheating when cut-and-choose is used assume that the parties inputs are fixed even before the cut-and-choose takes place. In Section 2.3 we present a high-level description of our protocol, and our solutions to the problems that arise in this setting with cut-and-choose.

Our protocol achieves very high efficiency. First, the overall time (offline and online) is much lower than running a separate execution for every computation. Thus, we do not obtain a very fast online time at the expense of a very slow offline time. Rather, the overall protocol is highly efficient, and most of the work can be carried out in the offline phase. Second, our online phase requires very little communication, the evaluation of a small number of circuits, and little overhead. Concretely, when 1,000 executions are prepared in the offline phase, then the online phase requires evaluating only 5 circuits; in modern implementations like [BHR12b] and [HMSG13], this is extremely fast (with more executions, this is even further reduced).

Our basic protocol for the online/offline setting, presented in Section 7.1, is the first (efficient) 2PC protocol in that setting with a constant-round online phase and security in the standard model (with security under the DDH assumption). In Section 7.2 we show how to further reduce the complexity of the online stage, including a method for significantly reducing the communication of the online stage to be *independent of* $|C|$, in the random-oracle model. We stress that the most efficient protocols of [NNOB12, DPSZ12, DKL$^+$13], which also work in the random-oracle model, require at least $\mathcal{O}(|C|)$ communication in the online stage, and at least depth(C) rounds.

Concurrent work. In independent concurrent work, [HKK$^+$14] show how to amortize the number of garbled circuits for multiple-executions of secure computation in a similar fashion to ours. However, here, we additionally focus on reducing the overhead of the cheating-recovery step (e.g. by amortizing its number of garbled circuits as well, and by moving most of its cost to the offline stage) and on minimizing the number of exponentiations in the online stage. We note that in the cut-and-choose of [HKK$^+$14], P_2 always checks half of the circuits. In contrast, we show that better results can be obtained using different parameters; we believe that our analysis can be used in their protocol in a straightforward way.

2 High Level Description of Our Techniques

We describe the main ideas behind our protocols. For simplicity, we focus here on specific parameters, though in Section 4 we give a more general analysis of the possible parameters.

We begin by describing how cut-and-choose on Yao's protocol can be made more efficient (with low amortized cost) in batch settings where many computations take place. Then, we show how to achieve security in the online/offline setting where parties' inputs are fixed in the online phase. The low amortized cost for the batch setting is relevant both to the online/offline setting and to a setting where many computations take place in parallel.

2.1 Amortized Cut-and-Choose in Multiple Executions

We now describe how the number of circuits in cut-and-choose can be dramatically reduced in the case that many secure computation executions are run between two parties (either in parallel or in an online/offline setting). Assume that P_1 and P_2 would like to execute N protocols with maximum error probability of 2^{-s}, where s is a statistical security parameter. The naive approach of running the protocol of [Lin13] N times would require them to use a total number of garbled circuits of $N{\cdot}s$. As discussed earlier, our main goal in this paper is to reduce the number of garbled circuits by amortizing the overhead when many invocations of 2PC are executed.[3] The ideas described here will be used in both the batch protocol (Section 2.2) and the online/offline protocol (Section 2.3).

Recall that in cut-and-choose based two-party computation, P_1 prepares s garbled circuits, P_2 asks P_1 to open a random subset of them which are then checked by P_2, and then P_2 evaluates the remaining circuits. The main idea behind our technique is to run the cut-and-choose on *many* circuits, and then *randomly combine* the remaining ones into N sets (or "buckets"), where each set will be used for a single evaluation. The intuition behind this idea is as follows. The cheating recovery method of [Lin13] (described below in Section 2.2) ensures that security is preserved unless all evaluation circuits in a single set are incorrect. Now, by checking many circuits together and randomly combining them, the probability that one set will have all incorrect circuits (but yet no incorrect circuits were checked) is very small.

[3]We remark that it is possible to increase the number of check circuits and reduce the number of evaluated circuits in an online/offline version of the protocol of [Lin13], in order to improve the online time. For example, in order to maintain error of 2^{-40}, one can construct 80 circuits overall, and can check 70 and evaluate only 10. This will reduce the online time from approximately 20 to 10 (since in [Lin13] approximately half the circuits are evaluated). However, as we can see from this example, the total number of circuits grows very fast, rendering this approach ineffective.

In more detail, in our technique P_1 prepares $2N \cdot B$ garbled circuits and sends them to P_2, where B is a parameter we define later. For each circuit, P_2 chooses with probability 1/2 whether to check it or to use it later for evaluation. (This means that on average, P_2 checks $N \cdot B$ circuits. In our actual protocol we make sure that *exactly* $N \cdot B$ circuits remain. In addition, as we discuss below, we will typically not check half of the circuits and lower probabilities give better results.) Then, P_2 chooses a random mapping function $\pi : [N \cdot B] \to [N]$ that maps each of the remaining circuits in a "bucket" of B circuits, which will later be used as the evaluation-circuits of a single two-party protocol execution. Clearly, a malicious P_1 could prepare a small number of incorrect garbled circuits (say $\mathcal{O}(\beta)$), and not be caught in the checks with good probability (here $\beta < s$ and so $2^{-\beta}$ probability is too high). However, since π is chosen at random by P_2, we show that unless there are *many* incorrect circuits, the probability that any one of the buckets contains only incorrectly constructed garbled circuits is smaller than 2^{-s}. We prove that when $B \geq \frac{s}{1+\log N} + 1$, the probability that any bucket contains B incorrect circuits (and so all are incorrect) is at most 2^{-s}. Thus, the total number of circuits is $2N \cdot B = \frac{2Ns}{1+\log N} + 2N$. When $\log N > \frac{2s}{s-2} - 1$ we have that $2N \cdot B < N \cdot s$ and so a concrete improvement is obtained from just using [Lin13] even for just a few executions. Asymptotically, the number of circuits per execution is $\mathcal{O}(\frac{s}{\log N})$, which shows that when N gets larger, the amortized number of circuits becomes small. When plugging in concrete numbers that are relevant in practice, the improvement is striking. For example, consider $s = 40$ and $N = 512$ executions (observe that $\log N = 9$ and $\frac{2s}{s-2} - 1 = 1.10$ and so the condition is fulfilled). Now, for these parameters we have $B = \lceil \frac{s}{1+\log N} + 1 \rceil = 5$, and so only 512×10 garbled circuits are needed overall, with just 5 circuits evaluated in each execution. This is better by a factor of 4 compared to the Ns option. When many executions are run, even better numbers are obtained. For example, with $N = 524288$ we obtain that only 524288×6 circuits are needed overall (better by a factor of $6\frac{2}{3}$ than the naive option).

We remark that the probability of checking or evaluating a circuit greatly influences the number of circuits. Above, we have assumed that this probability is $\frac{1}{2}$. In Section 4 we analyse the above parameters in the general case. As we will see, better parameters are often achieved with different probabilities of checking a circuit. In addition, when working in the online/offline setting, this flexibility actually provides a tradeoff between the number of circuits in the online and in the offline phase. This is due to the fact that checking more circuits in the offline phase reduces the number of circuits to be evaluated in the online phase but increases the number of circuits checked in the offline phase.

In the protocol of [Lin13] secure computation is also used for the cheating recovery mechanism (described below in Section 2.2). This mechanism works as long as a *majority* of the circuits in a bucket are good. In the multiple-execution setting, we use a similar method for bucketizing these circuits, while guaranteeing that a majority of the circuits in any bucket be good (rather than just ensuring at least one good circuit). Using this method we significantly reduce the number of circuits needed for the cheating recovery. E.g., for $N = 1024$ protocol executions we need only buckets of size $B = 12$, and a total number of circuits of 24576 (i.e,, 24 circuits per execution). The protocol of [Lin13] requires about 125 circuits per execution, and thus we obtain an improvement of a factor of 5 in this part of the protocol (for these parameters).

More concrete examples. In Section 4 we provide a full analysis of the cheating probability for different choices of parameters. We describe some concrete examples here with $s = 40$, in order to provide more of an understanding of the efficiency gains obtained. In Tables 2 and 3 in Appendix A we show the cost for many different choice of parameters. When considering 2^{10} and 2^{20} executions, the best choices and the resulting cost is summarized in the following table (the bucket size is the number of circuits evaluated in the online phase):

Number of executions N	p	**Bucket size(B)**	**Overall number of circuits ($\lceil B \cdot N/p \rceil$)**	**Average # circuits per execution**
2^{10}	0.1	4	40,960	40.00
2^{10}	0.65	5	7,877	7.69
2^{10}	0.85	6	7,229	7.06
2^{20}	0.65	3	4,839,582	4.62
2^{20}	0.98	4	4,279,903	4.08

Table 1: Best parameters for $s = 40$ (p is the probability that a circuit is *not* checked)

Observe that in the case of $p = 0.1$, the average number of circuits is the same as in a single execution. However, it has the lowest online time. In contrast, at the price of just a single additional circuit in the online time, the offline time is reduced by a factor of over 5. In general, the bigger p is, the smaller the total number of balls is (up to a certain limit). However, the number of balls in each bucket grows proportionally with p. This means that using p it is possible to obtain a tradeoff between online and offline time. Specifically, a higher p means less circuits overall but more circuits in the online stage (where each bucket is evaluated), thereby reducing the offline time at the expense of increasing the online time. Conversely, a lower p means more circuits in the offline stage and smaller bucket and so less computation in the online stage.

We remark that improvements are not only obtained for large values of N. In the case of $N = 32$, with $p = 0.75$ we obtain buckets of size 10 (so 10 evaluations in the online phase) and an average of 13.34 circuits overall per execution. This is a considerable improvement over 40 circuits as required in [Lin13]. Of course, as N becomes smaller, the improvement is less significant. Nevertheless, for $N = 10$, with $p = 0.55$ we obtain an average of 20 circuits per execution, which is half the cost of [Lin13]. Going to the other extreme, with a huge number of executions the amortized cost becomes very small. Taking $N = 2^{30}$ (which isn't practical today but may be in the future), we can take $p = 0.99$ and obtain buckets of size 3 and an overall overage of just 3.03 circuits per execution. In Appendix A we also present graphs of the dependence of B and the total number of circuits in p, and how the average number of balls per bucket decreases as the number of buckets grows.

Regarding the number of circuits required for the cheating-recovery mechanism, for $N = 2^{10}$ we get that $B = 12$, and that the total number of circuits is $12 \times 1024 \times 2 = 24576$ (i.e,, 24 circuits per execution). For $N = 2^{20}$ we get that $B = 6$, and that the total number of circuits is $6 \times 1048576 \times 2 = 12,582,912$ (i.e,, 12 circuits per execution). This is in contrast to 125 circuits, as required in [Lin13].

2.2 Batch Two-Party Computation

The protocol of [Lin13] requires s garbled circuits per 2PC execution for achieving soundness of 2^{-s}. Here we would like to reduce this overhead when multiple executions of 2PC are executed in a batch setting; i.e., run in parallel. In this section, we assume that the reader is familiar with the protocol of [Lin13].

If we try to use the protocol of [Lin13] as-is in the batch setting, and take advantage of the ideas presented in Section 2.1, two problematic issues arise. We now describe these issues and how we solve them.

First, in the cut-and-choose oblivious transfer of [Lin13], the receiver uses only one input to all OTs, whereas in the batch setting, P_2 should be able to input many different inputs, and they have to be consistent in each bucket. This consistency of P_2's input is enforced by having P_2 prove in zero knowledge that its OT queries are for the same input in all circuits. In order to enable P_2 to use separate inputs in each bucket, we modify the protocol as follows. First, P_2 privately selects which circuits to use and how to bucket them before the OTs are executed. Then, the parties run the cut-and-choose OT, where P_2 inputs its j-th input in the circuits that it chose to be in the j-th bucket. However, P_2 does *not* prove consistency of its input at this point (since the buckets are not yet known to P_1), but rather postpones this proof until after it sends the cut and random mapping to buckets to P_1. After the mapping to buckets has been given to P_1, it is possible for P_2 to separately prove in zero knowledge for every bucket that its OT queries in the j-th bucket are for the same input. Observe also that since this proof is given before P_2 can evaluate any circuit, no information can be gained if P_2 tries to cheat.

A second issue that arises when trying to use the protocol of [Lin13] in the batch setting is what P_2 does in the case that it gets different outputs in some of the evaluated circuits. We call this mechanism of [Lin13] $\mathsf{cheating\ recovery}$ since it enables P_2 to obtain correct output when P_1 has tried to cheat. In order for this mechanism to work, [Lin13] uses the same output labels in *all* circuits, and in case P_2 gets different labels for the same wire (meaning different outputs), the two labels allow it to recover P_1's input. Unfortunately, this technique cannot work in the batch setting, since there, naturally, P_2 would get different outputs from different buckets, and thus will always learn two labels of some output wire. This would enable a cheating P_2 to learn P_1's input.

Our solution to this problem is as follows. For simplicity, assume that there is only one output wire, and assume that D is a special constant that is revealed to P_2 in the case that it receives different output values on the wire in different circuits (we later describe how this "magic" happens). Recall that in [Lin13], a second, lighter, two-party computation is executed with a boolean circuit C', where P_1 inputs (x, D) (with x being the value used in computing the actual circuit), P_2 inputs d, and $C'(x, D, d) = x$ if $d = D$, and 0 otherwise. Thus, if P_2 obtained D due to receiving different outputs in different circuits, then in the second two-party computation it inputs $d = D$ and learns x, thereby enabling it to locally compute the correct output $f(x, y)$. Otherwise, if learns nothing about x; in addition, P_1 does not know if P_2 learned x or not.

Instead of using the same output labels in all garbled circuits, P_1 uses random ones (as in the standard Yao's circuit). After P_2 announces the "cut" in the offline stage and the mapping to the buckets, P_1 opens the check circuits and P_2 verifies them as described before. Then in the online stage the parties follow the next steps. For every bucket (separately), P_1 chooses a random D. Concretely, consider the j-th bucket; then P_1 chooses random values D_j and R_j. Denote the garbled circuits in the j-th bucket by $gc_1, gc_2, \ldots gc_B$. Furthermore, denote the output-wire labels of circuit gc_i by W_i^0, W_i^1. P_1 sends the encryptions $\{\ \mathsf{Enc}_{W_i^0}(R_j),\ \mathsf{Enc}_{W_i^1}(R_j \oplus D_j)\ \}_{i=1,\ldots,B}$. P_1 also sends P_2 the hash $\mathsf{Hash}(D_j)$. The purpose of these encryptions and hash is that in case P_2 learns two output labels that correspond to different outputs, P_2 can learn both R_j and $R_j \oplus D_j$ and can use it to recover D_j. It then verifies that it has the right D_j using $\mathsf{Hash}(D_j)$. (In the case of many output wires, each output wire in a bucket encrypts in the above way using a different R_j. Thus, D_j can be obtained from any *pair* of output wire labels in the j-th bucket.)

After P_2 evaluates the circuits of C, it learns a set of labels $W' = \{W'_1, \ldots, W'_B\}$. P_2 uses the values of W' to decrypt the corresponding $c_i^0 = \mathsf{Enc}_{W_i^0}(R_j)$ or $c_i^1 = \mathsf{Enc}_{W_i^1}(R_j \oplus D_j)$. In case P_2 learns both W_i^0 and W_i^1, it can recover $d_j = \mathsf{Dec}_{W_i^0}(c_i^0) \oplus \mathsf{Dec}_{W_i^1}(c_i^1)$ (which should equal $D_j = R_j \oplus (R_j \oplus D_j))$. In case P_2 gets many "potential" D's (which can happen if P_1 does not construct the values honestly), it can identify the correct one using the value $\mathsf{Hash}(D_j)$. Next, the parties execute the 2PC protocol with the circuit $C'(x, D, d)$, and P_2 learns x in case it learned the correct D_j earlier. Finally, P_2 verifies that P_1 constructed all of the values for the cheating recovery correctly. This check is carried out after the 2PC protocol for C' has concluded, since at this point revealing D_j to P_2 can cause no damage. For this check, P_1 reveals all of the pairs W_i^0, W_i^1, allowing P_2 to check that the encryptions $\{\mathsf{Enc}_{W_i^0}(R_j), \mathsf{Enc}_{W_i^1}(R_j \oplus D_j)\}_{i=1,\ldots,B}$ and $\mathsf{Hash}(D_j)$ are consistent. Since P_1 can cheat regarding the output labels W_i^0, W_i^1, we require that when it sends a garbled circuit (before the cut is revealed), it also sends commitments on all the output wire labels of that circuit. These commitments are checked if the circuit is chosen to be checked in the cut-and-choose. Thus, any good circuit has the property that the output labels encrypt R_j and $R_j \oplus D_j$.

Unfortunately, the above does not suffice to ensure that P_2 learns D_j in the case that there are two different outputs. This is due to the fact that it is only guaranteed that *one* circuit in the bucket is good. Now, if P_2 receives two different outputs in two different circuits, then the second circuit may *not* be good and so P_2 may obtain the correct R_j from the good circuit but some value $S_j \neq R_j \oplus D_j$ from the other.

Nevertheless, in the case that P_2 received different outputs, but did not obtain D_j that is consistent with the hashed value $\mathsf{Hash}(D_j)$ sent by P_1, party P_2 simply outputs the output of the garbled circuit for which the output labels it received from the evaluation are all consistent with the output labels that were decommitted. To see why this suffices, observe that P_2 receives two different outputs, and one of them is from a good circuit. Denote the two circuits from which P_2 receives different outputs by gc_1, gc_2, and denote by gc_1 the circuit that was correctly garbled. Then, there are two possibilities: **(1)** P_2 obtained the correct D_j, and thus recovers x using the second 2PC (and can output the correct $f(x, y)$ by just computing the function f with P_1's input x); **(2)** P_2 did not recover the correct D_j, meaning that the output labels it received do not decrypt R_j and $R_j \oplus D_j$. However, since gc_1 is correct, including the commitments on its output labels, and since $\mathsf{Enc}_{W_i^0}(R_j)$ and $\mathsf{Enc}_{W_i^1}(R_j \oplus D_j)$ are checked, gc_1 gives P_2 the correct value (either R_j or $R_j \oplus D_j$, depending on the output bit in question). Now, if the output label that P_2 received from gc_2 also decrypts its corresponding R_j or $R_j \oplus D_j$, then P_2 should have learnt the correct D_j. This means that the label that P_2 received in gc_2 does *not* match the label that P_1 revealed from the decommitment on gc_2's output labels. Thus, P_2 knows that gc_1 is the correct circuit and not gc_2, and can take the output of the computation to be the output of gc_1. (Note that by what we have explained, if P_2 does not obtain D_j and the checks on the commitments and encryptions passed, then there is only *one circuit* in which the output labels obtained by P_2 are consistent with the commitments. Thus, there is no ambiguity regarding the output.)

Although the above issues are the main parts of the cheating-recovery process of our protocols, there are other small steps that are needed in order to make sure that the protocol is secure. For example, P_2 should verify that P_1 inputs the correct D to C'. Also, efficiency-wise, recall that $3s$ garbled circuits of C' are used in the protocol of [Lin13]; here, we amortize their cut-and-choose as well, as described above. These issues are dealt with in the detailed description of the protocol in Section 6.

2.3 Two-Party Computation with Online/Offline Stages

Protocols for secure computation in the presence of malicious adversaries via cut-and-choose on garbled circuits employ a number of methods to prevent cheating. First, many circuits are sent and a fraction checked, in order to ensure that some of the garbled circuits are correct (this is the basic cut-and-choose). Second, since many circuits are evaluated in the evaluation phase, it is necessary to force P_1 and P_2 to use the same input in every circuit in an evaluation. Third, so-called selective OT attacks must be thwarted (where a cheating P_1 provides correct circuits but partially incorrect values in the oblivious transfer phase where P_2 receives keys to decrypt the circuits, based on its input). Finally, the cheating recovery technique described in Section 2.2 is used to enable P_2 to complete the computation correctly in case some of the evaluation circuits are correct and some are incorrect. In all existing protocols, some (if not all) of the aforementioned checks utilize the fact that the parties' inputs are given and fixed before the checks are carried out (in fact, in [Lin13] even the basic cut-and-choose on circuits is intertwined with the selective OT attack prevention and so requires the inputs to already be fixed). Thus, these protocols do not work in the online/offline setting.

In this section, we describe how to deploy these methods in an online/offline setting where the checks are carried out in the offline setting, and the online setting should be very fast.[4] Ideally, the online setting should have no exponentiations, and should involve some minimal communication (that is independent of the circuit size) and the evaluation of the circuits in the bucket only. Our protocol achieves this goal, with some small additional work in the online stage. We note that in the standard model we do require some exponentiations in the online phase, but just two per circuit which in practice is insignificant. In addition, P_1 needs to transmit B garbled circuits to P_2 for evaluation in the online phase, where B is the bucket size (in practice, a small constant of between 4 and 6). We also present a variant of our protocol in the random oracle model that requires no exponentiations whatsoever in the online phase, and has very little communication; in particular, the communication is independent of the circuit size. The use of a random oracle is due to problems that arise when adaptively-secure garbled circuits [BHR12a] are needed. This issue is discussed separately in Section 2.4.

Ensuring correctness of the garbled circuit. Intuitively, the aim of the cut-and-choose process is to verify that the garbled circuits are correct. Thus, it is possible to run this process (send all circuits and then open and check a fraction of them) in an offline stage even before the parties have inputs. Then, in the online stage, when the parties have inputs and would like to compute the output of the computation as fast as possible, they only need to evaluate the remaining "evaluation" circuits, which results in a much lower latency.

Enforcing P_1's input consistency. We start with the approach taken in [MF06, LP11, SS11]. Let wire j be an input-wire of P_1. In a standard garbling process, two random strings are chosen as the labels of wire j. However, here, the two labels are chosen to be commitments to the actual value they represent, e.g., the label that corresponds to the bit 0 is actually a commitment to 0 (more exactly, the label is the output of a hash function, which is also a randomness extractor, on the appropriate commitment). In addition, the commitments used have the property that one can

[4]Our aim here is to reduce the work of the online stage as much as possible, in order to achieve very fast computation in the online stage. Tradeoffs between the offline and online stages are of course possible, and we leave this for future work.

prove equality of multiple committed messages with high efficiency, without revealing the actual messages.

This solution can be used in the online/offline setting in a straightforward way. Namely, when a circuit is checked, these commitments are checked as well. In contrast, when a set of circuits is used for evaluation, P_1 sends the commitments that correspond to its input, along with a proof that they are all commitments to the same bit 0 or 1. However, the disadvantage of this method is that it requires a few exponentiations per bit of P_1's input, and we would like to move all exponentiations possible to the offline stage. In order to achieve this, instead of directly computing $f(x, y)$, we modify the garbled circuit to compute the function $f'\big(x^{(1)}, x^{(2)}, y\big) = f\big(x^{(1)} \oplus x^{(2)}, y\big)$, where $x^{(1)}$ and $x^{(2)}$ are P_1's inputs and are chosen randomly by P_1 under the constraint that $x^{(1)} \oplus x^{(2)} = x$. In the garbling process, the garbled labels of the wires of $x^{(1)}$ are constructed using the commitment method of [MF06, LP11, SS11], while the labels of the wires of $x^{(2)}$ are standard (i.e., random strings). In addition, for each wire of $x^{(2)}$, P_2 sends commitments on the two input-wire labels (i.e., if the labels are W^0, W^1, P_1 sends $\mathsf{Com}(0\|W^0), \mathsf{Com}(1\|W^1)$). Now, in the offline stage, when a circuit is checked, P_2 verifies that all of the above was followed correctly. Furthermore, in the circuits that are to be evaluated, P_1 chooses a *random* $x^{(1)}$ and sends the commitments that correspond to $x^{(1)}$ along with the proof of message equality. This proves to P_2 that P_1's input $x^{(1)}$ is the same in all evaluated circuits (of course, at least in the properly constructed circuits). All this is carried out in the offline phase.

In the online stage, when P_1 knows x, it sends P_2 the actual value of $x^{(2)} = x^{(1)} \oplus x$, along with the decommitments of the labels that correspond to $x^{(2)}$ (the decommitments prove that the same $x^{(2)}$ is sent in all circuits). We stress that $x^{(2)}$ is sent in the clear, and is the same for all evaluated circuits (this reveals nothing about x since $x^{(1)}$ is random and not revealed). As a result, the same $x^{(1)}$ and $x^{(2)}$ is used in all circuits (the consistency of $x^{(1)}$ is enforced in the offline phase, and the consistency of $x^{(2)}$ is immediate since it is sent in the clear) and so the same x is used in all evaluated circuits. Note that no exponentiations are needed in the online stage, and only a small number of decommitments and decryptions are computed.

In summary, online/offline consistency of P_1's input is obtained by randomly splitting P_1's input into a secret part $x^{(1)}$ (which is dealt with in the offline stage), and a public part $x^{(2)}$ which can be revealed in the online stage. Since $x^{(2)}$ can be chosen to equal $x \oplus x^{(1)}$ in the online phase, after x is known, the correct result is obtained and consistency is preserved at very little online cost.

Protecting against selective-OT attacks. We use a variant of the cut-and-choose oblivious transfer protocols of [LP11, Lin13], and modify it to work in the online/offline setting. The modification is similar to the method used for P_1's input; i.e., instead of computing the function $f'\big(x^{(1)}, x^{(2)}, y\big) = f\big(x^{(1)} \oplus x^{(2)}, y\big)$ as above, the parties compute $f''\big(x^{(1)}, x^{(2)}, y^{(1)}, y^{(2)}\big) = f\big(x^{(1)} \oplus x^{(2)}, y^{(1)} \oplus y^{(2)}\big)$, where P_2 uses a random value for $y^{(1)}$ in the offline stage, and later uses $y^{(2)} = y^{(1)} \oplus y$ once it knows its input y in the online stage. The cut-and-choose oblivious transfer protocol is used for protecting against selective OT attacks on the OTs that are used for P_2 to learn the garbled labels of $y^{(1)}$. In contrast, the labels of $y^{(2)}$ are obtained by having P_2 send $y^{(2)}$ in the clear and having P_1 send the associated garbled labels (these labels are committed in the offline phase and thus the labels are sent to P_2 as decommitments, which prevents P_1 from changing them). As before, all exponentiations are carried out in the offline stage alone.

Cheating recovery. The protocol of [Lin13] uses a cheating recovery process for allowing P_2 to learn x in case P_2 obtains different outputs from the evaluated circuits. This method allows for only s circuits to be used in order to obtain 2^{-s} cheating probability, since an adversary can only cheat if *all* checked circuits are correct and *all* evaluated circuits are incorrect. However, the protocol of [Lin13] requires the parties to run the cheating recovery process *before* the check circuits are opened, which obviously is unsatisfactory in the online/offline setting since now P_2 does all the expensive checking in the online stage again.

Our solution for this problem is the same solution as described above for the batch setting; see Section 2.2. Namely, assume that D is a special constant that is revealed to P_2 in the case that it receives different output values on the wire in different circuits, and for simplicity assume that there is only one output wire. We would like to securely compute the boolean circuit $C'\big(x^{(1)}, D, d\big)$, where $(x^{(1)}, D)$ are P_1's input, d is P_2's input, and $C'\big(x^{(1)}, D, d\big) = x^{(1)}$ if $d = D$, and 0 otherwise. We note that only P_2 receives output (since the method requires that P_1 not know if P_2 learned D or not). Recall that $x^{(1)}$ is the secret part of P_1's input, and so if $x^{(1)}$ is obtained by P_2 then it can compute $x = x^{(1)} \oplus x^{(2)}$ and obtain P_1's real input. Everything else in this solution is identical to the solution described in Section 2.2; the use of $x^{(1)}$ instead of x enables us to check the circuits used in the cheating-recovery mechanism in the offline phase.

There are several other subtle issues to take care of regarding the secure computation of C'. First, we require P_1 to use the same $x^{(1)}$ in C and C'. This is solved by using commitments for the input-wire labels for $x^{(1)}$ as described above. Second, we need to protect the OTs for P_2 to learn the labels of d from selective-OT attacks. This is solved using the variant of cut-and-choose OT we use for the OTs for C. Third, in order to push all the expensive exponentiations to the offline stage, we split the parties inputs in the cheating-recovery circuit C' into random inputs in the offline stage and public inputs in the online stage as we did with the inputs of C. Note that the above issues are only part of the cheating-recovery process of our protocols, and additional steps are needed in order to make sure that the protocol secure.

2.4 On Adaptively Secure Garbled Circuits in the Online/Offline Setting

The standard security notion of garbled circuits considers a *static* adversary who chooses its input before seeing the garbled circuit. While this notion suffices for standard 2PC protocols (e.g., [LP07, LP11, SS11] where the oblivious transfers that determine P_2's input can be run before the garbled circuits are sent), it causes a problem in the online/offline setting. This is due to the fact that we would like to send all the garbled circuits in the offline stage in order to reduce the online stage communication. However, this means that the circuits are sent before the parties (and in particular the adversary) have chosen their inputs.

Recently, [BHR12a, AIKW13] introduced an *adaptive* variant of garbled circuits, in which the adversary is allowed to choose its input *after* seeing the garbled circuit. Indeed, adaptively secure garbling scheme would allow us to send all the garbled circuits in the offline stage before the parties have chosen their inputs. However, the only known efficient constructions of adaptively secure garbled circuit are in the random-oracle model [BHR12a, AIKW13].[5]

[5][BHR12a] also present a construction in the standard model which requires the online stage communication to be the same size as the garbled circuit, but this does not help us to reduce the online communication. In addition, [BHK13] presents a construction in the standard model based on *UCE-hash* functions. However, the only known proven construction of UCE-hash is in the ROM.

We do not try to present new solutions to the adaptively-secure garbled-circuit problem in this work. Rather, we present two options based on current constructions. Our first solution is in the standard model and works by having P_1 send only the checked garbled circuits in the offline stage. In contrast, the evaluation garbled circuits are sent in the online stage. These latter circuits are committed (using a trapdoor commitment) in the offline stage, and this enables the simulator to actually construct the garbled circuit after the input is given, solving the adaptive problem. The drawback of this solution is that significant communication is needed in the online stage, incurring considerable cost. Our second solution is to use the random-oracle construction of [PSSW09, BHR12a]. In this case, *all* of the garbled circuits are sent in the offline stage, and the communication of the online stage depends only on the number of inputs and outputs of the circuits (and the security parameters). Thus, we obtain a clear tradeoff between the security model and efficiency. We believe that any future construction of efficient adaptively secure garbled circuits in the standard model may be plugged into the second construction in order to maintain its low communication and remove the random-oracle.

3 Preliminaries

3.1 Notation and Basic Primitives

We assume that all parties agree on a cyclic group $\mathbb{G}$ of prime order q and with generator g. Let $\mathsf{Hash}(\cdot)$ be a collision-resistant hash function, $\mathsf{REHash}(\cdot)$ be a collision-resistant hash function that is a suitable randomness extractor (e.g., see [DGH$^+$04]), $\mathsf{Enc}(key, m)$ or $\mathsf{Enc}_{key}(m)$ be a symmetric encryption of message m under key key, $\mathsf{Com}(\cdot)$ be a commitment, and $\mathsf{PCommit}_h(m, r)$ be a Pedersen commitment [Ped92] on message m with generators g, h and randomness r; we denote by $\mathsf{PCommit}_h(m)$ a Pedersen commitment as above with a uniformly chosen r. Last, let s be a statistical security parameter and k be a computational security parameter (and so q is of length k). We denote the concatenation of string x with string y by $x|y$.

Boolean Circuits. Let C be a boolean circuit, and denote by $C(x)$ the evaluation of C on the input x. We assume that all wires are identified by unique, incremental identifiers, and the input wires are first. Assuming that the input is of length l, we let $\mathsf{Inputs}(C) = \{1, \ldots, l\}$ be the set that identifies the input wires, and let $\mathsf{Outputs}(C)$ be the set that identifies the output wires.

Garbled Circuits. Our main protocol uses the garbled circuit of Yao [Yao86] and can work with standard garbling schemes (see [LP09, BHR12b] for examples). A garbling scheme consists of two algorithms Garble and $\mathsf{Evaluate}$. Given a circuit C and security parameter 1^k, $\mathsf{Garble}(1^k, C)$ returns the triplet (gc, en, de), where gc is the garbled circuit, en is the set of ordered pairs of input labels $\left\{(W_i^0, W_i^1)\right\}_{i \in \mathsf{Inputs}(C)}$ and de is the set of ordered pairs of output labels $\left\{(W_i^0, W_i^1)\right\}_{i \in \mathsf{Outputs}(C)}$. We also require that each wire i is associated with two labels, denoted by W_i^0, W_i^1.

Given input $x \in \{0,1\}^l$, and the set of labels $X = \left\{W_i^{x_i}\right\}_{i \in \mathsf{Inputs}(C)}$ that correspond to x, one can use $\mathsf{Evaluate}(gc, X)$ to compute the garbled output Y, and then map Y to the actual output $y = C(x)$ using the set de.

See [BHR12b, BHR12a] for formal definitions of security for garbling schemes. Next, we informally describe the properties needed in our protocols. For *correctness*, we require that the decoding of $\mathsf{Evaluate}(gc, X)$ equals $C(x)$ for every circuit C, input $x \in \{0,1\}^l$, security parameter $k \in \mathbb{N}$ and

$(gc, en, de) \leftarrow \mathsf{Garble}(1^k, C)$. *Privacy* with respect to *static* adversaries (e.g., [LP07, BHR12b]) is defined by a game in which **(1)** the adversary chooses a circuit $C(\cdot)$ and input x; **(2)** the challenger flips a coin b and sends (gc, de, X) that is the result of garbling $C(\cdot)$ and x if $b = 1$, and the result of a simulator $\mathsf{Sim}(1^k, C(x))$ otherwise.[6] We say that the adversary wins the game if it guesses b correctly. We require that for every polynomial-time Adv there exists a simulator Sim such that the probability that Adv wins is negligibly close to $1/2$. *Authenticity* is defined by a game in which the adversary chooses circuit $C(\cdot)$ and input x, and receives back (gc, X) that is the result of garbling $C(\cdot)$ and x (but not de). We say that the adversary wins the game if it outputs Y such that the decoding of Y using de results in $y \neq C(x)$ (and, obviously, $y \neq \perp$).

In this work we also need the stronger notion of *adaptive security*, in which the adversary picks its input after seeing gc. That is, privacy is defined by a game in which **(1)** the adversary sends $C(\cdot)$ and gets gc, de; **(2)** the challenger flips a coin b and sends (gc, de) that is the result of garbling $C(\cdot)$ if $b = 1$, and the result of a simulator $\mathsf{Sim}(1, 1^k)$ otherwise; **(3)** the adversary sends input x and receives X, which is an garbled encoding of x if $b = 1$, or the output of $\mathsf{Sim}(2, C(x))$ otherwise. We say that the adversary wins the game if it guesses b correctly. Similarly, authenticity is defined by a game in which **(1)** the adversary chooses circuit $C(\cdot)$ and receives back gc that is the result of garbling $C(\cdot)$ (but without de); **(2)** the adversary sends input x and receives X which is the garbled encoding of x. We say that the adversary wins the game if it outputs Y such that the decoding of Y using de results in $y \neq C(x)$ (and $y \neq \perp$). We require that no polynomial-time adversary can win with non-negligible probability.

[BHR12a] and [AIKW13] present adaptively secure garbling schemes in the random-oracle model. [AIKW13] proves that a garbling scheme in which X is shorter than the output length *cannot* be adaptively private in the standard model. [BHK13] shows instantiations based on *UCE hash functions* (which is possibly a weaker assumption/abstraction than the ROM). If the UCE hash function is instantiated by a random-oracle, then there is no additional overhead for transforming from static to adaptive secure garbling. The problem of how to construct an adaptively secure garbling scheme from standard and efficient primitives (say, symmetric key encryption) is an interesting open question.

3.2 Security for Multiple Executions

In the online/offline setting, the parties run multiple executions, and inputs in later executions may depend on outputs already received. In this section, we define security for this setting, which differs from the ordinary stand-alone setting for secure computation. We remark that in the batch setting, security is defined by simply considering the stand-alone notion for a functionality $F(\vec{x}, \vec{y}) = (f(x_1, y_1), \ldots, f(x_N, y_N))$ that receives a vector of inputs from each party and computes the function separately on each corresponding pair of inputs. This notion is the standard one since inputs are not chosen adaptively. We now proceed to the definition for the online/offline setting.

In Figure 1 we define the ideal functionality for *multiple adaptive secure two-party computations* (where by *adaptive* we mean that inputs in later executions can depend on outputs already received).

Security is defined using the ideal-real paradigm, where the execution in the real world is compared with a simulated execution in the ideal world in which the parties compute the function with the help of the ideal functionality, and thus, is secure by definition.

[6]We assume that the topology of $C(\cdot)$ is public.

Functionality $\mathcal{F}^f_{M2PC}$

$\mathcal{F}^f_{M2PC}$ parameterized by a function $f(\cdot,\cdot)$ that always receives inputs of the same length.

- Upon receiving a message $\mathsf{start}(N)$ from P_2, continue as following for $j = 1, \ldots, N$:
  - Upon receiving a message $\mathsf{input}(j, x_j)$ from P_1, store (j, x_j). Similarly, store (j, y_j) upon receiving a message $\mathsf{input}(j, y_j)$ from P_2.
  - If $x_i = \perp$ or $y_i = \perp$, send $\perp$ to both parties and halt.
  - After receiving both inputs (j, x_j) and (j, y_j), compute $f(x_j, y_j)$ and send $\mathsf{output}(j, f(x_j, y_j))$ to P_2.
  - If Adv controls P_2 and P_2 sends $\mathsf{abort}(j)$, then send $\mathsf{abort}(j)$ to P_1 and halt. Otherwise, send $\mathsf{output}(j, f(x_j, y_j))$ to P_1.

Figure 1: Ideal functionality $\mathcal{F}^f_{M2PC}$.

In the standalone setting, we define security as follows. In order to allow the parties (including honest ones) to choose their inputs adaptively, based on the outputs of previous steps, we follow [Lin08] and define $\bar{M} = (M_1, M_2)$ to be *input-selecting machines.* Machine M_i is a probabilistic polynomial-time Turing machine that determines P_i's inputs given an initial input, the index j of the input to be generated (x_j or y_j), and outputs $f(x_i, y_i)$ that were obtained from executions that have already concluded. (The size of the initial input is not bounded. In particular, it can consist of N inputs, where the j-th input is used in the j-th 2PC invocation, which results in a non-adaptive set of inputs.)

We use two security parameters. The standard computational security parameter is denoted k, and all parties run in time that is polynomial in k. In addition, we let s denote a statistical security parameter; this is used to allow a statistical error of 2^{-s} that is independent of the computational cryptographic primitives.

Let $\text{REAL}_{\Pi,\mathcal{A}(aux),i,\bar{M}}(x, y, k, s)$ be the output vector of the honest party and the adversary Adv controlling P_i from the real execution of Π, where aux is an auxiliary information, x is P_1's initial input, y is P_2's initial input, $\bar{M} = (M_1, M_2)$ are input-selecting machines, k is the computational security parameter, and s is the statistical security parameter. Likewise, denote the output vector of the honest party and the adversary Sim controlling P_i from the execution in the ideal model where a trusted party computes the ideal functionality $\mathcal{F}^f_{M2PC}$ depicted in Figure 1 by $\text{IDEAL}_{\mathcal{F}^f_{M2PC},\mathsf{Sim}(aux),i,\bar{M}}(x, y, k, s)$.

Definition 3.1. *A protocol* Π securely computes f with multiple executions *if for any non-uniform probabilistic polynomial-time adversary* Adv *controlling* P_i *(for* $i \in \{1, 2\}$*) in the real model, and for any pair of probabilistic polynomial-time input-selecting machines* $\bar{M} = (M_1, M_2)$*, there exists a non-uniform probabilistic polynomial-time adversary* Sim *controlling* P_i *in the ideal model, such that for every non-uniform probabilistic polynomial-time distinguisher* $\mathcal{D}$ *there exists a negligible function* μ *such that for all* $x, y, aux \in \{0, 1\}^*$ *and for all* $k, s \in \mathbb{N}$*:*

$$\Big| \Pr\Big[\mathcal{D}(\text{REAL}_{\Pi,\mathcal{A}(aux),i,\bar{M}}(x, y, k, s)) = 1\Big] - \Pr\Big[\mathcal{D}(\text{IDEAL}_{\mathcal{F}^f_{M2PC},\mathsf{Sim}(aux),i,\bar{M}}(x, y, k, s)) = 1\Big]\Big| \le \mu(k) + \frac{1}{2^s}.$$

Alternatively, security can be defined in the *UC framework* of [Can01], requiring that the protocol *UC-realizes* $\mathcal{F}^{f}_{M2PC}$ in the $(\mathcal{F}_{AUTH}, \mathcal{F}_{SMT})$-hybrid model (with error $\mu(k) + 2^{-s}$). The downside of this alternative is that the ZKPoKs needed in our protocols become slightly less efficient.

We present our protocols with respect to the standalone definition, and briefly discuss how to adapt them to be UC-secure in Section **??**.

4 Combinatorics of Multiple Cut-and-Choose: Balls and Buckets

In this section we deal with *balls* and *buckets*. A ball can be either normal or cracked. Similarly to cut-and-choose, we describe a game in which party P_1 prepares a bunch of balls, P_2 checks a subset of them and aborts if some of them are cracked, and otherwise randomly places the remaining ones in buckets. Our goal is to bound the probabilities that (a) one of the buckets consists of only cracked balls (i.e., a fully-cracked bucket), and (b) there is a bucket in which the majority of the balls are cracked (i.e., a majority-cracked bucket).[7] We follow the analysis of [Nor13, Theorem 4.4] and [Nor13, Theorem 6.2], while handling different and slightly more general parameters.

4.1 The Fully-Cracked Bucket Game

Let Game 1 be the following game. P_2 chooses three parameters p, N and B, and sets $M = \left\lceil \frac{NB}{p} \right\rceil$ and $m = NB$. A potentially adversarial P_1 (who we will denote by Adv) prepares M balls and sends them to P_2. Then, party P_2 chooses at random a subset of the balls of size $M - m$; these balls are checked by P_2 and if one of them is cracked then P_2 aborts. Index the balls that are not checked by $1, \ldots, m$. P_2 chooses a random mapping function $\pi : [m] \to [N]$ that places the unchecked balls in buckets of size B. We define that $\mathsf{Game}_1(\mathcal{A}, N, B, p) = 1$ if and only if P_2 does not abort and there is a fully cracked bucket (note that $M = \left\lceil \frac{NB}{p} \right\rceil$ and $m = NB$, and so M and m are fully defined by N, B and p). We prove the following theorem:

Theorem 4.1. *Let s be a statistical security parameter, and let B, $N \in \mathbb{N}$ and $p \in (0,1)$ be as above. If $N \geq \frac{1}{1-p}$ and*

$$B \geq \frac{s + \log N - \log p}{\log(N - Np) - \frac{\log p}{1-p}},$$

then for every adversary Adv *it holds that* $\Pr[\mathsf{Game}_1(\mathcal{A}, N, B, p) = 1] < 2^{-s}$.

Proof. Let t be the number of cracked balls that Adv sends. Let E_{na} denote the event that P_2 does not abort (given that t balls are cracked), and let E_{fc} be the event that given m balls of which t of them are cracked, the mapping of π results a fully-cracked bucket. We have that

$$\Pr[\mathsf{Game}_1(\mathcal{A}, N, B, p) = 1] \;=\; \Pr[E_{na} \wedge E_{fc}] = \Pr[E_{na}] \cdot \Pr[E_{fc}]$$

since the two events are independent.

[7]The balls in this game represent garbled circuits, and cracked balls are incorrectly formed garbled circuits. The first game, relating to a fully-cracked bucket, provides an analysis of how many circuits are needed for the main computation; this is according to the methodology of [Lin13] where security holds unless *all* circuits are bad. The second game, relating to a majority-cracked bucket, provides an analysis of how many circuits are needed for the cheating recovery computation, where an adversary can only cheat if a *majority* of the circuits are bad.

We start by bounding $\Pr[E_{na}]$. The probability that P_2 does not check the t cracked balls is equal to the probability that all t cracked balls are chosen to *not* be checked, and then $m-t$ more balls are chosen to *not* be checked (amongst the $M-t$ remaining balls). Thus, we have:

$$\Pr[E_{na}] = \frac{\binom{M-t}{m-t}}{\binom{M}{m}}.$$

Since $M \geq \frac{m}{p}$, we have that

$$\frac{\binom{M-t}{m-t}}{\binom{M}{m}} \leq \frac{\binom{\frac{m}{p}-t}{m-t}}{\binom{\frac{m}{p}}{m}}. \tag{1}$$

In addition,

$$\binom{\frac{m}{p}-i-1}{m-i-1} = \frac{m-i}{\frac{m}{p}-i} \cdot \binom{\frac{m}{p}-i}{m-i} \leq p \cdot \binom{\frac{m}{p}-i}{m-i}, \tag{2}$$

for $i = 1, \ldots, t$. Thus, by Equations (1) and (2) we get that

$$\Pr[E_{na}] \leq p^t. \tag{3}$$

We proceed to bound $\Pr[E_{fc}]$. Fix B balls out of m balls, denoted β, and let $\pi : [m] \to [N]$ be a B-regular random mapping (i.e., each value $\ell \in N$ has exactly B pre images). We analyze the probability that all β are in the same bucket:

$$\Pr[\exists \ell \in [N] \ : \ \forall i \in \beta \ \pi[i] = \ell].$$

The number of functions π overall that are B-regular equals $\frac{m!}{(B!)^N}$. This is because one way to define such a function is to take a permutation over $[m]$ and define the first bucket to be the first B elements and so on. Since order does not matter inside each bucket and there are N buckets, we divide by $(B!)^N$ ($B!$ times itself N times). Now, we count how many of these functions place all the balls in B in the same bucket. First, the number of functions that place all the balls in β in the ℓth bucket (for a fixed ℓ) is $\frac{(m-B)!}{(B!)^{N-1}}$. Since there are N buckets, we have that there are $N \cdot \frac{(m-B)!}{(B!)^{N-1}}$ such functions overall. This means that the probability that all balls in β are in the same bucket is

$$\frac{N \cdot \frac{(m-B)!}{(B!)^{N-1}}}{\frac{m!}{(B!)^N}} = N \cdot \frac{(m-B)!}{(B!)^{N-1}} \cdot \frac{(B!)^N}{m!} = N \cdot \frac{B \cdot (B-1) \cdots 1}{m \cdot (m-1) \cdots (m-B+1)} = N \cdot \binom{m}{B}^{-1}.$$

Recall that t is the number of cracked balls out of the m balls overall. Since there are $\binom{t}{B}$ subsets of size B, a union bound gives us the bound

$$\Pr[E_{fc}] \leq N \cdot \binom{t}{B}\binom{m}{B}^{-1}. \tag{4}$$

By Equations (3) and (4) we have that for any fixed t

$$\Pr[\mathsf{Game}_1(\mathcal{A}, N, B, p) = 1] \leq p^t \cdot N \cdot \binom{t}{B}\binom{m}{B}^{-1}.$$

.

Let $\alpha(t, N, B, p) = p^t \cdot N \cdot \binom{t}{B}\binom{m}{B}^{-1}$ and observe that

$$\frac{\alpha(t+1, N, B, p)}{\alpha(t, N, B, p)} = p \cdot \frac{\binom{t+1}{B}}{\binom{t}{B}} = p \cdot \frac{(t+1)!(t-B)!}{t!(t+1-B)!} = p \cdot \frac{t+1}{t+1-B},$$

so $\alpha(\cdot, N, B, p)$ is increasing when $t < \frac{B-1+p}{1-p}$ and decreasing otherwise. Thus, $\alpha(\cdot, N, B, p)$ is maximized at $t' = \frac{B-1+p}{1-p}$. Since $p < 1$ we have that $t' < \frac{B}{1-p}$. For simplicity, assume that $\frac{B}{1-p} \in \mathbb{N}$; otherwise, take $\left\lceil \frac{B}{1-p} \right\rceil$. Recalling that $m = N \cdot B$, we have:

$$\begin{aligned}
\binom{t'}{B}\binom{m}{B}^{-1} &< \binom{\frac{B}{1-p}}{B}\binom{N \cdot B}{B}^{-1} \\
&= \frac{(\frac{B}{1-p})(\frac{B}{1-p} - 1)\cdots(\frac{B}{1-p} - B + 1)}{(N \cdot B)(N \cdot B - 1)\cdots(N \cdot B - B + 1)} \\
&\leq \frac{(\frac{B}{1-p})(\frac{B}{1-p})\cdots(\frac{B}{1-p})}{(N \cdot B)(N \cdot B)\cdots(N \cdot B)} \\
&= (N - Np)^{-B},
\end{aligned}$$

where the inequality from the second to third line holds because $\frac{B}{1-p} \leq N \cdot B$ (by the assumption in the theorem that $N \geq \frac{1}{1-p}$). The inequality thus follows from the fact that $\frac{a-1}{b-1} \leq \frac{a}{b}$ if and only if $a \leq b$. Thus:

$$\begin{aligned}
\alpha(t', N, B, p) &< p^{t'} \cdot N \cdot (N - Np)^{-B} \\
&= p^{\frac{B-1+p}{1-p}} \cdot N \cdot (N - Np)^{-B} \\
&= 2^{\log p \cdot \frac{B-1+p}{1-p} + \log N - B \log(N - Np)} \\
&= 2^{\frac{B \log p}{1-p} - \log p + \log N - B \log(N - Np)}.
\end{aligned}$$

If $B \geq \frac{s + \log N - \log p}{\log(N-Np) - \frac{\log p}{1-p}}$ then $B \log(N - Np) - \frac{B \log p}{1-p} \geq s + \log N - \log p$, and so

$$\frac{B \log p}{1-p} - \log p + \log N - B \log(N - Np) \leq -s.$$

This implies that $\alpha(t', N, B, p) < 2^{-s}$. Since $\Pr[\mathsf{Game}_1(\mathcal{A}, N, B, p) = 1] \leq \alpha(t, N, B, p) \leq \alpha(t', N, B, p)$ for all t, we conclude that for every Adv, $\Pr[\mathsf{Game}_1(\mathcal{A}, N, B, p) = 1] < 2^{-s}$.

■

A simple asymptotic bound. The following corollary is obtained by simply plugging in $p = \frac{1}{2}$ to Theorem 4.1 (we present this to obtain a clearer asymptotic bound, but warn that $p = \frac{1}{2}$ is typically *not* the best choice).

Corollary 4.2. *Let s, B and N be as above, and consider the case that $p = \frac{1}{2}$. If $B \geq \frac{s}{\log N + 1} + 1$, then for every adversary Adv it holds that $\Pr\left[\mathsf{Game}_1\left(\mathcal{A}, B, N, \frac{1}{2}\right) = 1\right] < 2^{-s}$.*

Directly computing the probability. In the analysis of Theorem 4.1 we prove a general upper bound of the function

$$\frac{\binom{M-t}{m-t}}{\binom{M}{m}} \cdot N \cdot \binom{t}{B}\binom{m}{B}^{-1} \tag{5}$$

that bounds the probability that the adversary succeeds in cheating. However, it is possible to directly compute the probability in Eq. (5) for concrete sets of parameters, in order to obtain slightly better parameters. For example, Theorem 4.1 states that for $s = 40$, $N = 1024$ and $p = 0.7$, B should be 6. However, by analytic calculation, for this set of parameters we actually have that the maximal cheating probability is at most $2^{-51.07}$. If we take $B = 5$ we have that the maximal cheating probability is at most $2^{-40.85}$. This means that instead of using $\frac{1024\times 6}{0.7} = 8778$ balls, we can use only $\frac{1024\times 5}{0.7} = 7315$ balls for the same p and N! This "gap" is significant even for smaller values of N. For parameters $s = 40$, $N = 32$ and $p = 0.75$, Theorem 4.1 requires B to be 10. The maximum of Equation 5 for these parameters is at most 2^{-44}, which, again, is much smaller than the 2^{-40} bound given by Theorem 4.1. In fact, if we take $N = 32$, $p = 0.8$ and $B = 10$, we get that the maximum of Equation 5 is at most $2^{-40.1}$, without increasing B as required if we had used Theorem 4.1 with $p = 0.8$. This reduces the expected number of balls per bucket from 13.34 (for $p = 0.75$) to only 12.5 (for $p = 0.8$).

We leave further optimizations and analysis of the above bounds for future work, and recommend computing analytically the exact bounds based on the above analysis whenever performance is critical. See further examples of the parameters of Theorem 4.1 in Section 4.3.

4.2 The Majority-Cracked Bucket Game

Let Game 2 be the same game as Game 1, but where Adv wins if P_2 is left with a bucket that consists of at least $\frac{B}{2}$ cracked balls. Define that $\mathsf{Game}_2(\mathcal{A}, N, B, p) = 1$ if and only if P_2 does not abort the game and there is a majority-cracked bucket; for simplicity, a bucket that is exactly half cracked and half not is also called "majority cracked". (Recall that $\mathsf{Game}_1(\mathcal{A}, N, B, p) = 1$ only if *all* of the balls in some bucket are cracked.)

We prove the following theorem:

Theorem 4.3. *Let s be a security parameter, and let B, $N \in \mathbb{N}$ $p \in (0, 1)$ be as above. If*

$$B \geq \frac{2s + 2\log N - \log(-1.25 \log p) - 1}{\log N + \log(-1.25 \log p) - 2} \ ,$$

then for every adversary Adv *it holds that* $\Pr[\,\mathsf{Game}_2(\mathcal{A}, N, B, p) = 1\,] < 2^{-s}$.

Proof. Let t be the number of cracked balls that Adv sends. Let E_{na} denote the event that P_2 does not abort (given that t balls are cracked), and let E_{mc} be the event that given m balls of which t of them are cracked, the mapping π results in a majority-cracked bucket. We have that

$$\Pr[\,\mathsf{Game}_2(\mathcal{A}, N, B, p) = 1\,] \ = \ \Pr[\,E_{na} \wedge E_{mc}\,] = \Pr[\,E_{na}\,] \cdot \Pr[\,E_{mc}\,]$$

since the two events are independent.

We already know that $\Pr[\,E_{na}\,] \leq p^t$ from Eq. (3), thus we only need to analyse $\Pr[\,E_{mc}\,]$. Fix a bucket. The probability that there are $t \geq t' \geq \lceil B/2 \rceil$ cracked balls in that bucket is equal to the probability that $t' \geq \lceil B/2 \rceil$ cracked balls are chosen to go into the fixed bucked, and then all other balls are chosen at random. Thus, we have that the probability equals:

$$\binom{t}{t'}\binom{m-t}{B-t'}\binom{m}{B}^{-1},$$

which for $B \geq t' \geq \lceil B/2 \rceil$ and $t' \leq t$ is equal to

$$\begin{aligned}
&\prod_{i=0}^{t'-1}\frac{t-i}{t'-i}\cdot\prod_{i=0}^{B-t'-1}\frac{m-t-i}{B-t'-i}\cdot\prod_{i=0}^{B-1}\frac{B-i}{m-i}\\
&\qquad=\prod_{i=0}^{t'-1}\frac{t-i}{t'-i}\cdot\prod_{i=0}^{B-t'-1}\frac{m-t-i}{B-t'-i}\cdot\prod_{i=0}^{B-t'-1}\frac{B-t'-i}{m-t'-i}\cdot\prod_{i=0}^{t'-1}\frac{B-i}{m-i}\\
&\qquad\leq\prod_{i=0}^{t'-1}\frac{t-i}{t'-i}\cdot\prod_{i=0}^{t'-1}\frac{B-i}{m-i}\\
&\qquad=\prod_{i=0}^{t'-1}\frac{B-i}{t'-i}\cdot\prod_{i=0}^{t'-1}\frac{t-i}{m-i}\\
&\qquad=\binom{B}{t'}\cdot\prod_{i=0}^{t'-1}\frac{t-i}{m-i}\quad\leq\quad\binom{B}{t'}\left(\frac{t}{m}\right)^{t'}
\end{aligned}$$

where the last inequality holds since $t \leq m$ (otherwise, $\Pr[E_{na}] = 0$ and this part of the analysis is not relevant). Thus, the probability that there are $t' \geq \lceil B/2 \rceil$ in the given bucket is at most

$$\sum_{t'=\lceil B/2 \rceil}^{B}\binom{B}{t'}\left(\frac{t}{m}\right)^{t'} \leq 2^{B-1}\left(\frac{t}{m}\right)^{\lceil B/2 \rceil}.$$

By union bound, the probability that *some* bucket is majority-cracked is

$$\Pr[E_{mc}] \leq N \cdot 2^{B-1}\left(\frac{t}{m}\right)^{\lceil B/2 \rceil}.$$

and thus, we get that $\Pr[\mathsf{Game}_2(\mathcal{A}, N, B, p) = 1] \leq p^t \cdot N \cdot 2^{B-1}\left(\frac{t}{m}\right)^{\lceil B/2 \rceil}$.

Let $\alpha(t, N, B, p) = p^t \cdot N \cdot 2^{B-1}\left(\frac{t}{m}\right)^{\lceil B/2 \rceil} = 2^{t\log p+\log N+B-1-\log(N)\lceil B/2 \rceil}\cdot\left(\frac{t}{B}\right)^{\lceil B/2 \rceil}$ (using the fact that $m = NB$ for the second equality). This function is maximized at $\tilde{t} = \dfrac{\lceil B/2 \rceil}{-\log p \ln 2}$, and thus is at most

$$\begin{aligned}
&2^{-\frac{\lceil B/2 \rceil}{\ln 2}+\log(N)(1-\lceil B/2 \rceil)+B-1}\cdot\left(\frac{\lceil B/2 \rceil}{-\log p\ln(2)B}\right)^{\lceil B/2 \rceil}\\
&\qquad<\quad 2^{-B/2+\log(N)(1-B/2)+B-1}\cdot\left(\frac{B+1}{-1.25\log(p)B}\right)^{(B+1)/2}\\
&\qquad\leq\quad 2^{-B/2+\log(N)(1-B/2)+B-1}\cdot\left(\frac{2}{-1.25\log(p)}\right)^{(B+1)/2}\\
&\qquad\leq\quad 2^{\log(N)(1-B/2)+B/2-1}\cdot 2^{(B+1)/2-\log(-1.25\log(p))(B+1)/2}
\end{aligned}$$

for $p \in (0,1)$ and $B \geq 1$.

By taking $B \geq \dfrac{s + \log N - \log(-1.25 \log p)/2 - 1/2}{\log N/2 + \log(-1.25 \log p)/2 - 1}$ we have that this probability is less than 2^{-s}. Since $\Pr\left[\,\mathsf{Game}_2(\mathcal{A}, N, B, p) = 1\,\right] \leq \alpha(t, N, B, p) \leq \alpha(\tilde{t}, N, B, p)$ for all t, we conclude that for every Adv, $\Pr\left[\,\mathsf{Game}_2(\mathcal{A}, N, B, p) = 1\,\right] < 2^{-s}$.

■

A simple asymptotic bound. As in the game for a fully-cracked bucket, a simple asymptotic bound (that is not concretely optimal) can be obtained by plugging in $p = \frac{1}{2}$ to Theorem 4.3:

Corollary 4.4. *Let s, B and N be as above, and consider the case that $p = \frac{1}{2}$. If $B \geq \frac{2s+3}{\log N - 2} + 1$, then for every adversary* Adv *it holds that* $\Pr\left[\,\mathsf{Game}_2\left(\mathcal{A}, B, N, \frac{1}{2}\right) = 1\,\right] < 2^{-s}$.

Directly computing the probability. In the analysis of Theorem 4.3 we prove a general upper bound of the function

$$\frac{\binom{M-t}{m-t}}{\binom{M}{m}} \cdot N \cdot 2^{B-1} \left(\frac{t}{m}\right)^{\lceil B/2 \rceil} \tag{6}$$

As shown for Theorem 4.1, by directly computing the probability of Eq. (6) for concrete values we can obtain better parameters.

For example, Theorem 4.3 requires that for $N = 1024$ and $p = 0.75$, B should be 15. However, by analytic calculation, for this set of parameters we actually have that the maximal cheating probability is at most $2^{-60.42}$. If instead we take $B = 12$ we have that the maximal cheating probability is at most $2^{-42.87}$. Thus, instead of using $\frac{1024 \times 15}{0.75} = 20,480$ circuits in total, only $\frac{1024 \times 12}{0.75} = 16,384$ circuits are needed.

Likewise, for parameters $N = 32$ and $p = 0.5$, Theorem 4.3 requires B to be 27. However, the cheating probability according to Eq. (6) for these parameters is at most $2^{-65.22}$. If we take B to be 18 we have that the maximal cheating probability is at most $2^{-40.32}$. Thus, it suffices to use only $\frac{32 \times 18}{0.5} = 1,152$ circuits in total instead of $\frac{32 \times 27}{0.5} = 1,728$.

4.3 Concrete Examples

Games with fully-cracked buckets. See Tables 2–4 in Appendix A for several concrete examples for the numbers of Theorem 4.1 with $s = 40$. We can see that the bigger p is, the smaller the total number of balls is (up to a certain limit). However, the number of balls in each bucket grows proportionally with p. This is relevant for the online/offline setting, since by using p it is possible to obtain a tradeoff between online and offline time. Specifically, a higher p means less circuits overall but more circuits in the online stage (where each bucket is evaluated), thereby reducing the offline time at the expense of increasing the online time. Conversely, a lower p means more circuits in the offline stage and smaller bucket and so less computation in the online stage.

Note that since a ceiling is used when computing B, the dependences are not smooth, and the parameters should be fine-tuned depending on the required s and N. Concretely, as can be seen in Tables 2, for 2^{10} executions, the best values to take are:

1. *Bucket of size 4:* take $p = 0.1$, yielding 40960 overall circuits (and so an average of 40 circuits overall per execution). This is no better than a single execution regarding the number of circuits, but has the advantage that there is a very low online time.

2. *Bucket of size 5:* take $p = 0.65$, yielding 7877 overall circuits (and so an average of 7.69 circuits overall per execution).

3. *Bucket of size 6:* take $p = 0.85$, yielding 7229 overall circuits (and so an average of 7.06 circuits overall per execution).

It is instructive to also consider the case of 2^{20} executions; see Table 3. In this case, we have that the best parameters are:

1. *Bucket of size 3:* take $p = 0.65$, yielding 4839582 overall circuits (and so an average of 4.62 circuits overall per execution)

2. *Bucket of size 4:* take $p = 0.98$, yielding 4279903 overall circuits (and so an average of 4.08 circuits overall per execution

Games with majority-cracked buckets. See Tables 5–7 in Appendix A for several concrete examples for the numbers of Theorem 4.3 with $s = 40$. We can see that the effect of p on B and the total number of balls is similar to those dependences in Game 1, although the concrete numbers are different. For $N = 1024$ and $p = 0.7$, only 20 garbled circuits are needed on average per execution (as opposed to 125 in the cut-and-choose of [Lin13]) and only 14 circuits are used in the online stage. For larger values of N, these numbers decrease significantly, e.g. for $N = 1048576$ and $p = 0.9$ only 8.89 circuits are needed on average per execution, where only 8 are used in the online stage. In addition, we get a significant improvement over the cut-and-choose of [Lin13] also for small values of N, e.g., for $N = 32$ and $p = 0.6$, only 51.69 circuits are needed on average per execution (which is less than half than needed in [Lin13]).

5 Tweaked Batch Cut-and-Choose OT

To simplify the description of our protocols, we define an adaptation of the cut-and-choose OT protocol of [LP11] to the multiple executions setting, where both parties have more than one input.

5.1 The Functionality $\mathcal{F}_{tcot}$

In Figure 5.1 we define the ideal functionality for *tweaked batch cut-and-choose OT* that receives inputs from sender S and receiver R and returns a message to each of them. The functionality is parameterized by integers B, N, M and l.

Although $\mathcal{F}_{tcot}$ is a stand-alone functionality, we explain the "meaning" behind the parameters B, N, M, l and the other elements of the functionality, in the context of Yao-based secure computation in the batch setting. Regarding the parameters: N is the number of 2PC executions the parties would like to run, B and M are chosen according to Theorems 4.1 and 4.3, and l is the bit length of party P_2's input to the secure computation. Recall that B is the bucket size, which translates to the number of circuits actually evaluated for every execution, and M is the total number of circuits sent (including check and evaluation circuits). Note that according to this, the number of circuits checked is $M - NB$, and the remaining NB circuits are randomly mapped to N bins of size B each.

In the functionality description in Figure 5.1, the sender S has input pairs $(\mu^0_{j,i}, \mu^1_{j,i})$, for every $j = 1, \dots, M$ and $i = 1, \dots, l$. In the protocol for 2PC, $(\mu^0_{j,i}, \mu^1_{j,i})$ is the pair of garbled values (keys) on the ith input wire in the jth circuit (recall that M is the overall number of circuits, and

l is the number of input bits). In addition, the sender S has a series of input strings $s_1, \ldots, s_M$. In the protocol for 2PC, s_j is the "random seed" used for generating the entire jth circuit.

The receiver R inputs N strings $y_1, \ldots, y_N$ where y_i is its input in the ith 2PC execution. In addition, it inputs a set $J \subset [M]$ where the jth circuit is opened to be checked for every $j \in J$. Finally, R inputs a mapping function π that maps the remaining NB circuits to N buckets of size B. In the output, R receives the "opening" to all the circuits $j \in J$, and receives the garbled values associated with the input y_i for all the circuits (from $j \notin J$) in the ith bucket. The opening of the jth circuit for $j \in J$ includes s_j (the seed for generating the jth circuit) and the set $\{(\mu^0_{j,i}, \mu^1_{j,i})\}_{i=1}^{l}$. Given these values, R can verify that the jth circuit was correctly constructed and that the input garbled values given by S to this circuit are correct.

The above description covers the case that the parties send valid inputs to the functionality. However, a corrupted R may attempt to input a set of indices J of a different size, or otherwise cheat. In order to make our protocol for $\mathcal{F}_{tcot}$ more efficient, we do not prevent such cheating and only catch it at the end of the protocol. In this case, a corrupted R may learn more than it is allowed. However, since it will be caught, this makes no difference to the security of the 2PC protocol that uses $\mathcal{F}_{tcot}$. We capture this by enabling a corrupted R to send a special GetAll message; if it does so then it receives all of S's input. However, S receives $\perp$ as output and thus knows that R has cheated and so aborts the entire 2PC protocol, with no damage done.

FIGURE 5.1 (The $\mathcal{F}_{tcot}$ Functionality)**.**

Public parameters: The values M, B, N, l are publicly known parameters.
Inputs:

- S inputs k-bit strings $s_1, \ldots, s_M$, and M sets of l pairs $\left\{(\mu^0_{j,i}, \mu^1_{j,i})\right\}_{i=1}^{l}$, for $j = 1, \ldots, M$.
- R inputs N strings $y_1, \ldots, y_N$ of length l. In addition, it inputs a set of indices $J \subset [M]$ of size $M - NB$ and a B-to-1 mapping function $\pi : [NB] \to [N]$.

  If R inputs anything else (e.g., J of a different size) or inputs GetAll, then $\mathcal{F}_{tcot}$ sends all of S's input to R, sends $\perp$ to S, and halts. Otherwise, $\mathcal{F}_{tcot}$ proceeds to Outputs phase.

Outputs:

- Set $c = 1$. For $j = 1, \ldots, M$:
  - If $j \in J$, functionality $\mathcal{F}_{tcot}$ sends R the string s_j and the entire set $\left\{(\mu^0_{j,i}, \mu^1_{j,i})\right\}_{i=1}^{l}$.
  - If $j \notin J$, functionality $\mathcal{F}_{tcot}$ sets $Y = y_{\pi(c)}$ and sends R the set $\left\{\mu^{Y_i}\right\}_{i=1}^{l}$, where Y_i is the ith bit of Y. In addition, $\mathcal{F}_{tcot}$ sets $c = c + 1$.
- If R is corrupted and sends $\mathcal{F}_{tcot}$ the string abort then $\mathcal{F}_{tcot}$ sends $\perp$ to S and halts. Otherwise, if R sends continue then $\mathcal{F}_{tcot}$ proceeds to the next step.
- Functionality $\mathcal{F}_{tcot}$ sends S the set J and mapping π.

5.2 Constructing $\mathcal{F}_{tcot}$

Let $(\mathbb{G}, g, q)$ be such that $\mathbb{G}$ is a group of order q, with generator g. We define the function $RAND(w, x, y, z) = (u, v)$, where $u = (w)^t \cdot (y)^{t'}$ and $v = (x)^t \cdot (z)^{t'}$, and the values $t, t' \in_R \mathbb{Z}_q$ are random. In Figure 5.2 we show how to realize $\mathcal{F}_{tcot}$. When r is used without subscript, it denotes an independent random value that is not referred to in other steps of the protocol. We denote the bits of the ith input string of R by $y_i = y_{i,1}, \ldots, y_{i,l}$.

PROTOCOL 5.2 (Protocol for Securely Realizing $\mathcal{F}_{tcot}$)**.**

Initialization phase:

1. R chooses a random $r \in \mathbb{Z}_q$, sets $g_0 = g$ and $g_1 = g^r$ and sends g_1 to S. In addition, it proves in zero-knowledge that it knows the discrete log of g_1, relative to g_0.
2. S chooses a random $r \in \mathbb{Z}_q$, sets $h = g^r$ and sends h to R. In addition, it proves in zero-knowledge that it knows the discrete log of h, relative to g.
3. R sends $\mathsf{PCommit}_h(\pi)$ and proves in zero-knowledge that it knows the corresponding decommitment. (To reduce cost, instead of using a truly random mapping π, R picks a PRF seed $seed_\pi$ and uses $\mathrm{PRF}_{seed_\pi}(\cdot)$ to deterministically compute a mapping function π. Then, the commitment is only to $seed_\pi$. Recall also that $\mathsf{PCommit}_h$ is a Pedersen commitment using generators g, h.)
4. For $j = 1, \ldots, M$, R chooses a random t_j and sets $h_{j,0} = (g_0)^{t_j}$. If $j \in J$, it sets $h_{j,1} = (g_1)^{t_j}$. Otherwise, it sets $h_{j,1} = (g_1)^{t_j+1}$. R sends $(h_{j,0}, h_{j,1})$ and proves in zero-knowledge that it knows t_j (i.e., $\log_{g_0} h_{j,0}$). (These proofs can be batched, as described in [LP11].)

Transfer phase:

1. R sets $c = 1$. For $j = 1, \ldots, M$ and $i = 1, \ldots, l$, R chooses a random $r_{j,i} \in \mathbb{Z}_q$ and sends $(G_{j,i}, H_{j,i}) = ((g_{y_{\pi(c),i}})^{r_{j,i}}, (h_{j,y_{\pi(c),i}})^{r_{j,i}})$ if $j \notin J$, and $(G_{j,i}, H_{j,i}) = ((g_0)^{r_{j,i}}, (h_{j,0})^{r_{j,i}})$ otherwise.
2. The players execute M 1-out-of-2 OTs where in the jth OT, S inputs s_j and a random $\tilde{s}_j \in \{0,1\}^k$, and R inputs 0 if $j \in J$ and 1 otherwise. (*Any* OT can be used here, including OT extensions.)
3. For $j = 1, \ldots, M$ and $i = 1, \ldots, l$, the sender S operates in the following way:
   (a) Sets $(u^0_{j,i}, v^0_{j,i}) = RAND(g_0, G_{j,i}, h_{j,0}, H_{j,i})$ and $(u^1_{j,i}, v^1_{j,i}) = RAND(g_1, G_{j,i}, h_{j,1}, H_{j,i})$.
   (b) Sends $(u^0_{j,i}, w^0_{j,i})$ and $(u^1_{j,i}, w^1_{j,i})$, where $w^0_{j,i} = \mathsf{REHash}(v^0_{j,i}) \oplus \mu^0_{j,i}$ and $w^1_{j,i} = \mathsf{REHash}(v^1_{j,i}) \oplus \mu^1_{j,i}$.

Check for malicious behavior phase:

1. R decommits the value of π and sends J. If π is not a valid mapping or J is not of size $M - NB$ then S outputs $\perp$ and halts. Otherwise, it proceeds to the next step.
2. For $j = 1, \ldots, M$, R proves in zero-knowledge that $(g_0, g_1, h_{j,0}, h_{j,1})$ is a DDH tuple if $j \in J$, and that $(g_0, g_1, h_{j,0}, h_{j,1}/g_1)$ is a DDH tuple otherwise. In addition, it sends the values $\tilde{s}_j$ it has learnt. S verifies that everything is correct and consistent; in particular it checks that for every $j \notin J$ the correct $\tilde{s}_j$ was sent.
3. Let $\phi : [M] \to [NB]$ be the mapping function that maps an index $j \notin J$ to the value of $\pi(c)$ for the c used with that j in the transfer phase. (Both parties know it given π and J.)
4. For $n = 1, \ldots, N$,
   (a) Let $E_n = \{j \mid \phi(j) = n\}$.
   (b) R proves in zero-knowledge that all $\{(g_0, G_{j,i}, h_{j,0}, H_{j,i})\}_{j \in E_n}$ OR all $\{(g_1, G_{j,i}, h_{j,1}, H_{j,i})\}_{j \in E_n}$ are DDH tuples, for $i = 1, \ldots, l$.

Output computation phase:

1. For $j = 1, \ldots, M$ and $i = 1, \ldots, l$, R's output is as following:
   (a) If $j \in J$, R outputs the pairs $\left(w^0_{j,i} \oplus \mathsf{REHash}\left((u^0_{j,i})^{r_{j,i}}\right), w^1_{j,i} \oplus \mathsf{REHash}\left((u^1_{j,i})^{r_{j,i}t_j}\right)\right)$. In addition, it outputs the value s_j that it obtained (for every $j \in J$.
   (b) If $j \notin J$, R outputs the values $w^{y_{\phi(j),i}}_{j,i} \oplus \mathsf{REHash}\left((u^{y_{\phi(j),i}}_{j,i})^{r_{j,i}}\right)$.
2. S outputs π and J.

Theorem 5.3. *Assume that the Decisional Diffie-Hellman problem is hard in the group $\mathbb{G}$ and that* $\mathsf{REHash}(\cdot)$ *is collision-resistant and a suitable randomness extractor. Then, the protocol of Figure 5.2 securely computes $\mathcal{F}_{tcot}$.*

Proof (sketch). The proof follows the proofs of [LP11, Proposition 3.9] and [Lin13, Theorem 6.4], and thus we only give here a proof sketch.

We prove security in a hybrid model where 1-out-of-2 OT, zero-knowledge proofs and proofs of knowledge (ZKPoK) are computed by ideal functionalities $\mathcal{F}_{ot}, \mathcal{F}_{zkp}, \mathcal{F}_{zkpok}$ (where the prover sends $(x; w)$ and the functionality sends 1 to the verifier if and only if $(x; w)$ is in the relation). We separately prove the case that S is corrupted and the case that R is corrupted.

S is corrupted. During the setup phase, the simulator Sim extracts $\log_g h$ from the zero-knowledge proofs and commits to zero (instead of sending $\mathsf{PCommit}_h(\pi)$). Then, it chooses $h_{j,0}, h_{j,1}$ so that all $(g_0, g_1, h_{j,0}, h_{j,1})$ are DDH tuples.

In the transfer phase, Sim uses $J = [M]$ and extracts all of S's inputs to the OTs. In addition, it extracts all of the s_j and $\tilde{s}_j$ values input by S to $\mathcal{F}_{ot}$. Sim sends all these values to the ideal functionality as the sender's inputs.

After receiving from the ideal functionality the values of J and π, Sim decommits to π (by utilizing the fact that it knows $\log_g h$ and can decommit to anything), and cheats in the ZKPoKs in a way that is consistent with J and π.

Note that the simulation differs from the real execution in two ways: **(1)** Sim commits to zero (instead of $\mathsf{PCommit}_h(\pi)$); **(2)** Sim picks all tuples $(g_0, g_1, h_{j,0}, h_{j,1})$ to be DDH tuples. However, since $\mathsf{PCommit}_h(\cdot)$ is perfectly-hiding commitment, the commitment is distributed identically to the real execution. The second part is computationally-indistinguishable assuming the DDH assumption holds. (Note that [PVW08] shows that once $(g_0, g_1, h_{j,0}, h_{j,1})$ is a DDH tuple, S's OT queries carry no information about its input, information-theoretically.)

R is corrupted. During the initialization phase, the simulator Sim extracts π and all t_j-s from the ZKPoK, allowing it to determine if $(g_0, g_1, h_{j,0}, h_{j,1})$ is a DDH tuple or not. Sim sets the set of indexes in which these tuples are DDH tuples to be the set J.

In the transfer phase, Sim extracts R's inputs by utilizing its knowledge of t_j. In addition, Sim extracts R's inputs to $\mathcal{F}_{ot}$. In particular, it learns the set of indexes in which R inputs 0. Let this set be J'. If the inputs are not all consistent with honest behavior (with some π, J and $y_1, \ldots, y_N$), then Sim sends GetAll to the ideal functionality and receives back all of S's inputs. It then completes the simulation honestly, using S's real inputs.[8] Otherwise, Sim sends π, J and $y_1, \ldots, y_N$ to the ideal functionality, and receives its outputs. Sim sends to R the OT answers with those values. Note that in the OT answers for the s_j and $\tilde{s}_j$ OTs, Sim sends s_j as received from $\mathcal{F}_{tcot}$ for every $j \in J$ where R input 0; if R input 1 then Sim sends a random $\tilde{s}_j$. For $j \notin J$, Sim always sends a random $\tilde{s}_j$; in this case, we have that R must have used input 1 as described above. (Note that for $j \notin J$, the OT values that R choose not to learn are information-theoretically hidden once $(g_0, g_1, h_{j,0}, h_{j,1})$ is not a DDH tuple.)

[8]The only exception is that if for some $j \in J$, R uses input 1 to the 1-out-of-2 OT and learns $\tilde{s}_j$ instead of s_j then this is ignored by Sim as cheating. We stress that the opposite case, where R uses input 0 when $j \notin J$, is dealt with by Sim sending GetAll to $\mathcal{F}_{tcot}$.

Last, in the outputs phase, Sim checks R's proofs and $\tilde{s}_j$ values. If any of them fail or are incorrect, then Sim sends abort to the ideal functionality. Otherwise, it sends continue. Then, Sim outputs whatever Adv outputs.

Note that in case $J \neq J'$, R must fail (with overwhelming probability) in some of the proofs of the outputs phase. Also, note that R is committed to π or else (using the ZKPoK extraction), Sim obtains two decommitments for the same commitment, which breaks the security of the Pedersen commitment. Beside these differences, the only step in which the simulation is different from the real execution is for the inputs Sim uses in the transfer phase. However, as shown in [PVW08], for non DDH tuples (i.e., for $j \notin J$) this difference is indistinguishable. Finally, note that if for some $j \notin J$, a corrupted R learns s_j (instead of $\tilde{s}_j$), then except with negligible probability it will not be able to send the correct $\tilde{s}_j$ and S will abort. (Observe that the opposite case where R learns $\tilde{s}_j$ for $j \in J$ instead of s_j is of no consequence, since it just means that R receives "less output". Formally, this is indistinguishable to R since Sim also uses random $\tilde{s}_j$ for these OT inputs, just like a real S.)

■

5.3 Optimization for Our Protocols

In our protocols we need P_1 (i.e. the sender in $\mathcal{F}_{tcot}$) to send the garbled circuits *before* knowing π and J. However, in order to be able to simulate the garbled circuits (in the standard model), the OTs must be executed *before* the garbled circuits are sent. (This enables the simulator in the full 2PC protocol to send different circuits for those to be evaluated and those to be checked).

One solution for this issue is to ask P_1 to commit on all the garbled circuits using a trapdoor commitment (e.g., $\mathsf{PCommit}_h(\cdot)$) before calling $\mathcal{F}_{tcot}$, and then send the garbled circuits only after it (when π and J are already known to the P_1/simulator).

An alternative solution is to slightly modify $\mathcal{F}_{tcot}$ so that in the transfer phase, S inputs an additional message η (in plain) that R receives as is. Indeed, this modification seems less modular then the first solution, but it reduces the cost of committing (and decommitting) on all garbled circuits, and thus, in practice, is preferable. In the protocol, the sender S sends this message between Steps 2 and 3 of the transfer phase in the protocol. Observe that at this point, S does not yet know π and J, but the simulator (in the proof of security) has already learned R's inputs and also knows if R is cheating (in which case it just sends GetAll). Thus, this achieves the needed requirements. In the simulation of the protocol, this is not a problem since if S is corrupted, then Sim receives η from the corrupted S before it needs to provide output to S (thus it can send η to the trusted functionality as part of S's input). Likewise, if R is corrupted, then Sim obtains R's input before it has to provide η. Thus, Sim can send the corrupted R's input before having to provide η in the protocol simulation.

6 Secure Two-Party Computation in the Batch Setting

In this section we describe our protocols for batch execution of many 2PC computations. We start with the main protocol, which is based on standard assumptions, and then show how to improve its efficiency by using less standard assumptions.

6.1 The Main Protocol

Say that P_1 has inputs $x_1, \dots, x_N$ and P_2 has inputs $y_1, \dots, y_N$ and they wish to compute the output of $f(x_1, y_1), \dots, f(x_N, y_N)$. We assume that only P_2 should learn the output. The techniques of [LP07, MR13, SS13] can be used to transform our protocol into one in which both parties learn the output (or possibly different outputs).

See Protocol 6.1 for a full description of our protocol for this setting (intuition appears in Section 2.2). We use the two optimizations discussed in Sections 5.2 and 5.3, so that P_2 generates π using a PRF seed, and that P_1 sends the garbled circuits in the transfer phase of $\mathcal{F}_{tcot}$. When r is used without subscript in the protocol description, it denotes an independent random value that is not referred to in other steps of the protocol.

PROTOCOL 6.1 (Batch Two-Party Computation of $\vec{f}(\vec{x}, \vec{y}) = (f(x_1, y_1), \dots, f(x_N, y_N))$).

Setup:

1. The parties agree on circuits $C(x, y)$ and $C'(x, D, d)$ as described in Section 2.2, and parameters s, N, p and p'. Denote P_1's input in the jth execution by x_j, and P_2's input by y_j.
2. P_1 prepares the set $\left\{\left(i, g^{a_i^0}, g^{a_i^1}\right)\right\}_{i=1,\dots,|x|}$ (as in [Lin13]) and sends it to P_2 with a random $h \in \mathbb{G}$.

Cut-and-choose for C**:**

1. Let B be according to Theorem 4.1 (for parameters s, N, p) and let $M = \frac{NB}{p}$. (Assume for simplicity that no rounding of M is needed.)
2. P_2 chooses:
   (a) *The cut:* P_2 sets J to be a random subset of $[M]$ that includes exactly $M - NB$ elements.
   (b) *The mapping:* P_2 picks a PRF seed $seed_\pi$ and uses $\text{PRF}_{seed_\pi}(\cdot)$ to compute a mapping function $\pi : [N \cdot B] \to [N]$ that maps exactly B elements to each bucket.
3. For $j = 1, \dots, M$,
   (a) P_1 picks a PRF seed $seed_j$. All the randomness needed in the next two steps is derived from $\mathsf{PRF}_{seed_j}(\cdot)$.
   (b) P_1 picks r_j and prepares a garbled circuit gc_j for C, in which the labels of P_1's input wire i are $\mathsf{REHash}\left(g^{a_i^0 r_j}\right)$ and $\mathsf{REHash}\left(g^{a_i^1 r_j}\right)$ (as in [Lin13]).
   (c) P_1 computes a commitment lc_j on all the output wire labels of gc_j.
4. P_1 acts as the sender in $\mathcal{F}_{tcot}$ and P_2 as the receiver.
   (a) P_2 inputs $y_1, \dots, y_N$, $seed_\pi$ and J.
   (b) P_1 inputs the wire labels of P_2's input (i.e., in the jth set it inputs $|y|$ pairs of keys) and the seeds $seed_1, \dots, seed_M$. In addition, as described in Section 5.3, P_1 also inputs to $\mathcal{F}_{tcot}$ a message η that consists of $(g^{r_1}, gc_1, lc_1), \dots, (g^{r_M}, gc_M, lc_M)$ (in plain).
   (c) P_2 receives η. In addition, for every $j \in J$, P_2 receives $seed_j$ and the entire jth set of labels; for every $j \notin J$, P_2 receives the labels associated with $y_1, \dots, y_N$ according to π (see Figure 5.1).
   (d) P_1 receives J and $seed_\pi$ (and constructs π).
5. For every $j \in J$, P_2 checks that the tuple (g^{r_j}, gc_j, lc_j) is constructed properly from $seed_j$, and consistently with the outputs of $\mathcal{F}_{tcot}$.
6. Denote the remaining garbled circuits according to their placement by π, i.e. let $gc_{j,i}$ be the ith circuit of the jth bucket (for $j = 1, \dots, N$ and $i = 1, \dots, B$). (From here on, j indexes the executions $1, \dots, N$, and not $J \subset [M]$.)

Protocol 6.1 – continued

Cheating recovery - step 1:

1. For $j = 1, \ldots, N$, P_1 picks $D_j \in_R \{0,1\}^k$ and sends $\mathsf{Hash}(D_j)$. In addition, for $v \in \mathsf{Outputs}(C)$,
   (a) P_1 chooses $R_{j,v} \in_R \{0,1\}^k$.
   (b) Let $W^b_{j,i,v}$ be the b-th label of wire v of $gc_{j,i}$. P_1 sends $\mathsf{Enc}_{W^0_{j,i,v}}(R_{j,v})$ and $\mathsf{Enc}_{W^1_{j,i,v}}(R_{j,v} \oplus D_j)$ for every $i \in \pi^{-1}(j)$.

Evaluations of C:

1. P_1 sends the garbled values that correspond to its inputs in all garbled circuits of C according to π; e.g., it sends the value $g^{a^1_i \cdot r_j}$ if its input to wire i of the jth circuit is one.
2. For $j = 1, \ldots, N$ (can be done in parallel),
   (a) P_2 evaluates the garbled circuits of bucket j, and then uses the output wire labels to decrypt the corresponding $R_{j,v}$ and $R_{j,v} \oplus D_j$ values (specifically in the jth bucket). In case it learns both $R_{j,v}$ and $R_{j,v} \oplus D_j$ for some output wire, it checks if the XOR of them is indeed D_j (by applying $\mathsf{Hash}(\cdot)$ and comparing with the value that P_1 has sent). If so, it sets d_j to D_j. Otherwise, it chooses a random $d_j \in_R \{0,1\}^s$.
   (b) If all evaluations (that ended) returned the same output, set z_j to be that output.

Cut-and-choose for C': The parties run the instructions for the cut-and-choose for C above, with the following modifications (we use prime when referring to elements of this cut-and-choose):

1. The players use s, N, p' and set B' according to Theorem 4.3 (instead of working with s, N, p, B).
2. The players use circuit C' (instead of C) and P_2 uses the input d_j in the jth bucket (instead of y_j).
3. Instead of computing commitments on the output labels (as in Step 3c of the "cut-and-choose for C" phase), we set lc'_j to be the set $\left\{\left(i, \mathsf{Com}(W^0_{j,i}), \mathsf{Com}(W^1_{j,i})\right)\right\}_{i \in I_D}$ where I_D is the set of wires for input D and $W^b_{j,i}$ is the bth label of the ith wire of gc'_j.

Cheating recovery - step 2: For $j = 1, \ldots, N$ (can be done in parallel),

1. P_1 sends the garbled values that correspond to its input x_j for the garbled circuits for C' in the j-th bucket. It also proves that its inputs are consistent, both for C' *and* C, using ZK (as done in [Lin13], including both proving that all inputs are consistent, and that they use the right values of of $g^{a^b_i}$).
2. P_1 sends D_j and decommits the labels that are associated with that D_j in all the garbled circuits $gc'_{j,i}$ for $i = 1, \ldots B'$ (i.e., for the corresponding commitments of $lc'_{j,i}$).
3. P_1 decommits $lc_{j,i}$ of $gc_{j,i}$ for $i = 1, \ldots B$ (i.e. revealing all output wire labels of the garbled circuits of C). P_2 checks all commitments, all the encryptions $\mathsf{Enc}_{W^0_{j,i,v}}(R_{j,v})$ and $\mathsf{Enc}_{W^1_{j,i,v}}(R_{j,v} \oplus D_j)$, and the hash $\mathsf{Hash}(D_j)$, and aborts if there is an inconsistency.
4. P_2 evaluates $gc'_{j,i}$, for $i = 1, \ldots B'$, and takes the majority output to be $\hat{x}_j$.

P_2's outputs: For $j = 1, \ldots, N$,

1. If all evaluation circuits of C in bucket j returned the same output z_j, then P_2 leaves z_j as is.
2. Else, if P_2 has learned earlier d_j such that $\mathsf{Hash}(d_j) = \mathsf{Hash}(D_j)$, then it sets $z_j = f(\hat{x}_j, y_j)$.
3. Else, let $gc_{j,i}$ be a circuit for which all the output labels that P_2 received from its evaluation were also the labels that were decommitted earlier from $lc_{j,i}$. P_2 sets z_j to be the output of $gc_{j,i}$.

P_2 outputs $z_1, \ldots, z_N$.

We prove the following theorem:

Theorem 6.2. *Assume that the Decisional Diffie-Hellman problem is hard in the group $\mathbb{G}$, that the hash function is collision resistant and a suitable randomness extractor, the commitment is perfectly binding, and the garbling circuit scheme is secure. Then, for any polynomial-time computable functionality f, Protocol 6.1 securely computes $\vec{f}(\vec{x}, \vec{y}) = (f(x_1, y_1), \ldots, f(x_N, y_N))$ in the presence of malicious adversaries.*

Proof. We prove security in a hybrid model where tweaked batch cut-and-choose OT, zero-knowledge proofs and proofs of knowledge (ZKPoK) are computed by ideal functionalities $\mathcal{F}_{tcot}, \mathcal{F}_{zkp}, \mathcal{F}_{zkpok}$. The fact that any zero knowledge proof of knowledge securely computes this functionality has been formally proven in [HL10].

We separately prove the case that P_1 is corrupted and the case that P_2 is corrupted.

P_1 is corrupted. A corrupted P_1 can cheat in a number of ways, though the main ones are the following: (1) P_1 can use some invalid garbled circuits; (2) It can try to learn information about P_2's input using selective-OT attacks; (3) It can use different inputs in the evaluated garbled circuits. The first two are taken care of using the cut-and-choose, while the third is verified using the ZK on P_1's inputs. Modulo verifying other smaller parts, once we protect from the above three attacks, P_2's output is guaranteed to be correct, and its input remains private. Note that if for some j, gc_j and lc_j are not generated properly by $seed_j$, then if j is chosen to be checked, P_1 is always caught. This holds also if the OT answers (for P_2 to learn the labels of its input wires) of that circuit are not consistent with the labels of gc_j. Thus, if no abort happens in the cut-and-choose phase, we get that for at least one circuit (or majority for C'), gc_j, lc_j and the OT inputs are correctly generated from $seed_j$.

More formally, we describe a sequence of hybrid games that ends with the simulated execution. (We mention only the differences between each consecutive hybrids.)

HYBRID$_0$: An execution of the protocol with a simulator that emulates honest P_2 with inputs $y_1, \ldots, y_N$. Since we work in an hybrid model with the ideal functionality $\mathcal{F}_{zkpok}$, the simulator can extract P_1's witnesses, and in particular, extract the witness P_1 uses for proving consistency of its inputs, and by that, learn P_1's inputs to the correctly garbled circuits. In addition, the simulator learns all P_1's inputs to $\mathcal{F}_{tcot}$.

HYBRID$_1$: We say that set j is *good* if: (1) $seed_j$ correctly derives g^{r_j}, gc_j and lc_j; (2) P_1's inputs to $\mathcal{F}_{tcot}$ for the labels of gc_j are consistent with the actual labels of gc_j.

The simulator aborts if there exists a bucket j such that none of its sets for C is good, or that the majority of the sets for C' are not good.

By the cut-and-choose stages and Theorems 4.1 and 4.3, we know that at least one of the sets of C, and that most of the sets of C' are good with probability at least 2^{-s}, thus, HYBRID$_1$ and HYBRID$_0$ are 2^{-s}-indistinguishable.

HYBRID$_2$: If there is a good set that is used for evaluation in which P_1 sends a decommitment to a different value then was generated by $seed_j$, then the simulator aborts the execution at the end of the protocol.

Note that the simulator aborts in case P_1 has successfully cheated in the commitment, thus, by assuming that $\mathsf{Com}(\cdot)$ is secure, this abort happens with only a negligible probability, and therefore HYBRID$_2$ and HYBRID$_1$ are computationally indistinguishable.

HYBRID$_3$: Let x_j be P_1's input to the jth bucket (extracted from its witness to $\mathcal{F}_{zkpok}$). The simulator emulates an honest P_2 with random inputs $y_1, \ldots, y_N$, and if the emulated P_2 does not abort, the simulator outputs $(f(x_1, y_1), \ldots, f(x_N, y_N))$ as P_2's output.

Recall that at this stage, P_1 does not get any information about y_j and d_j. Therefore, the only difference between the executions in HYBRID$_3$ and HYBRID$_2$ is in case P_2's output at the end is different. Since we know that at least one of the sets of C, and that most of the sets of C' are good, P_2's evaluations in HYBRID$_2$ of the good sets of C would output the correct output as in HYBRID$_3$. Still, this does not suffice since P_2 might get some other outputs, which leaves it with the problem of determining which output is the right one. There are two options in this case: (1) That P_2 learns d_j such that $\mathsf{Hash}(d_j) = \mathsf{Hash}(D_j)$, and, (2) That it does not learn such d_j. In the first option, P_2 would input d_j to C' and get x_j in the majority of the garbled circuits of C' (since most of them are good). In the second option, P_2 would use random d_j for the evaluation of the garbled circuits of C', but later would be able to determine the right output once P_1 deocmmits the output labels. (To see why there are no other options, let's assume that gc_1 is a good garbled circuit and gc_2 is a bad one. gc_1's output is correct. Assume that gc_2's output is different than of gc_i in the first bit. If gc_2's output label is the one that is later decommitted from lc_2, then it means that the xor of this label and the one from gc_1 is D_j, which would allow P_2 to learn this value. If gc_2's output label is not the one that is later revealed from lc_2, then P_2 would ignore the output of gc_2 and use the one of gc_1, which is the correct one again.)

Summing all up, if indeed at least one garbled circuit for C and most of the garbled circuits for C' are good, then P_2's output in HYBRID$_3$ is the right output, or an abort independently of $y_1, \ldots, y_N$. Thus, HYBRID$_3$ and HYBRID$_2$ distributed the same.

HYBRID$_4$: Instead of computing (and outputting) $f(x_1, y_1), \ldots, f(x_N, y_N)$ by itself at the end, the simulator sends $x_1, \ldots, x_N$ to the trusted third party and outputs whatever Adv outputs. (If Adv or the emulated P_2 abort the execution, the simulator sends abort to the trusted third party.) Since the only difference between HYBRID$_4$ and HYBRID$_3$ is that P_2's output is computed by the trusted third party, the two are distributed the same.

The simulator: We take the simulator to be the last simulator from above. I.e., the simulator emulates an honest P_2 for the adversary with the next modifications:

- Sets y_j and d_j to be random strings, for all $j = 1, \ldots, N$.
- Extracts Adv's inputs to $\mathcal{F}_{tcot}$ and aborts if there exists a bucket j such that none of its sets for C is good, or that the majority of the sets for C' are not good.
- Extracts Adv's input x_j from $\mathcal{F}_{zkpok}$ for all $j = 1, \ldots, N$.
- Checks Adv's commitments as done in HYBRID$_2$ and aborts if there is a commitment that is "opened" to two different inputs.
- Sends abort to the trusted third party if the emulated P_2 aborts or if Adv does.
- Sends $x_1, \ldots, x_N$ to the trusted third party and outputs whatever Adv outputs.

By the above sequence of hybrid games, we conclude that the simulated execution is $(\mu(k) + 2^{-s})$-indistinguishable from the real execution.

P_2 **is corrupted.** A corrupted P_2 can do less damage, since the only meaningful values it gets to decide upon are its inputs $y_1, \ldots, y_N$. The correctness of its OT queries is verified by $\mathcal{F}_{tcot}$, as well as the consistency of its input in each bucket. Besides those parts, the rest is secure following the security the garbling scheme in use.

As before, we describe a sequence of hybrid games that ends with the simulated execution.

HYBRID$_0$: An execution of the protocol with a simulator that emulates honest P_1 with inputs $x_1, \ldots, x_N$. Since we work in an hybrid model with ideal functionality $\mathcal{F}_{tcot}$, the simulator can extract P_2's inputs to $\mathcal{F}_{tcot}$. In particular, it extracts: (1) P_2's OT inputs; (2) The value of J; (3) The value of π; (4) All these also for the cut-and-choose of C'.

HYBRID$_1$: For all $j \notin J$, the simulator generates the j-th garbled circuit set with true randomness (and not pseudo-random randomness). Since for such j, P_2 does not get any information about $seed_j$ anyhow (since it gets only c_j from $\mathcal{F}_{tcot}$), HYBRID$_1$ and HYBRID$_0$ are computationally-indistinguishable by the security of the PRF in use.

HYBRID$_2$: The simulator uses random strings in the OT answers that P_2 chose not to learn when $j \notin J$. Since $\mathcal{F}_{tcot}$ does not send to P_2 anything about those values, HYBRID$_2$ and HYBRID$_1$ are distributed the same.

HYBRID$_3$: Before sending the garbled circuits, the simulator computes the values $z_j = f(x_j, y_j)$ for $j = 1, \ldots, N$. Then, let i be an index such that $i \notin J$ and that is mapped by π to the jth bucket. The simulator replaces the garbled circuit gc_i with a simulated garbled circuit that always output z_j. By the security of the garbling scheme (or the underlying encryption in use) HYBRID$_3$ and HYBRID$_2$ are computationally-indistinguishable.

(We remark that the above step cannot use the simulator from Section 5.2 as a black-box since, here, the garbled circuits are modified in the middle of the protocol of Figure 5.2. However, this merely means that we use that simulator in a non black-box manner, by asking it to output P_2's inputs and only then ask P_1 for its plain message.)

HYBRID$_4$: The simulator aborts if P_2 uses (in the input of $\mathcal{F}_{tcot}$ for C') input d_j such that $D_j = d_j$ for some bucket j. We argue that this abort happens with a negligible probability (that depends on the security parameter k).

Assume, towards contradiction, that this abort happens with a non-negligible probability. We show that P_2 can be used for breaking the encryption or the hash function in use. Let HYBRID$_{4,1}$ be like HYBRID$_4$, but where the simulator replaces the commitments of $lc_{j,i}$ with commitments to random strings. Clearly, the executions in HYBRID$_4$ and HYBRID$_{4,1}$ up until the call to $\mathcal{F}_{tcot}$ for C' are run are indistinguishable by the security of the commitment. Let HYBRID$_{4,2}$ be like HYBRID$_{4,2}$, but where the simulator (who knows at this point already the output labels of $gc_{j,i}$-s) replaces the "cheating-recovery" encryptions of the labels that P_2 is not supposed to learn with encryptions of random strings. Since P_2 does not learn those output labels (because of the authenticity property of the garbled scheme), its view in HYBRID$_{4,1}$ and HYBRID$_{4,2}$ are indistinguishable by the security of the encryption (again, up until the call to $\mathcal{F}_{tcot}$ for C'). Next, let HYBRID$_{4,3}$ be like HYBRID$_{4,2}$, but where instead of picking D_j by itself, the simulator asks the challenger of the one-wayness of $\mathsf{Hash}(\cdot)$ for a challenge, and sends it as $\mathsf{Hash}(D_j)$. (Note that in the opened "cheating-recovery" encryptions, the messages are random anyhow.) Then, the simulator checks if $\mathsf{Hash}(d_j)$ equals to the challenge. Obviously, if it equals, then the simulator can win the one-wayness challenge, and thus, by the one-wayness property of the hash in use, this could happen with only a negligible probability. Note however that the executions in HYBRID$_{4,3}$ and HYBRID$_{4,2}$ are distributed the same since D_j is chosen at random in both. Therefore, we get that on one side, the abort in HYBRID$_{4,3}$ happens with a negligible probability, and on the other side, the abort in HYBRID$_4$ happens with a noticeable probability, while the two are indistinguishably different (up to call to $\mathcal{F}_{tcot}$), which is a contradiction.

The above hybrids are used only for showing that the abort in HYBRID$_4$ happens with a negligible probability. However, the next hybrids follow HYBRID$_4$, and not HYBRID$_{4,3}$.

HYBRID$_5$: The simulator replaces the garbled circuits for C' with simulated garbled circuits that always output 0 for all circuits not chosen to be checked, and replaces the labels that P_2 chose not to learn in the inputs to $\mathcal{F}_{tcot}$ with random strings. Also, it replaces the commitments of lc for those circuits, that are not decommitted, with commitments to random strings. Since P_2 does not know the values of D_j-s, whatever it inputs to C' should return 0 with high probability. Thus, by the security of the garbling scheme, the OTs, and the commitment in use, HYBRID$_5$ and HYBRID$_4$ are computationally-indistinguishable (following the same arguments as above).

HYBRID$_6$: The simulator uses only zeros as the P_1's inputs. The only information about x_j-s that P_2 sees comes from the commitments $\{g^{a_i^{x_{j,i}} r_j}\}$ and their corresponding ZK proofs. The latter are the same in HYBRID$_6$ and HYBRID$_5$ since we work with an ideal functionality for ZKPoK, while the former are computationally-indistinguishable by the DDH assumption.

HYBRID$_7$: Instead of computing $z_j = f(x_j, y_j)$ by itself, the simulator sends $y_1, \ldots, y_N$ to the trusted third party and receives $z_1, \ldots, z_N$. (If Adv or the emulated P_1 abort the execution during Part 1 of the protocol, the simulator sends $\perp$ to the trusted third party. If they abort during Part 2, it sends abort.) HYBRID$_7$ and HYBRID$_6$ are distributed the same since P_2 does not see the call to the trusted third party, and the honest P_1 has no output.

The simulator: We take the simulator to be the last simulator from above. I.e., the simulator emulates an honest P_1 for the adversary with the next modifications:

- During the execution, it extracts Adv's inputs to $\mathcal{F}_{tcot}$, and its witnesses to $\mathcal{F}_{zkpok}$. It uses the latter to recover $y_1, \ldots, y_N$.
- Sets $x_1 = x_2 = \cdots = x_N = 0$.
- For all circuits that Adv chose to check, the simulator answers properly in its response in $\mathcal{F}_{tcot}$ (i.e., with correct garbled circuit, etc).
- If Adv or the emulated P_1 abort the execution during Part 1 of the protocol, sends $\perp$ to the trusted third party.
- After Adv sends its $\mathcal{F}_{tcot}$ inputs, the simulator sends $y_1, \ldots, y_N$ to the trusted third party and receives $z_1, \ldots, z_N$. Then, in the evaluation circuits, it uses fake garbled circuits for C that always output the right z_j, and fake garbled circuits for C' that always output 0. (The simulator knows π since it extracts $seed_\pi$.) In addition, it commits to random strings in the commitments that are not opened for the garbled circuits of C'.
- If Adv uses D_j in $\mathcal{F}_{tcot}$ for C', the simulator aborts.
- If Adv or the emulated P_1 abort the execution during Part 2 of the protocol, sends abort to the trusted third party.

- Outputs whatever Adv outputs.

By the above sequence of hybrid games, we conclude that the simulated execution is computationally-indistinguishable from the real execution.

■

Achieving UC security. Observe that the above simulators are straightline simulators that use the adversary in a black-box fashion. If the ideal functionalities are realized by UC-secure protocols, then our protocol can be shown to be secure under the UC-security notion. [LP11] presents such realizations in the standard model, that result in a multiplicative overhead of $\mathcal{O}(s)$ per ZK proof.

6.1.1 Protocol Complexity and Comparison

We consider the following two sets of parameters (see Appendix A): **(1)** $N = 32$, $s = 40$, $p = 0.75$, $B = 10$ and $p' = 0.6$, $B' = 31$, and **(2)** $N = 1024$, $s = 40$, $p = 0.85$, $B = 6$ and $p' = 0.7$, $B' = 12$. We focus on the costs that are related to $|C|$. The protocol of [Lin13] requires $6.5sN|C|$ symmetric key operations only for the garbled circuits. for $N = 32$, it equals $8320|C|$, while in our protocol only $2776|C|$ operations are needed, and for $N = 1024$, that protocol requires $266240|C|$ operations, while our protocol requires only $46989|C|$.

The protocol of [NNOB12] requires at least 300 hash invocations per gate of the circuit (when the bucketizing of that protocol is minimal and equals 2). For 32 2PC executions they need about $9600|C|$ hashes, which is almost 3.5 times more than our protocol needs. For 1024 2PC executions, they require $307200|C|$ hashes, which is more than 6.5 times more than required by our protocol. Note, however, that our protocol requires a significant number of exponentiations that depend on the number of inputs (although using methods from [MR13] it is possible to significantly reduce the number of exponentiations). Therefore, the total cost of our protocol might be larger than of the protocol of [NNOB12] if the circuit is not large and communication is cheap. Our protocol is more efficient mostly when communication is the dominant cost and the circuit is large.

6.2 Reducing Exponentiations for P_2's Inputs

Protocol 6.1 requires a constant number of exponentiations per P_2's input bit, per garbled circuit constructed, which is approximately $\mathcal{O}(M|y| + M'k)$ exponentiations overall.

[LP07] shows an alternative solution for the selective-OT attack, which works with any oblivious transfer in a black-box way, based on an encoding of P_2's input in a way that any leakage of a small portion of the bits does not reveal significant information about P_2's input. Formally, the encoding can be done with respect to a boolean matrix E that is s-probe-resistant, as defined below.

Definition 6.3 (Based on [LP07, SS13]). *Matrix $E \in \{0,1\}^{l \times n}$ for some $l, n \in \mathbb{N}$ is called s-probe-resistant for some $s \in \mathbb{N}$ if for any $L \subset \{1, 2, \ldots, l\}$, the Hamming distance of $\bigoplus_{i \in L} E_i$ is at least s, where E_i denotes the i-th row of E.*

[LP07] show how to construct such matrix E with $n = \max(4l, 8s)$. [SS13] show an alternative construction with $n \leq \log(l) + l + s + s \cdot \max(\log(4l), \log(4s))$.

Now, instead of working with the function $f(x, y)$, the parties work with the function $f'(x, y') = f(x, y'E)$, for which P_2 chooses a random y' such that $y = y'E$. As long as E is s-probe-resistant, even if P_1 learns $s' < s$ bits of y', it cannot learn any significant information about y. Since in order to learn s bits it has to selective-OT attack s wires, this means that if it tries to attack s wires, it gets caught with probability at least $1 - 2^{-s}$. In addition to working with $f'(x, y')$, the parties can use one OT invocation for many circuits, allowing P_2 to input the same y' for many circuits while learning the corresponding labels in all of them together. Therefore, the number of OTs needed is n for the entire set of evaluated circuits.

[SS13] shows that since E is a binary matrix, the subcircuit that computes $y'E$ can be garbled very efficiently using the Free-XOR technique [KS08], with only $\mathcal{O}(n)$ symmetric-key operations. This modification requires assuming that the garbling scheme in use is secure with the Free-XOR technique (see [CKKZ12]). Moreover, in the random-oracle model, many OTs can be implemented very efficiently (i.e., with a small number of symmetric-key operations per OT) using the OT-extension of [NNOB12], thus the above solution can be implemented by $\mathcal{O}(n)$ symmetric-key operations (and $\mathcal{O}(s)$ seed-OTs).

Adapting the main batch protocol. We adapt Protocol 6.1 to work with this method, and by that reduce the overhead for P_2 to learn the labels of its inputs from $\mathcal{O}(M|y|+M'k)$ exponentiations to $\mathcal{O}(M|y| + M'k)$ symmetric-key operations.

The main idea we use for this adaptation is to split the OTs into two steps, as shown by [Bea95] for preprocessing OTs. We describe the modifications based on the steps of the cut-and-choose stage for C in Protocol 6.1:

- Step 3b: P_1 creates the garbled circuits using the Free-XOR method, and where P_2's input is encoded using a (publicly known) s-probe-resistant matrix. (Still, P_1's input labels are constructed as in [Lin13].)
- Step 3c: In addition to the commitment lc_j, P_1 also computes the set of commitments $\left\{\left(i, \mathsf{Com}(W^0_{j,i}), \mathsf{Com}(W^1_{j,i})\right)\right\}_{i \in I_2}$ where I_2 is the set of wires for input y and $W^b_{j,i}$ is the bth label of the ith wire of gc_j. Denote this set by $p2lc_j$.
- Step 4 is replaced with following steps:
  - P_2 commits on $seed_\pi$ and J using a trapdoor commitment (as done in Section **??**).
  - For $j = 1, \dots, N$, the players run $\mathcal{O}(|y|)$ OTs (according to the above parameters) in which P_1 inputs random strings of length $2k \cdot B$ and P_2 inputs its input y_j.
  - P_1 sends the sets $(g^{r_1}, gc_1, lc_1, p2lc_1), \dots, (g^{r_M}, gc_M, lc_M, p2lc_M)$.
  - P_2 decommit $seed_\pi$ and J. P_1 reveals the seed $seed_j$ of the checked circuits $j \in J$.
  - For bucket $j = 1, \dots, N$, P_1 sends corrections to the random strings it has used in the OTs for this bucket and the input labels for the circuits of this bucket and their decommitments. E.g., if P_1 used the $2k \cdot B$ bit strings a_0, a_1 in the first OT for bucket j, it sends the xor of a_0 with the concatenation of all the labels that correspond to 0 for all the circuits in the bucket, and their decommitments (for the commitments in $p2lc_j$). It does the same with the xor of a_1 and the labels that that correspond to 1.

    P_2 computes the corrected labels of its inputs and verifies the decommitments.
- Step 5: P_2 also checks $p2lc_j$ (and ignores the OTs part).

The proof of security is similar to the proof of Theorem 6.2 and therefore omitted.

6.3 Reducing Exponentiations for P_1's Input

The main protocol requires a constant number of exponentiations per P_1's input bit, per garble circuit constructed. This is needed because of the solution we use for enforcing P_1 to use the same input in the different circuits of the bucket. [MR13] presents an alternative solution to this issue that requires only a small number of symmetric-key operations per P_1's input bit. This solution can also be plugged in into our protocol, though it requires an efficient protocol for OT-extension, which we currently know how to instantiate only in the random-oracle model [NNOB12]. Since the combination of our protocol with the techniques of [MR13] result is rather complicated, we omit further details and leave the question of coming up with a simpler solution in the standard model, as an open problem.

In the setting in which P_1's input is fixed for all computations (e.g., when P_1 has a key to an AES encryption, and P_2 wants to compute the encryptions of its messages), the exponentiations needed in our protocol for verifying the consistency of P_1's input can be removed, and the technique of [SS13] for checking consistency can be used instead, assuming that the garbling scheme in use is secure with the Free-XOR technique [KS08]. [SS13] uses a universal-hash function that can be computed very efficiently, using only a linear number (in P_1's input length and s) of symmetric key operations.

7 Secure Two-Party Computation in the Online/Offline Setting

See Section 2 for a high-level description of our techniques. Instead of working directly with a circuit that computes $f(x, y)$, we work with the circuit $C(x^{(1)}, x^{(2)}, y^{(1)}, y^{(2)})$ that computes $f(x^{(1)} \oplus x^{(2)}, y^{(1)} \oplus y^{(2)})$. P_1's inputs are $x^{(1)}$ and $x^{(2)}$, while P_2's inputs are $y^{(1)}$ and $y^{(2)}$. We only require that $x^{(1)}$ and $y^{(1)}$ must remain private at the end of the protocol (i.e., $x^{(2)}, y^{(2)}$ can, and will, be published).

We make use of input wires with public values, which are inputs of the circuits that both parties know at some point of the protocol execution. We call these *public-input wires*. For example, the input wires for $x^{(2)}, y^{(2)}$ of C are public-input wires in our protocol since the values $x^{(2)}, y^{(2)}$ are known to both parties during the protocol execution.

In order to implement the cheating recovery technique, the parties also work with the modified circuit $C'(x^{(1)}, D, d^{(1)}, d^{(2)})$ that outputs $x^{(1)}$ if $D = d^{(1)} \oplus d^{(2)}$ and 0 otherwise, with $d^{(2)}$ and D being public-input wires. An honest P_1 should use the same $x^{(1)}$ in all the circuits C and C' in a bucket. This requirement is enforced by our protocol.

7.1 The Main Protocol

The protocol is described in Figure 7.1 and Protocols 7.2–7.3. Since similar steps are run for C and C', Protocol 7.2 describes the entire offline stage which executes the protocols of Figure 7.1 twice (i.e., once for C and once for C'). In the protocol description, when r is used without subscript in the protocol, it denotes an independent random value that is not referred to in other steps of the protocol.

FIGURE 7.1.

Creating a Garbled-Circuit Bundle

Public Parameters:

- A circuit $C\left(x^{(1)},x^{(2)},y^{(1)},y^{(2)}\right)$ with $x^{(2)}$ and $y^{(2)}$ being public-input wires, and $\left|y^{(1)}\right| = \left|y^{(2)}\right|$.
- The set $\left\{\left(i, g^{a_i^0}, g^{a_i^1}\right)\right\}_{i=1,\dots,|x^{(1)}|}$ (these are actually chosen by P_1; see Figure 7.2).

Construct the bundle:

- Pick a seed $seed \in_R \{0,1\}^k$. All the randomness needed in the next steps is derived from $\mathsf{PRF}_{seed}(\cdot)$.
- Pick $r \in_R \mathbb{Z}_q$
- Construct a garbled circuit (gc, en, de) in which the input-wire labels of input i are $\mathsf{REHash}\left(g^{a_i^0 r}\right), \mathsf{REHash}\left(g^{a_i^1 r}\right)$ for $i = 1,\dots,|x^{(1)}|$ (these are the labels of P_1's input $x^{(1)}$), and where the output-wire labels are the actual output bits concatenated with random labels. (E.g., the output label for bit zero is $0|l$ where $l \in_R \{0,1\}^k$).
- Commit to all public-input wires (for $x^{(2)}$ and $y^{(2)}$) by
$$\left\{\left(i, \mathsf{Com}(W_i^0), \mathsf{Com}(W_i^1)\right)\right\}_{i=|x^{(1)}|+1}^{|x^{(1)}|+|x^{(2)}|} \bigcup \left\{\left(i, \mathsf{Com}(W_i^0), \mathsf{Com}(W_i^1)\right)\right\}_{i=|x^{(1)}|+|x^{(2)}|+|y^{(1)}|+1}^{|x^{(1)}|+|x^{(2)}|+|y^{(1)}|+|y^{(2)}|}$$
- Commit to all output-wire labels by $\left\{\left(i, \mathsf{Com}(W_i^0), \mathsf{Com}(W_i^1)\right)\right\}_{i\in \mathsf{Outputs}(C)}$.[a]
- Let lc be the union of the above sets of label commitments, and let lcd be the set of all the corresponding decommitments.
- Output $(gc, lc, g^r; seed, en, de, lcd, r)$.

The Cut-and-Choose Mechanism

Public parameters:

- Let $s, N, B \in \mathbb{N}$ and $p \in (0,1)$ parameters. Let $M = \frac{NB}{p}$. (Assume no rounding of M is needed.)
- A circuit $C\left(x^{(1)},x^{(2)},y^{(1)},y^{(2)}\right)$ with $x^{(2)}$ and $y^{(2)}$ being public-input wires, and that $|y^{(1)}| = |y^{(2)}|$.
- g_0, g_1, h and the set $\left\{\left(i, g^{a_i^0}, g^{a_i^1}\right)\right\}_{i=1,\dots,|x^{(1)}|}$ (see Protocol 7.2).

Picking the cut, the buckets, and the offline inputs:

- *The cut:* P_2 sets σ to be a random string of length M that has exactly NB ones.
- *The mapping:* P_2 picks a PRF seed $seed_\pi$ and uses $\mathrm{PRF}_{seed_\pi}(\cdot)$ to compute a mapping function $\pi : [N \cdot B] \to [N]$ that maps exactly B elements to each bucket.
- *"Offline" inputs:* P_2 chooses $y_1^{(1)},\dots,y_N^{(1)} \in_R \{0,1\}^{|y^{(1)}|}$.

The cut-and-choose:

- For $j = 1,\dots,M$, (can be done in parallel)
  - P_1 runs the garbled-circuit bundle construction procedure above with the circuit C, and receives $(gc_j, lc_j, g^{r_j}; seed_j, en_j, de_j, lcd_j, r_j)$.
  - P_1 sends $\mathsf{PCommit}_h(\mathsf{Hash}(gc_j|lc_j))$ and g^{r_j}.
- P_1 acts as the sender in $\mathcal{F}_{tcot}$ and P_2 as the receiver. P_2 inputs $y_1^{(1)},\dots,y_N^{(1)}$, $seed_\pi$ and J.
  Let dpc_i be the decommitment of $\mathsf{PCommit}_h(\mathsf{Hash}(gc_i|lc_i))$. P_1 inputs the wire labels of P_2's input (i.e., in the jth set it inputs $|y|$ pairs of keys) and the strings $seed_1|dpc_1,\dots,seed_M|dpc_M$.
- P_2 computes the set $\{gc_i, lc_i\}_{j\in J}$ using the seeds it received from $\mathcal{F}_{tcot}$, and verifies that all decommitments $\{dpc_i\}_{j\in J}$ are correct and that all OT answers are consistent for those garbled circuits.

[a]This part is unnecessary when the circuit in use is C', but since the additional overhead is small, we ignore this optimization here.

PROTOCOL 7.2 (The Offline Stage)**.**

Setup:

- s is a statistical security parameter, N is the number of online 2PC executions that P_2 wishes to run, p, p' are probabilities, and B, B' are chosen according to Theorems 4.1 and 4.3.
- The parties decide on two circuits: **(1)** A circuit $C\left(x^{(1)}, x^{(2)}, y^{(1)}, y^{(2)}\right)$ that computes $f\left(x^{(1)} \oplus x^{(2)}, y^{(1)} \oplus y^{(2)}\right)$, with $x^{(2)}$ and $y^{(2)}$ being public-input wires. **(2)** A circuit $C'\left(x^{(1)}, D, d^{(1)}, d^{(2)}\right)$ that computes $\left[D = (d^{(1)} \oplus d^{(2)})\ ?\ x^{(1)} \mid 0^{|x^{(1)}|}\right]$, with $d^{(2)}$ and D being public-input wires. (Recall that $|x^{(1)}|$ should be the same as in C.)
- P_1 chooses $a_1^0, a_1^1, \ldots, a_{|x^{(1)}|}^0, a_{|x^{(1)}|}^1 \in_R \mathbb{Z}_q$, and sends the set $\left\{\left(i, g^{a_i^0}, g^{a_i^1}\right)\right\}_{i=1,\ldots,|x^{(1)}|}$.
- Let $g_0 = g$. P_2 chooses $r \in_R \mathbb{Z}_q$, sends $g_1 = g^r$ and proves using a ZKPoK that it knows $\log_g(g_1)$.
- P_2 chooses $r \in_R \mathbb{Z}_q$, sends $h = g^r$ for $\mathsf{PCommit}_h(\cdot)$ and proves, using ZKPoK, that it knows $\log_g(h)$.

Running the cut-and-choose for C and for C':

- The parties run the cut-and-choose part from Figure 7.1 with the circuit C and parameters p, N and B.
- The parties run the cut-and-choose protocol from Figure 7.1 with the circuit C' and parameters p', N and B'. (Note that the same N is used in both invocations of the protocol from Figure 7.1, so both result in the same number of buckets.)

From now on, we refer to the elements of the second execution with prime. E.g. π' is the mapping function of the second execution from above (while π is of the first one).

Finishing bucketizing:

- For bucket $j = 1, \ldots, N$ (can be done in parallel),
  - P_1 picks $x_j^{(1)} \in_R \{0,1\}^{|x^{(1)}|}$. Let $t = x_j^{(1)}$.
  - For $v = 1, \ldots, |x^{(1)}|$, P_1 sends the commitments that correspond to t_v in $\{gc_{j,i}\}_{i=1,\ldots,B}$ and $\{gc'_{j,i}\}_{i=1,\ldots,B'}$, and proves using ZK proofs that they are consistent with each other, as in [Lin13].

Storing buckets for the online stage: For bucket $j = 1, \ldots, N$:

- P_1 stores $x_j^{(1)}$, the decommitment of $\mathsf{PCommit}_h(\mathsf{Hash}(gc_{j,i}|lc_{j,i}))$ and $(gc_{j,i},\ en_{j,i},\ de_{j,i},\ lc_{j,i},\ lcd_{j,i})$ for $i = 1, \ldots, B$, and similarly for all the bundles of C'.
- P_2 stores $y_j^{(1)}$. In addition, it stores $\mathsf{PCommit}_h(\mathsf{Hash}(gc_{j,i}|lc_{j,i}))$, the labels it has received for its input $y_j^{(1)}$ from the OTs, the values of $\{\mathsf{REHash}\left(g^{a_v^{t_v} r_{j,i}}\right)\}_{v=1,\ldots,|x^{(1)}|}$, where $t = x_j^{(1)}$, for $i = 1, \ldots, B$, and similarly for all the bundles of C'.

The goal of the offline stage is to prepare small buckets of garbled circuits that will be used in the online stage, one bucket per online 2PC invocation. Note that after the offline stage, except with probability 2^{-s}, we have that *for each bucket*:

- At least one garbled circuit of C in the bucket, and a majority of the garbled circuit for C' in the bucket, are correctly garbled circuits, and consistent with the OT inputs for P_2 to learn the labels of its inputs.
- P_1 is committed to the same input $x^{(1)}$ in all the good garbled circuits (C and C').
- P_2's input $y^{(1)}$ is the same in all of the circuits C, and the same goes for input $d^{(1)}$.
- No party has learned anything about the other party's inputs ($x^{(1)}$ and $y^{(1)}$).

Once a bucket fulfills the above conditions, executing a 2PC with that bucket in the online stage would be secure for the reasons described earlier in Section 2.

PROTOCOL 7.3 (The Online Stage)**.**

We focus here on a single 2PC with a single bucket. For simplicity, we omit the bucket index j when we refer to its garbled circuits, etc.

Private inputs: P_1's input is x. P_2's input is y.

Evaluating C:

- P_2 sends $y^{(2)} = y \oplus y_j^{(1)}$.
- P_1 sends $x^{(2)} = x \oplus x_j^{(1)}$.
- For $i = 1, \ldots, B$,
  - P_1 decommits $\mathsf{PCommit}_h(\mathsf{Hash}(gc_i|lc_i))$ and sends gc_i, lc_i.
  - P_1 sends the input-wire labels for $y^{(2)}$ and $x^{(2)}$ in gc_i, and the decommitments of those labels for the corresponding commitments in lc_i.
- P_1 picks $D \in_R \{0,1\}^k$.
- For $v \in \mathsf{Outputs}(C)$,
  - P_1 chooses $R_v \in_R \{0,1\}^k$.
  - Let $W_{i,v}^b$ be the b-th label of output wire v of gc_i, where $v \in \mathsf{Outputs}(C)$. P_1 sends $\mathsf{Enc}(W_{i,v}^0, R_v), \mathsf{Enc}(W_{i,v}^1, R_v \oplus D)$ for $i = 1, \ldots, B$.
- P_1 sends $\mathsf{Hash}(D)$.
- P_2 evaluates gc_i, for $i = 1, \ldots, B$, and then uses the output wire labels to decrypt the associated R_v and $R_v \oplus D$ values. In case it learns both R_v and $R_v \oplus D$ for some output wire, it checks if the XOR of them is indeed D (by applying $\mathsf{Hash}(\cdot)$ and comparing with the value that P_1 has sent). If so, it sets d to D. Otherwise, it sets $d \in \{0,1\}^s$.
- If all evaluations (that ended) returned the same output, set z to be that output.

Evaluating C':

- Let $d^{(1)}$ the input that P_2 used in the OTs for circuit C' in bucket j. P_2 sets $d^{(2)} = d \oplus d^{(1)}$ and sends it.
- P_2 sends D, and for $i = 1, \ldots B'$,
  - Decommits $\mathsf{PCommit}_h(\mathsf{Hash}(gc'_i|lc'_i))$ and sends gc'_i, lc'_i.
  - Sends the labels that correspond to D and $d^{(2)}$ in gc'_i, and decommits the corresponding commitments from lc'_i.
- P_2 decommits the commitments on the output labels of gc_i, for $i = 1, \ldots B$ (i.e. revealing all output wire labels of the garbled circuits for C).
- P_2 verifies all decommitments, all the encryptions $\mathsf{Enc}(W_{i,v}^0, R_v), \mathsf{Enc}(W_{i,v}^1, R_v \oplus D)$, for $i = 1, \ldots, B$ and $v \in \mathsf{Outputs}(C)$, and the hash $\mathsf{Hash}(D)$, and aborts if there is a problem.
- P_2 evaluates gc'_i, for $i = 1, \ldots B'$, and takes the majority output to be $\hat{x}^{(1)}$. (Recall that P_2 already has the labels associated with input $x^{(1)}$ from the offline stage.)

P_2's output:

- If all evaluation circuits of C returned the same output z, then P_2 outputs z.
- Else, if P_2 has learned earlier d such that $\mathsf{Hash}(d) = \mathsf{Hash}(D)$, then it outputs $f(\hat{x}^{(1)} \oplus x^{(2)}, y)$.
- Else, let gc_i be a circuit for which all the output labels that P_2 received from its evaluation were also the labels that were decommitted earlier from lc_i. P_2 outputs the output of gc_i.

We prove the following theorem:

Theorem 7.4. *Assume that the Decisional Diffie-Hellman problem is hard in the group $\mathbb{G}$, and that $\mathsf{Com}(\cdot)$, $\mathsf{Enc}(\cdot,\cdot)$, $\mathsf{Hash}(\cdot)$ and $\mathsf{REHash}(\cdot)$ are secure. Then, the protocol of Figure 7.1 and Protocols 7.2–7.3 securely computes any polynomial-time f with multiple executions in the online/offline setting in the presence of malicious adversaries.*

Before proving the above theorem, we note that if one wants only a single invocation of 2PC in the online/offline setting, then all that one needs to change in the above protocol is the selection of the cut: P_1 prepares s (or $3s$) sets of garbled circuits for C (or C'), and P_2 picks a random subset (or a 2/5 portion) of them for evaluation. The rest of the steps remain the same except for ignoring the bucket related parts. As a result we get the following theorem:

Theorem 7.5 (Informal)**.** *The modified protocol described above securely computes any polynomial-time f in the online/offline setting in the presence of malicious adversaries.*

We omit the proof of Theorem 7.5 since it is a specific case of Theorem 7.4.

Proof of Theorem 7.4. We prove security in a hybrid model where tweaked batch cut-and-choose OT, zero-knowledge proofs and proofs of knowledge (ZKPoK) are computed by ideal functionalities $\mathcal{F}_{tcot}, \mathcal{F}_{zkp}, \mathcal{F}_{zkpok}$.

We separately prove the case that P_1 is corrupted and the case that P_2 is corrupted.

P_1 **is corrupted.** The intuition here is similar to the one of the proof of Theorem 6.2. We describe a sequence of hybrid games that ends with the simulated execution. (We mention only the differences between each consecutive hybrids.)

HYBRID$_0$: An execution of the offline stage and the online stage with a simulator that emulates honest P_2 with inputs $y_1, \ldots, y_N$. Since we work in an hybrid model with the ideal functionality $\mathcal{F}_{zkpok}$, the simulator can extract P_1's witnesses, and in particular, extract the witness P_1 uses for proving consistency of its inputs, and by that, learn P_1's inputs to the correctly garbled circuits. In addition, the simulator learns both P_1's inputs to $\mathcal{F}_{tcot}$.

HYBRID$_1$: We say that set j is *good* if: (1) $seed_j$ correctly derives the jth bundle; (2) The OT answers for the labels of gc_j are consistent with the actual labels of gc_j; (3) The decommitment of $\mathsf{PCommit}_h(\mathsf{Hash}(gc_j|lc_j))$ that P_1 sent to $\mathcal{F}_{tcot}$ is correct and consistent with the seed $seed_j$ it sent.

The simulator aborts if there exists a bucket j such that none of its sets for C is good, or that the majority of the sets for C' are not good.

By the cut-and-choose stages and Theorems 4.1 and 4.3, we know that at least one of the sets of C, and that most of the sets of C' are good with probability at least 2^{-s}, thus, HYBRID$_1$ and HYBRID$_0$ are 2^{-s}-indistinguishable.

HYBRID$_2$: The simulator aborts if there exists a good set j that is not checked, and that one of the following occurs:

- The set's commitment $\mathsf{PCommit}_h(\mathsf{Hash}(gc_j|lc_j))$ is decommitted to something different than $\mathsf{Hash}(gc_j|lc_j)$.
- P_1 sent $gc'_j|lc'_j$ such that $\mathsf{Hash}(gc_j|lc_j) = \mathsf{Hash}(gc'_j|lc'_j)$ but $gc'_j|lc'_j$ is not derived from $seed_j$.

- P_1 sent decommitments to commitments of lc_j that are for different labels than the ones generated by $seed_i$.

Note that in all the above cases, the simulator aborts in case P_1 has successfully cheated in the commitment or the (collision resistancy of the) hash function. Thus, by assuming that the DDH assumption holds in $\mathbb{G}$ and that $\mathsf{Hash}(\cdot)$ and $\mathsf{Com}(\cdot)$ are secure, this abort happens with only a negligible probability.

HYBRID$_3$: Let $x_j^{(1)}$ be P_1's "offline" input to the jth bucket (extracted from its witness to $\mathcal{F}_{zkpok}$). The simulator emulates an honest P_2 with random inputs $y_1^{(2)}, \ldots, y_N^{(2)}$ in the online stage, and if the emulated P_2 does not abort, the simulator outputs $f(x_j^{(1)} \oplus x_j^{(2)}, y_j)$ as P_2's output.

Note that in HYBRID$_2$, $y_j^{(1)}$ is uniformly random in $\{0,1\}^{|y^{(2)}|}$ and thus $y_j^{(2)}$ is uniformly random as well. Similarly, $d_j^{(2)}$ is uniformly random in $\{0,1\}^k$. Therefore, the only difference between the executions in HYBRID$_3$ and HYBRID$_2$ is in case P_2's output at the end is different. However, as discussed in the proof of Theorem 6.2, if indeed at least one garbled circuit for C and most of the garbled circuits for C' are good, then P_2's output in HYBRID$_2$ is the correct output, or abort independently of $y_1, \ldots, y_N$. Thus, HYBRID$_3$ and HYBRID$_2$ are distributed the same.

HYBRID$_4$: Instead of computing (and outputting) $f(x_j^{(1)} \oplus x_j^{(2)}, y_j)$ by itself at the end, the simulator sends $x_1^{(1)} \oplus x_1^{(2)}, \ldots, x_N^{(1)} \oplus x_N^{(2)}$ to the trusted third party and outputs whatever Adv outputs. (If Adv or the emulated P_2 abort the execution, the simulator sends $\perp$ to the trusted third party.) Since the only difference between HYBRID$_4$ and HYBRID$_3$ is that P_2's output is computed by the trusted third party, the two are distributed the same.

The simulator: We take the simulator to be the last simulator from above. I.e., the simulator emulates an honest P_2 for the adversary with the next modifications:

- Sets $y_j^{(1)} = 0$ and $d_j^{(1)} = 0$ for $j = 1, \ldots, N$.
- Extracts Adv's inputs $x_1^{(1)}, \ldots, x_N^{(1)}$ from $\mathcal{F}_{zkpok}$ and its input to $\mathcal{F}_{tcot}$, as in HYBRID$_0$.
- Extracts Adv's inputs to $\mathcal{F}_{tcot}$ and aborts if there exists a bucket j such that none of its sets for C is good, or that the majority of the sets for C' are not good.
- Sets $y_j^{(2)}$ and $d_j^{(2)}$ to random values.
- Checks Adv's commitments as done in HYBRID$_2$ and aborts if there is a commitment/hash that is "opened" to two different inputs.
- Sends abort to the trusted third party if Adv or the emulated P_2 abort.
- Sends $x_1^{(1)} \oplus x_1^{(2)}, \ldots, x_N^{(1)} \oplus x_N^{(2)}$ to the trusted third party and outputs whatever Adv outputs.

By the above sequence of hybrid games, we conclude that the simulated execution is $(\mu(k) + 2^{-s})$-indistinguishable from the real execution.

P_2 **is corrupted.** As before, we describe a sequence of hybrid games that ends with the simulated execution.

$\textsc{hybrid}_0$: An execution of the offline stage and the online stage with a simulator that emulates honest P_1 with inputs $x_1, \ldots, x_N$. Since we work in an hybrid model with ideal functionalities $\mathcal{F}_{zkpok}$ and $\mathcal{F}_{tcot}$, the simulator can extract P_2's witnesses, and its inputs to $\mathcal{F}_{tcot}$. In particular, it extracts: (1) P_2's OT inputs; (2) π and J; (3) The value of $\log_g(h)$ which allows the simulator to cheat regarding $\mathsf{PCommit}_h(\cdot)$.

$\textsc{hybrid}_1$: For all $j \in J$, the simulator generates the j-th garbled circuit bundle with true randomness (and not a pseudo-random randomness). Since for such j, P_2 does not get any information about $seed_j$ anyhow (since it gets only c_j from $\mathcal{F}_{tcot}$), $\textsc{hybrid}_1$ and $\textsc{hybrid}_0$ are indistinguishable by the security of $\mathsf{PRF}(\cdot)$ in use.

$\textsc{hybrid}_2$: The simulator uses random strings in the OT answers that P_2 chose not to learn when $j \notin J$ (still, independent of bucket j). Since $\mathcal{F}_{tcot}$ does not send to P_2 anything about those values,$\textsc{hybrid}_2$ and $\textsc{hybrid}_1$ are distributed the same.

$\textsc{hybrid}_3$: After P_2 sends $y_j^{(2)}$ in the online stage, the simulator computes $z_j = f(x_j, y_j^{(1)} \oplus y_j^{(2)})$, and replaces the garbled circuits for C in the jth bucket with simulated garbled circuits that always output z_j. Also, it replaces the commitments of lc of the input labels that are not decommitted, with commitments to random strings. Note that the simulator knows $\log_g(h)$ and thus can decommit $\mathsf{PCommit}_h(\cdot)$ to whatever value it likes. By the security of the garbling scheme (or the underlying encryption in use) and the hiding property of the commitments, $\textsc{hybrid}_3$ and $\textsc{hybrid}_2$ are computationally-indistinguishable.

$\textsc{hybrid}_4$: The simulator aborts if P_2 sends $d_j^{(2)}$ such that $D_j = d_j^{(1)} \oplus d_j^{(2)}$. This abort happens with a negligible probability (that depends on the security parameter k), as shown in the proof in Theorem 6.2, and therefore $\textsc{hybrid}_4$ and $\textsc{hybrid}_3$ are computationally-indistinguishable.

$\textsc{hybrid}_5$: The simulator replaces the garbled circuits for C' with simulated garbled circuits that always output 0 for all the circuits that are not chosen to be checked. Also, it replaces the commitments of lc for those circuits, that are not decommitted, with commitments to random strings. As before, since the simulator knows $\log_g(h)$ it can decommit $\mathsf{PCommit}_h(\cdot)$ to whatever value it likes. Since P_2 does not know the values of D_j-s, whatever it inputs to C' should return 0. Thus, by the security of the garbling scheme and the commitment in use, $\textsc{hybrid}_5$ and $\textsc{hybrid}_4$ are computationally-indistinguishable.

$\textsc{hybrid}_6$: The simulator sets $x_1^{(1)}, \ldots, x_N^{(1)}$ to be 0 in the offline stage, and picks $x_j^{(2)}$ of the jth bucket uniformly at random in the online stage. $x_j^{(2)}$ is uniform both in $\textsc{hybrid}_6$ and $\textsc{hybrid}_5$, thus the only difference is regarding $x_1^{(1)}, \ldots, x_N^{(1)}$. However, the only information about these values comes from the input commitments and their corresponding ZK proofs. The latter are the same in $\textsc{hybrid}_6$ and $\textsc{hybrid}_5$ since we work with an ideal functionality for ZKPoK, while the former are computationally-indistinguishable by the DDH assumption.

$\textsc{hybrid}_7$: Instead of computing $z_j = f(x_j, y_j^{(1)} \oplus y_j^{(2)})$ by itself, the simulator sends $y_j^{(1)} \oplus y_j^{(2)}$ to the trusted third party and receives z_j. (If Adv or the emulated P_2 abort in the online stage of bucket j, then Sim sends $\mathsf{abort}(j)$ to the trusted third party. If they abort in the offline stage, it sends $\perp$.) $\textsc{hybrid}_7$ and $\textsc{hybrid}_6$ are distributed the same since P_2 does not see the call to the trusted third party, and the honest P_1 has no output.

The simulator: We take the simulator to be the last simulator from above. I.e., the simulator emulates an honest P_1 for the adversary with the next modifications:

- During the execution, it extracts Adv's inputs to $\mathcal{F}_{OT}$, and its witnesses to $\mathcal{F}_{zkpok}$.
- Sets $x_j^{(1)} = 0$ for $j = 1, \ldots, N$
- Sends commitments $\mathsf{PCommit}_h(\cdot)$ to random values for all garbled circuit bundles.
- For all bundles that Adv chose to check, the simulator answers properly in its response in $\mathcal{F}_{tcot}$ (i.e., with correct garbled circuit, etc). For the other bundles, the OT answers that P_2 chose not to learn are replaced with random values.
- If Adv or the emulated P_1 abort in the offline stage, sends $\perp$ to the trusted third party.
- After Adv sends $y_j^{(2)}$ in the online stage, sends $y_j^{(1)} \oplus y_j^{(2)}$ to the trusted third party and receives z_j. Then, it decommits $\mathsf{PCommit}_h(\cdot)$ to fake garbled circuits for C that always output z_j, and the ones of C' to to fake garbled circuits that always output 0. The simulator does this for all the circuits of the jth bucket.
- If Adv uses $d_j^{(1)} \oplus d_j^{(2)} = D_j$, the simulator aborts.
- If Adv or the emulated P_1 abort in the online stage of bucket j, sends $\mathsf{abort}(j)$ to the trusted third party.
- Outputs whatever Adv outputs.

By the above sequence of hybrid games, we conclude that the simulated execution is indistinguishable from the real execution.

■

As in Section 6, if the ideal functionalities are realized by UC-secure protocols, then the above protocol can be shown to be secure under the UC-security notion.

7.2 Reducing Online Stage Latency in the ROM

As discussed in Sections 2.4 and 3, the standard security notions of garbled circuits are defined with respect to a static adversary who chooses its input before seeing the garbled circuit. Due to this, in the protocol from Section 7.1, P_1 does not send a garbled circuit before knowing whether P_2 wants to check it, or before P_2 chooses its input in case the circuit is used in the online stage. P_1 only commits on the circuit using a perfectly-hiding commitment. This allows the simulator to later change the garbled circuit based on P_2's input, and reduce security of the simulated garbled circuit to the security of the encryption in use. However, this technique has a significant disadvantage - it requires sending B garbled circuits in the online stage (and, in addition, computing the hash of them twice: once by P_1 and once by P_2).

In case we have an adaptively-secure garbling scheme, P_1 could simply send all garbled circuits straight away, instead of sending commitments on them. That way, all the heavy communication is done in the offline stage, while the remaining online stage communication is *independent* of $|C|$ (and depends only on the input and output lengths, and the security parameters).

As discussed in Section 3, the only known construction of adaptively-secure garbling scheme (with succinct online stage) is in the random-oracle model. Once we work in the random-oracle model, additional optimizations can be used. First, we can use the Free-XOR technique [KS08], which is secure in the random-oracle model, to reduce the garbling of XOR gates (and the communication of sending them). Second, we can implement $\mathsf{Com}(\cdot)$ very efficiently using the random-oracle. Third, we can implement a UC-secure commitment very efficiently using the random-oracle, and by that significantly reduce the overhead of the UC-secure variant of our protocol.

We give a brief description of an adaptively secure garbling in the random-oracle model, which is based on the garbling scheme of [PSSW09]. Let $\mathrm{H}(\cdot)$ be a hash function modelled as a random-oracle. $\mathsf{Garble}(1^k, C)$ consists of the following steps:

- For each wire i of the circuit, choose two random labels $W_i^0, W_i^1 \in_R \{0,1\}^{k+1}$ such that the MSB of W_i^b is b if $i \in \mathsf{Outputs}(C)$, and otherwise, the MSB of $W_i^0 \oplus W_i^1$ is one.

- For each gate g of the circuit, connecting wires i, j to wire n, compute the "garbled gate"

$$\Big\{ \Big(c_0, c_1, \mathrm{H}(W_i^{c_0 \oplus b_i} | W_j^{c_1 \oplus b_j} | g) \oplus W_n^{b_n}\Big) \quad | \quad c_0, c_1 \in \{0,1\}, b_i = MSB(W_i^0), b_j = MSB(W_j^0), \\ b_n = \text{operation}[g](c_0 \oplus b_i, c_1 \oplus b_j) \Big\}.$$

- Pick $\Delta \in_R \{0,1\}^{k+1}$ and set gc be the set of all garbled gates, and en and de as follows:

$$en = \{ \big(W_i^0 \oplus \Delta, W_i^1 \oplus \Delta\big) \mid i \in \mathsf{Inputs}(C) \}$$
$$de = \{ (W_i^0, W_i^1) \mid i \in \mathsf{Outputs}(C) \}$$

Define $\mathsf{Evaluate}(gc, X, \Delta)$ to be the algorithm that XORs all the labels in X with Δ, and then evaluates gc in a topological order. Note that even given gc and X (but not Δ), the evaluator does not posses even a single valid label of gc. Thus, in order to make use of this fact, we require that the evaluator would learn Δ only *after* it chooses its input x. ([BHR12a] presents a similar construction assuming the hash is a UCE-hash. The construction of [BHR12a] is slightly more complicated than the one needed in this paper, though security follows the same reasons.)

To see why this construction is secure against adaptive adversaries, consider the view of the adversary during the game: First, it gets gc, which is just a set of random strings from its point of view. Then, it picks the input x (possibly bit after bit), receives X and at the end receives Δ. While it asks for the labels for x, it still has no information about any label of gc. Therefore, in the simulation, the simulator could program the random-oracle to whatever it likes before giving the adversary the value of Δ. Specifically, it can program it to output the fixed value $C(x)$ (as done with the simulated garbled circuit in the static setting).

Now, the modifications to the protocol from Section 7.1 are the following:

- Let $\mathrm{H}(\cdot)$ be the random-oracle. (As usual, when the random-oracle is instantiated with a hash function, a random string is chosen jointly by the parties and used as a key/prefix to all calls to the hash function in use.)

- Denote Δ of the j-th garbled circuit by Δ_j. Instead of sending the commitments with $\mathsf{PCommit}(\cdot)$ of the j-th garbled circuit and its commitments, P_1 simply sends gc_j, lc_j (where the commitments of lc_j are computed using the commitment $\mathsf{Com}(m,r) = \mathrm{H}(m|r)$). In addition, P_1 commits on Δ_j using $\mathrm{H}(\cdot)$ as well. (Note that now all garbled circuits are sent in the offline stage.)

- P_1 sends Δ_j in two cases: (1) after gc_j is selected to be checked, or, (2) After P_2 has sent its input in the online stage for gc_j. Thus, in the places in which P_1 decommits $\mathsf{PCommit}(\cdot)$, it instead decommits the commitment on Δ_j of the relevant garbled circuit.

Note that now, P_2 learns Δ_j of a bundle only *after* the simulator already knows how it should create the garbled circuit (i.e., whether it should be a valid garbled circuit or a simulated one). Security follows the proof of Theorem 7.4, except that in case P_2 is corrupted, the simulator sends random strings as gc_j, lc_j and also a random string as the commitment on Δ_j. Later, once the simulator knows how to construct the garbled circuit, it picks a random Δ_j, and programs the random-oracle so that gc_j would be a valid garbled circuit in case P_2 asked to check it, or, a fake garbled circuit that always outputs a fixed value z, in case it is used for evaluation. Similarly, the simulator programs the random-oracle for the commitments of lc_j that are revealed.

We remark that a similar construction may be based on UCE-hash functions [BHK13] instead of on the random-oracle. But as said earlier, the only proven instantiation we currently have of UCE-hash functions is in the random-oracle model as well.

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 *CRYPTO*, pages 166–184. Springer, 2013.

[ALSZ13] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *CCS*, pages 535–548. ACM, 2013.

[Bea95] Donald Beaver. Precomputing oblivious transfer. In *CRYPTO*, pages 97–109. Springer-Verlag, 1995.

[BHK13] Mihir Bellare, Viet Tung Hoang, and Sriram Keelveedhi. Instantiating random oracles via uces. In *CRYPTO*, pages 398–415. Springer, 2013.

[BHR12a] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In *ASIACRYPT*, pages 134–153. Springer-Verlag, 2012.

[BHR12b] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *CCS*, pages 784–796. ACM, 2012.

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

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

[DGH$^+$04] Yevgeniy Dodis, Rosario Gennaro, Johan Håstad, Hugo Krawczyk, and Tal Rabin. Randomness extraction and key derivation using the cbc, cascade and hmac modes. In *CRYPTO*, pages 494–510. Springer, 2004.

[DKL$^+$13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and NigelP. Smart. Practical covertly secure mpc for dishonest majority or: Breaking the SPDZ limits. In *ESORICS*, pages 1–18. Springer, 2013.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, pages 643–662. Springer, 2012.

[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 *EUROCRYPT*, pages 537–556. Springer, 2013.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *STOC*, pages 218–229. ACM, 1987.

[HEKM11] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, 2011.

[HKK$^+$14] Yan Huang, Jonathan Katz, Vladimir Kolesnikov, Ranjit Kumaresan, and Alex J. Malozemoff. Amortizing garbled circuits. In *CRYPTO*, 2014.

[HL10] Carmit Hazay and Yehuda Lindell. *Efficient Secure Two-Party Protocols: Techniques and Constructions.* Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.

[HMSG13] Nathaniel Husted, Steven Myers, Abhi Shelat, and Paul Grubbs. Gpu and cpu parallelization of honest-but-curious secure two-party computation. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 169–178. ACM, 2013.

[IKO$^+$11] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, Manoj Prabhakaran, and Amit Sahai. Efficient non-interactive secure computation. In *EUROCRYPT*, pages 406–425. Springer, 2011.

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer — efficiently. In *CRYPTO*, pages 572–591. Springer-Verlag, 2008.

[JS07] Stanislaw Jarecki and Vitaly Shmatikov. Efficient two-party secure computation on committed inputs. In *EUROCRYPT*, pages 97–114. Springer-Verlag, 2007.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. In *ICALP*, pages 486–498. Springer-Verlag, 2008.

[KSS12] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security*, pages 14–14, 2012.

[Lin08] Yehuda Lindell. Lower bounds and impossibility results for concurrent self composition. *Journal of Cryptology*, 21(2):200–249, 2008.

[Lin13] Yehuda Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In *CRYPTO*, pages 1–17, 2013.

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

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

[LP11] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In *TCC*, pages 329–346. Springer, 2011.

[MF06] Payman Mohassel and Matthew K. Franklin. Efficiency tradeoffs for malicious two-party computation. In *PKC*, pages 458–473. Springer, 2006.

[MR13] Payman Mohassel and Ben Riva. Garbled circuits checking garbled circuits: More efficient and secure two-party computation. In *CRYPTO*, pages 36–53, 2013.

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

[NO09] Jesper Buus Nielsen and Claudio Orlandi. Lego for two-party secure computation. In *TCC*, pages 368–386. Springer-Verlag, 2009.

[Nor13] Peter Sebastian Nordholt. *New Approaches to Practical Secure Two-Party Computation.* Institut for Datalogi, Aarhus Universitet, 2013.

[Ped92] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. CRYPTO, pages 129–140. Springer-Verlag, 1992.

[PSSW09] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In *ASIACRYPT*, pages 250–267, 2009.

[PVW08] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In *CRYPTO*, pages 554–571. Springer, 2008.

[SS11] Abhi Shelat and Chih-Hao Shen. Two-output secure computation with malicious adversaries. In *EUROCRYPT*, pages 386–405. Springer, 2011.

[SS13] Abhi Shelat and Chih-hao Shen. Fast two-party secure computation with minimal assumptions. In *CCS*, pages 523–534. ACM, 2013.

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

A Tables and Graphs

In this section, we present detailed tables and graphs describing the number of circuits needed for different parameters (number of executions and bucket size). Recall that the bucket size affects the online execution time, and the overall number of circuits affects both the end-to-end execution time and the offline stage execution time. (In the batch setting, the average number of circuits per execution is the only value of consequence.)

A.1 Examples for Game 1

Tables 2-4 present examples of concrete parameters for Game 1.

Number of executions N	p	**Bucket size**($\lceil B \rceil$)	**Overall number of circuits** ($\lceil B \cdot N/p \rceil$)	**Average # circuits per execution**
1024	0.05	4	81920	80.00
1024	0.1	4	40960	40.00
1024	0.15	5	34134	33.33
1024	0.2	5	25600	25.00
1024	0.25	5	20480	20.00
1024	0.3	5	17067	16.67
1024	0.35	5	14629	14.29
1024	0.4	5	12800	12.50
1024	0.45	5	11378	11.11
1024	0.5	5	10240	10.00
1024	0.55	5	9310	9.09
1024	0.6	5	8534	8.33
1024	0.65	5	7877	7.69
1024	0.7	6	8778	8.57
1024	0.75	6	8192	8.00
1024	0.8	6	7680	7.50
1024	0.85	6	7229	7.06
1024	0.9	7	7965	7.78
1024	0.95	7	7546	7.37
1024	0.96	8	8534	8.33
1024	0.97	8	8446	8.25
1024	0.98	9	9405	9.18
1024	0.99	11	11378	11.11
1024	0.999	34	34851	34.03

Table 2: Concrete parameters for Game 1 and $s = 40$.

Number of executions N	p	**Bucket size**(B)	**Overall number of circuits** ($\lceil B \cdot N/p \rceil$)	**Average # circuits per execution**
1048576	0.05	3	62914560	60.00
1048576	0.1	3	31457280	30.00
1048576	0.15	3	20971520	20.00
1048576	0.2	3	15728640	15.00
1048576	0.25	3	12582912	12.00
1048576	0.3	3	10485760	10.00
1048576	0.35	3	8987795	8.57
1048576	0.4	3	7864320	7.50
1048576	0.45	3	6990507	6.67
1048576	0.5	3	6291456	6.00
1048576	0.55	3	5719506	5.45
1048576	0.6	3	5242880	5.00
1048576	0.65	3	4839582	4.62
1048576	0.7	4	5991863	5.71
1048576	0.75	4	5592406	5.33
1048576	0.8	4	5242880	5.00
1048576	0.85	4	4934476	4.71
1048576	0.9	4	4660338	4.44
1048576	0.95	4	4415057	4.21
1048576	0.96	4	4369067	4.17
1048576	0.97	4	4324025	4.12
1048576	0.98	4	4279903	4.08
1048576	0.99	5	5295839	5.05
1048576	0.999	6	6297754	6.01

Table 3: Concrete parameters for Game 1 and $s = 40$.

Number of executions N	p	**Bucket size**(B)	**Overall number of circuits** ($\lceil B \cdot N/p \rceil$)	**Average # circuits per execution**
32	0.15	6	1280	40.00
32	0.35	7	640	20.00
32	0.55	8	466	14.56
32	0.65	9	444	13.88
32	0.75	10	427	13.34
32	0.8	11	440	13.75
32	0.85	12	452	14.13

Table 4: Concrete parameters for Game 1 and $s = 40$; only the best p for each bucket size is included here.

Figures 2 and 3 show how the probability p affects the size of a bucket and the total number of circuits. We can see from the graphs that, as expected, when p grows, B increases while the total number of balls $\frac{BN}{p}$ decreases. This highlights again the tradeoff between the total work and the online work. Figure 4 shows how the (amortized) number of balls needed per bucket depends on the number of buckets. The more buckets are used, the less number of balls are needed per bucket on average.

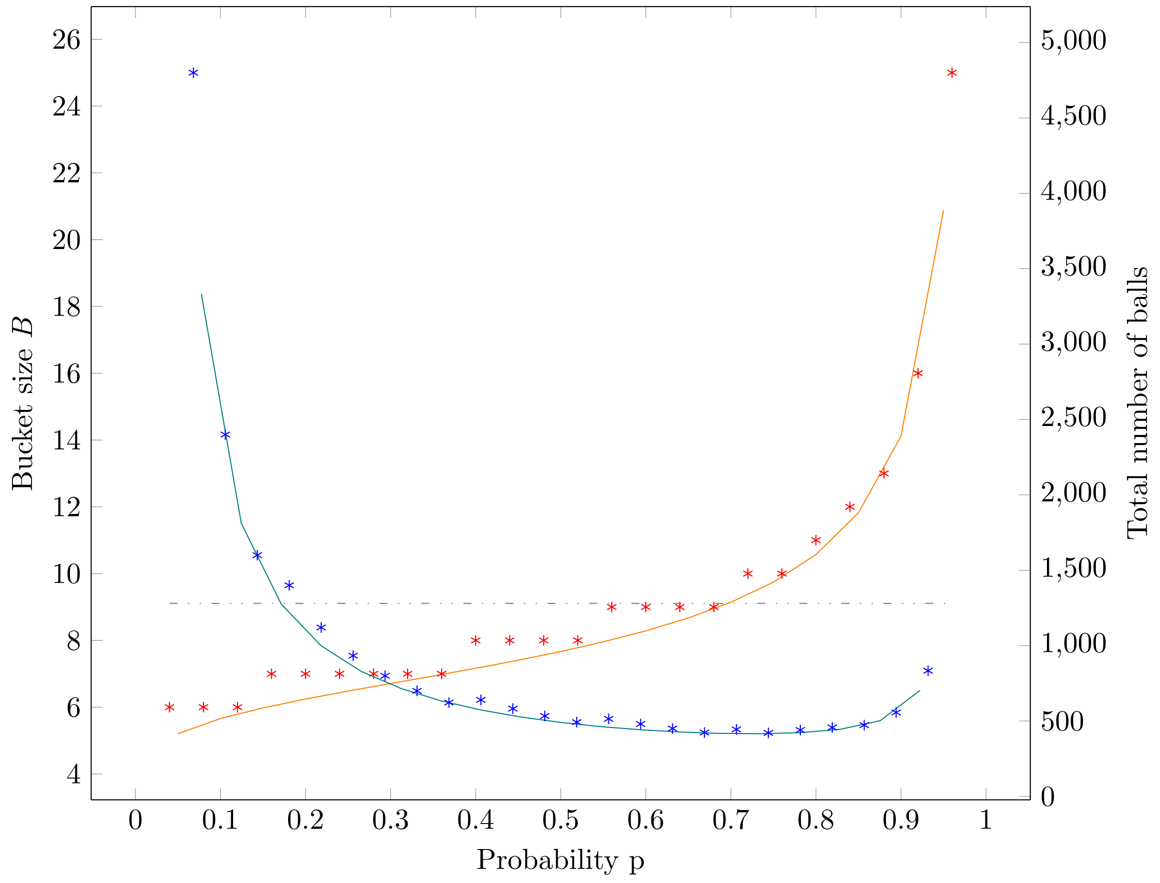

Figure 2: The size of a bucket and the total number of balls depending on the probability p for $s = 40$ and $N = 32$. The orange line represents the value of B without ceiling, i.e., the function $\frac{40+\log 32-\log p}{\log(32(1-p))-\log p/(1-p)}$, and the red asterisks represent values on the line after ceiling. The teal line represents the total number of balls without ceiling of B, i.e., the function $\frac{32}{p}\cdot\frac{40+\log 32-\log p}{\log(32(1-p))-\log p/(1-p)}$ and the blue asterisks represent values of the same function with ceiling, i.e., $\frac{32}{p}\cdot\left\lceil\frac{40+\log 32-\log p}{\log(32(1-p))-\log p/(1-p)}\right\rceil$. The dashed line is the number of balls required by the standard cut-and-choose, as done in [Lin13], i.e., $32 * 40$.

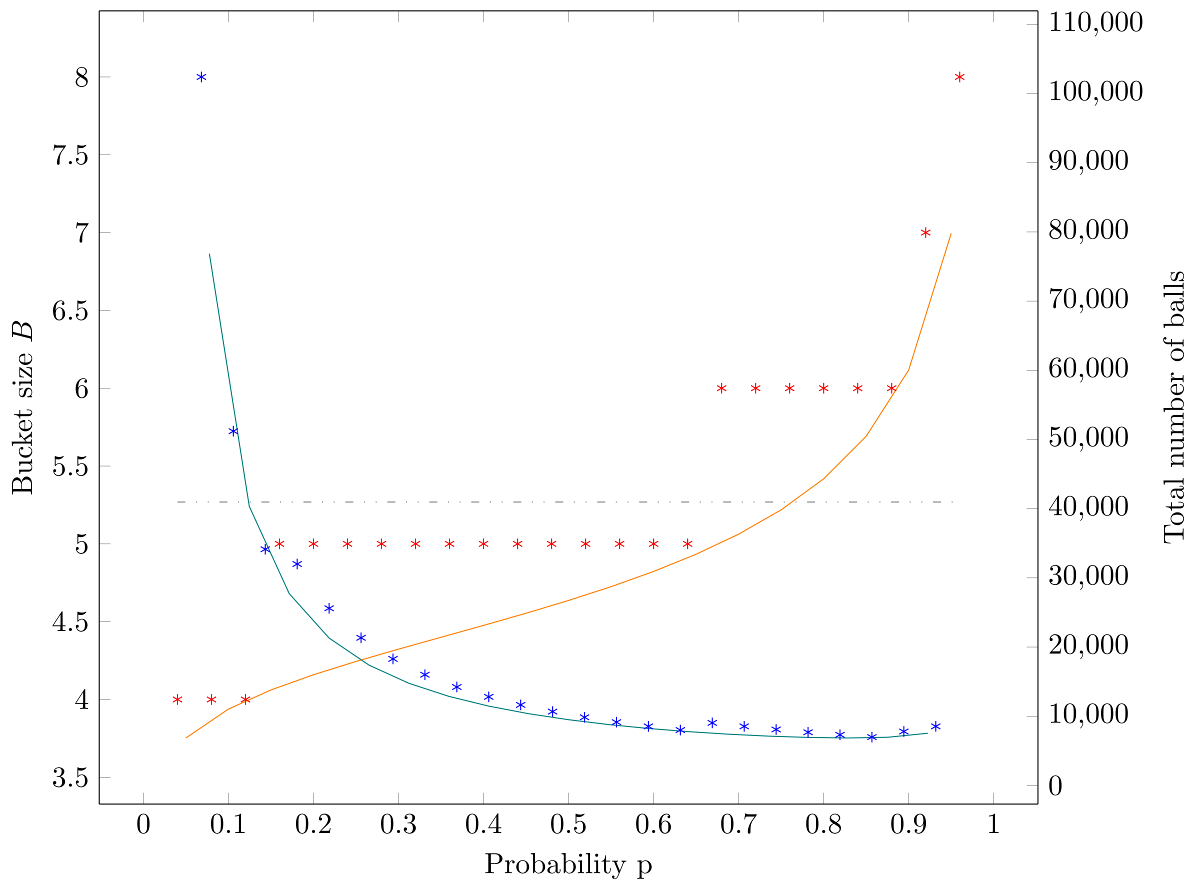

Figure 3: The size of a bucket and the total number of balls depending on the probability p for $s = 40$ and $N = 1024$. The orange line represents the value of B without ceiling, i.e., the function $\frac{40+\log 1024-\log p}{\log(1024(1-p))-\log p/(1-p)}$, and the red asterisks represent values on the line after ceiling. The teal line represents the total number of balls without ceiling of B, i.e., the function $\frac{1024}{p} \cdot \frac{40+\log 1024-\log p}{\log(1024(1-p))-\log p/(1-p)}$ and the blue asterisks represent values of the same function with ceiling, i.e., $\frac{1024}{p} \cdot \left\lceil \frac{40+\log 1024-\log p}{\log(1024(1-p))-\log p/(1-p)} \right\rceil$. The dashed line is the number of balls required by the standard cut-and-choose, as done in [Lin13], i.e., $1024 * 40$.

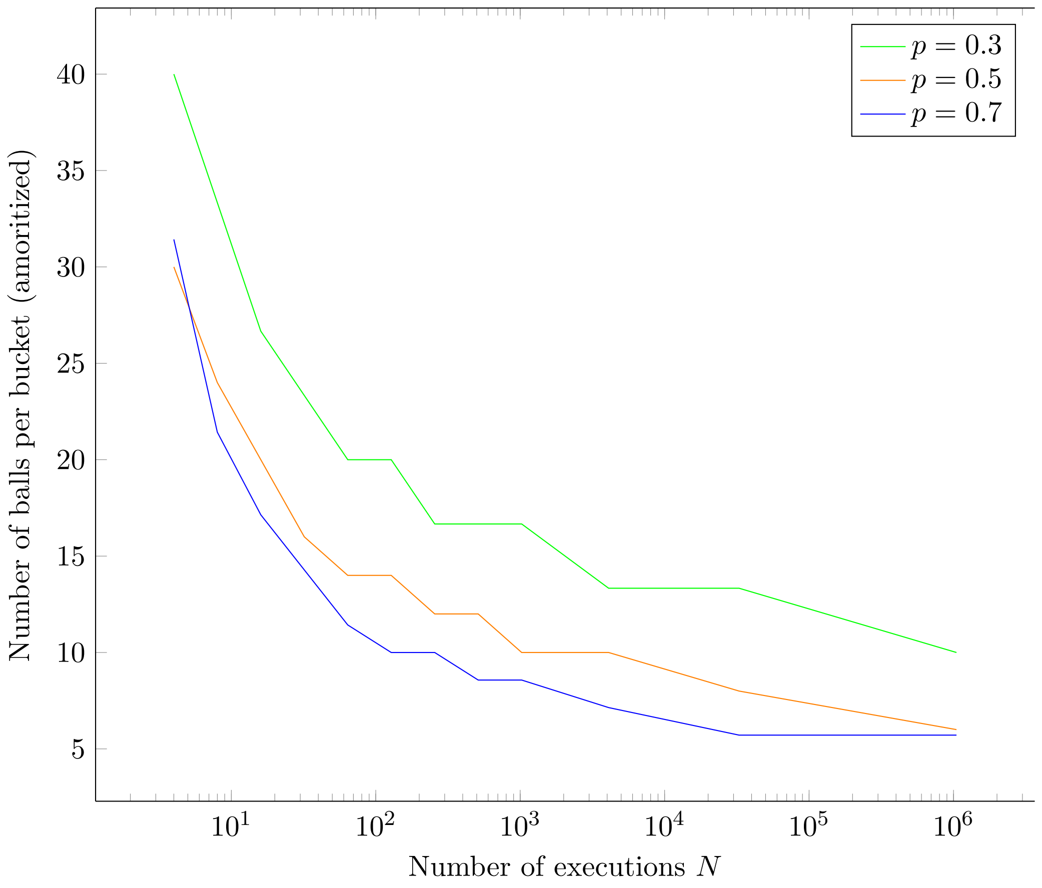

Figure 4: The total number of balls divided by the number of buckets, depending on N. (This is the average number of circuits needed per 2PC execution in our protocol.)

A.2 Examples for Game 2

Tables 5-7 present examples of concrete parameters for Game 2.

Number of executions N	p	**Bucket size**(B)	**Overall number of circuits** $(\lceil B \cdot N/p \rceil)$	**Average # circuits per execution**
1024	0.05	10	204800	200.00
1024	0.1	10	102400	100.00
1024	0.15	10	68267	66.67
1024	0.2	11	56320	55.00
1024	0.25	11	45056	44.00
1024	0.3	11	37547	36.67
1024	0.35	11	32183	31.43
1024	0.4	12	30720	30.00
1024	0.45	12	27307	26.67
1024	0.5	12	24576	24.00
1024	0.55	13	24204	23.64
1024	0.6	13	22187	21.67
1024	0.65	14	22056	21.54
1024	0.7	14	20480	20.00
1024	0.75	15	20480	20.00
1024	0.8	16	20480	20.00
1024	0.85	17	20480	20.00
1024	0.9	19	21618	21.11
1024	0.95	23	24792	24.21
1024	0.96	25	26667	26.04
1024	0.97	28	29559	28.87
1024	0.98	33	34482	33.67
1024	0.99	48	49649	48.49

Table 5: Concrete parameters for Game 2 and $s = 40$.

Number of executions N	p	**Bucket size**(B)	**Overall number of circuits** ($\lceil B \cdot N/p \rceil$)	**Average # circuits per execution**
1048576	0.05	6	125829120	120.00
1048576	0.1	6	62914560	60.00
1048576	0.15	6	41943040	40.00
1048576	0.2	7	36700160	35.00
1048576	0.25	7	29360128	28.00
1048576	0.3	7	24466774	23.33
1048576	0.35	7	20971520	20.00
1048576	0.4	7	18350080	17.50
1048576	0.45	7	16311183	15.56
1048576	0.5	7	14680064	14.00
1048576	0.55	7	13345513	12.73
1048576	0.6	7	12233387	11.67
1048576	0.65	7	11292357	10.77
1048576	0.7	7	10485760	10.00
1048576	0.75	8	11184811	10.67
1048576	0.8	8	10485760	10.00
1048576	0.85	8	9868951	9.41
1048576	0.9	8	9320676	8.89
1048576	0.95	9	9933878	9.47
1048576	0.96	9	9830400	9.38
1048576	0.97	9	9729056	9.28
1048576	0.98	10	10699756	10.20
1048576	0.99	11	11650845	11.11
1048576	0.999	15	15744385	15.02

Table 6: Concrete parameters for Game 2 and $s = 40$.

Number of executions N	p	**Bucket size**(B)	**Overall number of circuits** $(\lceil B \cdot N/p \rceil)$	**Average # circuits per execution**
32	0.05	16	10240	320.00
32	0.1	18	5760	180.00
32	0.15	19	4054	126.69
32	0.2	20	3200	100.00
32	0.25	21	2688	84.00
32	0.3	22	2347	73.34
32	0.35	23	2103	65.72
32	0.4	24	1920	60.00
32	0.45	26	1849	57.78
32	0.5	27	1728	54.00
32	0.55	29	1688	52.75
32	0.6	31	1654	51.69
32	0.65	34	1674	52.31
32	0.7	38	1738	54.31
32	0.75	44	1878	58.69
32	0.8	54	2160	67.50
32	0.85	74	2786	87.06
32	0.9	152	5405	168.91

Table 7: Concrete parameters for Game 2 and $s = 40$.