Cut-and-Choose Yao-Based Secure Computation in the Online/Offline and Batch Settings*

Yehuda Lindell and Ben Riva

Dept. of Computer Science
Bar-Ilan University, Israel
lindell@biu.ac.il. benr.mail@gmail.com

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 in the batch and online/offline settings to $\mathcal{O}\left(\frac{s}{\log N}\right)$ garbled circuits when N secure computations are run. 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 reduces 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.

* 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 under the European Union's Seventh Framework Program (FP7/2007-2013) under grant agreement n. 609611 (PRACTICE). A full version of this work appears in the *Cryptology ePrint Archive*, 2014.

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

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

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.*

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 checked 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.

[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.

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 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 $F(\boldsymbol{x}, \boldsymbol{y}) = (f(x_1, y_1), \ldots, f(x_N, y_N))$ where $\boldsymbol{x} = (x_1, \ldots, x_N)$ and $\boldsymbol{y} = (y_1, \ldots, y_N)$. 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 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 the full version, 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.

1.3 Organization

Due to the lack of space in this abstract, we provide only an outline and high-level description of our techniques. A full description of our protocols, proofs of security, and a full combinatorial analysis of the number of circuits needed appears in the full version.

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 3 and in the full version 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.

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

[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.

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 3 we analyse the above parameters in the general case. As we will see, better parameters are typically achieved with lower 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 phases. This is due to the fact that checking more circuits in the offline stage reduces the number of circuits to be evaluated in the online stage 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 3 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 the full version of this paper, 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 the full version of the paper 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] 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 checked 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 the full version.

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 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

[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.

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 checked

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

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 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). We follow the analysis of [Nor13, Theorem 4.4] and [Nor13, Theorem 6.2], while handling different and slightly more general parameters.

3.1 The Fully-Cracked Bucket Game

Let $\mathsf{Game}\ 1$ 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 $\mathcal{A}$) 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

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.

$M = \left\lceil \frac{NB}{p} \right\rceil$ and $m = NB$ and so are not separate parameters in the game). The proof of the following theorem can be found in the full version of this paper:

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

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

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

We remark that in the proof of Theorem 1 we show that the probability that the adversary wins in the game is at most

$$\frac{\binom{M-t}{m-t}}{\binom{M}{m}} \cdot N \cdot \binom{t}{B}\binom{m}{B}^{-1} \tag{2}$$

and then proceed to show that this is less than 2^{-s} as long as Eq. (1) holds, for general parameters. However, for concrete sets of parameters we can compute slightly tighter bounds or more optimized parameters. For example, Theorem 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 1 requires B to be 10. The maximum of Eq. (2) for these parameters is at most 2^{-44}, which, again, is much smaller than the 2^{-40} bound given by Theorem 1. In fact, if we take $N = 32$, $p = 0.8$ and $B = 10$, we get that the maximum of Eq. (2) is at most $2^{-40.1}$, without increasing B as required if we had used Theorem 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. More examples of the parameters obtained and discussion on recommended values appears in the full version of the paper.

3.2 The Majority-Cracked Bucket Game

Let Game 2 be the same game as Game 1, but where $\mathcal{A}$ 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. (Recall that $\mathsf{Game}_1(\mathcal{A}, N, B, p) = 1$ only if *all* of the balls in some bucket are cracked.)

In the full version of this paper, we prove the following theorem:

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

In the full version, we discuss in depth what parameters this yields with $s = 40$. Briefly, 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 obtain 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]).

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.

[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.

[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.

[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.

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

[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.

[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.

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

[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.