Blazing Fast 2PC in the Offline/Online Setting with Security for Malicious Adversaries*

Yehuda Lindell† Ben Riva†

June 21, 2016

Abstract

Recently, several new techniques were presented to dramatically improve key parts of secure two-party computation (2PC) protocols that use the cut-and-choose paradigm on garbled circuits for 2PC with security against malicious adversaries. These include techniques for reducing the number of garbled circuits (Lindell 13, Huang et al. 13, Lindell and Riva 14, Huang et al. 14) and techniques for reducing the overheads besides garbled circuits (Mohassel and Riva 13, Shen and Shelat 13).

We design a highly optimized protocol in the offline/online setting that makes use of all state-of-the-art techniques, along with several new techniques that we introduce. A crucial part of our protocol is a new technique for enforcing consistency of the inputs used by the party who garbles the circuits. This technique has both theoretical and practical advantages over previous methods.

We present a prototype implementation of our new protocol, which is also the first implementation of the amortized cut-and-choose technique of Lindell and Riva (Crypto 2014). Our prototype achieves a speed of just 7 *ms in the online stage* and just 74 ms in the offline stage per 2PC invoked, for securely computing AES in the presence of malicious adversaries (using 9 threads on two 2.9GHz machines located in the same Amazon region). We note that no prior work has gone below one second overall on average for the secure computation of AES for malicious adversaries (nor below 20ms in the online stage). Our implementation securely evaluates SHA-256 (which is a *much bigger circuit*) with 33 ms online time and 206 ms offline time, per 2PC invoked.

1 Introduction

Secure two-party computation enables a pair of parties with private inputs 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 [36, 14].

*An extended abstract of this work appeared at *ACM CCS* 2015. This is the *preliminary* full version. In previous versions of this work, the bounds in Lemmas 2.3 and 2.4 were erroneously copied from [26]. This has been fixed.

†Department of Computer Science, Bar-Ilan University, Israel. Email: lindell@biu.ac.il, benr.mail@gmail.com. Supported by the European Research Council under the ERC consolidators grant agreement n. 615172 (HIPS), and by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minster's Office.

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. One of the most popular approaches for constructing such protocols 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. If yes, then the parties 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). Many different works 2PC protocols have been designed based on this approach [27, 23, 25, 33, 16, 22, 28, 34], and several implementations have been presented to study the concrete efficiency of it in practice (e.g., [31, 33, 20, 34, 2]). In this work we focus on the cut-and-choose approach.

The number of garbled circuits. Let s be a statistical security parameter such that the probability that a malicious party can cheat should be bounded by 2^{-s} (plus a function that is negligible in n, where n is the computational security parameter). Then, the exact number of garbled circuits needed for achieving this bound was reduced in the past years from $17s$ [23], to approximately $3s$ [25, 33], and recently to s [22].

In [17, 26], it was shown that if multiple 2PC executions are needed, then the amortized number of garbled circuits per 2PC can be reduced even below s. Specifically, for N 2PC executions, only $\mathcal{O}(\frac{s}{\log N})$ garbled circuits are needed per 2PC. In addition, [17, 26] present protocols that work in the online/offline setting, where most of the computation and communication intensive steps are carried out in the offline stage, resulting in a very efficient online stage. In [26], a tight analysis was presented showing that opening significantly less than half of the circuits (depending on the parameters) can yield very significant concrete efficiency improvements.

Checking input consistency and preventing selective OT attacks. Running cut-and-choose itself does not suffice for obtaining a secure protocol since it only deals with the correctness of the garbled circuits. To make the protocol secure, we must additionally include mechanisms for ensuring that the party that prepares the garbled circuits **(a)** uses the same input in all the evaluated garbled circuits, and **(b)** provides correct inputs to the OTs for the other party to learn the input labels for its input. We refer to the first problem as *P_1's input consistency check* and the second as preventing a *selective OT attack*. (We note that it is easy to ensure that the party P_2 who evaluates the garbled circuits uses the same input in all circuits, by running a single OT for each bit of P_2's input for all circuits being evaluated. We therefore do not refer to this problem further.)

It is possible to check the consistency of P_1's input using $\mathcal{O}(s^2)$ inexpensive (symmetric) cryptographic operations per input bit [27, 23], but this results in huge communication. Alternate solutions using $\mathcal{O}(s)$ exponentiations per input bit were presented in [25, 33]; this reduces the communication size while significantly increasing the computation time. Recently, [34, 28] presented solutions that require only $\mathcal{O}(s)$ inexpensive (symmetric) cryptographic operations *per input bit*, resulting in only a minor overhead on top of the cut-and-choose protocol itself (as it already requires $\mathcal{O}(s)$ gates per input bit).

Selective OT attacks can be prevented by using a special encoding of P_2's input so that the leakage of a small number of bits of the encoded input reveals nothing about the actual input [23, 34]. We discuss this solution in more detail in Section 2.3. An alternative solution (using cut-and-choose OT) was presented in [25], but this requires many exponentiations and so is not as efficient.

Implementations of cut-and-choose based 2PC. The first implementation which evaluated the cut-and-choose approach in practice was [31]. In [33, 34], implementations with additional algorithmic improvements were presented. Both results focus on reducing the overheads of the input-consistency checks, and work with approximately $3s$ garbled circuits for soundness 2^{-s}. In [20], the protocol of [33] is implemented using mass parallelism, resulting in a system that utilizes a cluster of several hundreds of machines in parallel. Parallelism was taken a step further in [12, 11], who designed and implemented protocols on GPUs.

The fastest published secure computation of AES based on *cut-and-choose on garbled circuits*, that we are aware of, in the single-execution, non-massively concurrent setting is of [2]. This implementation requires approximately 6.39 seconds for a single evaluation of AES. However, massive concurrency can drastically improve performance. Using several tens of machines (each with 8 CPU cores), AES can be computed in about 40.6 seconds for 1024 executions, with security parameter $s = 80$ [34]. Using GPUs, AES can be computed in only 0.46 seconds, for $s = 40$ [11].

1.1 Our Contributions

We start by presenting a new technique for checking that P_1 uses the same input in all (good) garbled circuits. Our method has both theoretical and practical advantages over previous techniques. Then, we describe an optimized protocol for 2PC in the online/offline setting, based on the protocol of [26]; our protocol uses our new consistency check, plus the state-of-the-art techniques for the other checks and additional small optimizations. We present a prototype implementation of our optimized protocol, which is the first implemented 2PC protocol based on the cut-and-choose method that requires less then s garbled circuits per 2PC computation. Last, we evaluate the prototype with different circuits and sets of parameters. We proceed to provide more details on each contribution.

New P_1's input consistency check. Previous techniques for ensuring that P_1 uses the same input in all good garbled circuits have significant disadvantages. The best known methods to date require $\mathcal{O}(s)$ symmetric cryptographic operations per input bit, and are due to [28] and [34]. However, it is unclear how to use the technique of [34] in the online/offline setting (when many 2PC executions are needed), and the technique of [28] is (arguably) complicated and thus very difficult to implement.

Our new solution requires $\mathcal{O}(s)$ symmetric cryptographic operations *per garbled circuit*, rather than *per input bit*; in most cases this is much smaller, and especially in the offline/online setting where the number of circuits per execution is very small (about 5-10 for typical parameters). In addition, our solution is very simple to describe and implement, and can be plugged-in in a modular way into most 2PC protocols (based on the cut-and-choose method), including the ones in the online/offline setting. Our protocol can be implemented using only standard cryptographic assumptions (at the expense of adding 2 exponentiations per circuit which is negligible in the overall cost) or in the random oracle model (in which case no exponentiations are needed). We remark that our new consistency check is the best option today, even for single-execution protocols.

Optimized protocol in the ROM. We apply the new technique for checking P_1's input consistency and the *randomized encoding* technique of [23] for protecting against selective OT attacks, to the protocol of [26] in the online/offline setting. We further optimize several parts of the protocol in the random-oracle model, including further elimination of exponentiations, reducing communication, and more.

The online stage of the protocol is highly efficient. It requires only four messages between the players and the overall communication size depends only on the input length and the security parameters. (Note that the online stage of the fastest 2PC implementation in the online/offline setting, shown in [29], requires a number of rounds that depends on the depth of the circuit in use, and its communication size depends on the circuit size.) This is the first implemented protocol with online communication that is *independent of the circuit size* (and is concretely very small, as shown by our experiments).

Prototype implementation and evaluation. We implemented our optimized protocol on top of the SCAPI library [10, 1]. Our prototype uses state-of-the-art techniques like AES-NI instructions, fixed-key garbling [5], and the optimized OT-extension protocol of [3]. We evaluated the prototype on Amazon AWS machines. Performance of the online stage itself is three orders of magnitude better than previous protocols (without massive parallelism). For example, evaluating the AES circuit between two machines in the same region costs only 7 ms in the online stage. Furthermore, the offline stage costs only 74 ms per 2PC computation (for some sets of parameters). Even when the parties communicate via the Internet, the cost of the online stage remains small as our protocol requires only four rounds of communication. Specifically, we evaluated AES in 160 ms with a network roundtrip of 75 ms (so at least 150 ms is spent on communication). Observe that the offline stage itself is very competitive when compared to previous results. In particular, the *sum* of both the offline and online stages is far better than any single execution reported (81ms only). Thus, we do not obtain a fast online phase at the expense of a slow offline one. See Section 6 for more details and a comparison of our results with the performance of previous implementations.

2 Preliminaries

Let $\mathsf{H}(\cdot)$ denote a hash function, and $\mathsf{commit}(x)$ (resp., $\mathsf{commit}(x, r)$) denote a commitment to x (resp., a commitment to x using randomness r). We denote by l the length of each party's input, by $\mathsf{In}(C, x)$ the set of wire indexes of a boolean circuit C that correspond to a given input x, and by $\mathsf{Out}(C)$ the set of wire indices of the output wires of C.

2.1 Efficient Perfectly/Statistically-Hiding Extractable Commitment

Let $\mathsf{ExtractCom}(m)$ be a perfectly- or statistically-hiding extractable commitment. We assume that in the first phase of the scheme the simulator generates a trapdoor td that enables immediate extraction of commitments later on. In Appendix A we review the perfectly-hiding extractable commitment of [21] that works in the standard model and is secure under the DDH assumption. In the random-oracle model, such a commitment is trivial; we just define $\mathsf{ExtractCom}(m) = \mathsf{H}(m; r)$ where r is random. Note that this is not *perfectly* hiding. However it is statistically hiding (in the random oracle model) to any algorithm who can make only a polynomial number of queries to H, and this suffices for our needs.

2.2 Adaptively-Secure Garbling

The standard security notion of garbled circuits (e.g., [24]) deals with a *static adversary*, meaning that the adversary picks its input before seeing the garbled circuit. However, in the online/offline setting, inputs are chosen only in the online stage, and if we wish to send all garbled circuits in the offline stage then the static security notion does not suffice. (Note that it is possible to only commit to the garbled circuits in the offline phase. However, in order to achieve the necessary security here,

the decommitment would be the same size as the circuit, resulting in significant communication.) The security of garbled circuits in the presence of an *adaptive adversary* was defined in [6]; in this definition, the adversary first gets the garbled circuit and only then chooses its input. As discussed in [26], this allows proving security in the online/offline setting, even if all garbled circuits are sent in the offline stage.

We use the method described in [26] that slightly modifies the fixed-key AES-NI garbling scheme of [5] to be adaptively secure in the *random-permutation model*. Adaptive security is immediate in the (programmable) random-permutation model if P_2 (the evaluator) chooses its input in a single query. However, this is not true in case P_2 can obtain some valid input labels before all its input bits are chosen (and therefore evaluate some of the gates before the input is fully determined). This is a problem since the gates need to be "programmed" (in the random-oracle/random-permutation model) *after* the inputs are received. This is solved by ensuring that P_2 is unable to decrypt any gate before receiving all labels. We achieve this by having P_1 choose a random λ (of the same length as the garbled labels), and whenever P_2 should learn a label for some input bit, it actually learns the label XORed with λ. After P_2 receives all the garbled values (XORed with λ), party P_1 reveals λ, and then P_2 can obtain its labels and evaluate the circuit. (The value of λ can be viewed as part of the last label, which will be longer than the previous ones).

We recall the security game for an adaptively secure garbling scheme: Let ℓ be the input length. The challenger picks a coin b. If $b = 0$ the adversary receives a valid garbled circuit, and if $b = 1$ the simulator $\mathsf{Sim}(1, f)$ is run to generate a "fake" garbled circuit which is handed to the adversary. Then, for $i = 1, \ldots, \ell$, the adversary sends a bit x_i and receives its input label associated with the ith input bit. If $b = 0$ then the input label is correctly generated, whereas if $b = 1$ then it is generated by calling $\mathsf{Sim}(2, f, i, y)$, where $y = \perp$ if $i < \ell$ and $y = f(x_1, \ldots, x_\ell)$ otherwise. The goal of the adversary is to correctly guess b with probability greater than $1/2$.

In order to prove security of our scheme described above (using λ), we need to specify the computation of Sim and show that the probability that the adversary guesses b correctly is $\frac{1}{2} + neg(|\lambda|)$. $\mathsf{Sim}(1, f)$ chooses a random string as the garbled circuit. Then, for $i = 1, \ldots, \ell - 1$, $\mathsf{Sim}(2, f, i, y)$ chooses a random string for the input label. Finally, for $i = \ell$, $\mathsf{Sim}(2, f, i, y)$ also chooses a random λ, and programs the random-oracle/random-permutation so that for the used set of labels and the string that represented the garbled circuit, the garbled circuit is a fixed garbled circuit that always returns y. Observe that the distribution of the garbled circuit and the input labels is the same in real game and the simulated one. The only case in which the adversary can distinguish between the two is in case it guesses correctly one of the labels of the garbled circuit (either of the input wires, or internal ones), and then notice the programming of the oracle. However, in order to do that, the adversary must guess correctly some valid label or λ, and both happen with probability that is negligible in $|\lambda|$.

2.3 The Solution of [23] for Selective-OT Attacks

A solution for the selective-OT attack, which works with any oblivious transfer in a black-box way, was presented in [23]. The solution works by encoding 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 carried out using a Boolean matrix E that is s-probe-resistant, as defined below.

Definition 2.1 (Based on [23, 34]) *Matrix $E \in \{0,1\}^{l \times n}$ for some $\ell, n \in \mathbb{N}$ is called s-*probe-resistant *for some $s \in \mathbb{N}$ if for any $L \subset \{1, 2, \ldots, \ell\}$, the Hamming distance of $\bigoplus_{i \in L} E_i$ is at least s, where E_i denotes the i-th row of E.*

In [23], it is shown that such a matrix E can be constructed with $n = \max(4\ell, \frac{20s}{3})$, where ℓ is P_2's input length. Then, [34] show a different construction with $n \leq \lg(\ell) + \ell + s + s \cdot \max(\lg(4\ell), \lg(4s))$. For completeness, we describe these constructions in Figure 2.2. We note that both constructions can result in a matrix E for which there exists a vector y that for all vectors y', $Ey' \neq y$ (meaning that some input cannot be encoded). We therefore take E to be $[E'|I_\ell] \in \{0,1\}^{\ell \times (n+\ell)}$, where E' is an s-probe-resistant matrix and I_ℓ is the identity matrix of size ℓ. E is clearly also s-probe-resistant, and now, any vector y can be encoded using a vector y' that has random bits in the first n elements, and "corrections" in the rest of the bits so that $Ey' = y$.

FIGURE 2.2 (Generating s-probe-resistant matrix)

The construction of [23]:

- Set $n = \max(4\ell, \lceil \frac{20s}{3} \rceil)$.
- For $i = 1, \ldots, \ell$, let $E_i \in_R \{0,1\}^n$.
- Let E be the matrix in which its ith row is E_i.

The construction of [34]:

- Set $t = \lceil \max(\lg(4\ell), \lg(4s)) \rceil$.
- Decrease t by one until $2^{t-1} > s + \frac{\lg(\ell)+\ell+s}{t-1}$.
- Set $k = \lceil (\lg(\ell) + \ell + s)/t \rceil$.
- For $i = 1, \ldots, \ell$, choose a random polynomial $P_i(x)$ of degree $k-1$ over $\mathbb{F}_{2^t}$ (e.g., by choosing at random its k coefficients), and let E_i be the concatenation of the values $P_i(1), P_i(2), \ldots, P_i(k+s-1)$.
- Let E be the matrix in which its ith row is E_i. (Note that $n = t(k+s-1)$.)

Instead of working with the function $f(x,y)$, the parties work with the function $f'(x,y') = f(x,Ey')$ and P_2 chooses a random $y' \in \{0,1\}^{n+\ell}$ such that $y = Ey'$ (this ensures that $f'(x,y') = f(x,y)$). As long as E is s-probe-resistant, even if P_1 learns $s' < s$ bits of y', it cannot learn any information about y. This is due to the fact that for every $s' < s$ bits of y' and every y, there exists a y'' that is consistent with them (i.e., $Ey'' = y$ and $y'' = y'$ on the $s' < s$ bits revealed). Now, in order to learn s bits, P_1 has to carry out a selective-OT attack on s wires (meaning that for s wires it provides one valid OT input and one invalid OT input, and if no abort occurs then it knows that the valid input was chosen). However, for every such wire it is caught with probability $1/2$, which 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 $n + \ell$ for the entire set of evaluated circuits.

As described in [34], since E is a binary matrix the subcircuit that computes Ey' can be garbled using *only* XOR gates. This is due to the fact that E is fixed (and known), and so multiplying a row of E with y' is the same as XORing a certain subset of y''s bits together. This is therefore very efficient when using the Free-XOR technique [19]. Moreover, assuming correlation-robust hash functions, many OTs can be implemented very efficiently (i.e., with a small number of symmetric-key operations per OT) using an efficient OT extension protocol. Specifically, the above solution can be implemented with $\mathcal{O}(n)$ symmetric-key operations, and only $\mathcal{O}(s)$ seed-OTs [3].

2.4 Cut-and-Choose Parameters

The offline/online method for cut-and-choose uses the following parameters: **(a)** the number of circuits B evaluated per online 2PC; **(b)** the number of 2PC executions N; **(c)** the fraction of circuits evaluated p (and so a $1-p$ fraction are checked); and **(d)** the statistical security parameter s. Note that the overall number of circuits used is $\frac{NB}{p}$, since after checking a $(1-p)$-fraction the number of circuits that remain are $\frac{NB}{p} - (1-p)\cdot\frac{NB}{p} = \frac{NB-NB+p\cdot NB}{p} = NB$, which is what is required to have B circuits per execution. A comprehensive analysis of the soundness of the amortized cut-and-choose, with respect to the above parameters, was presented by [26].

For completeness, we repeat the description of the cut-and-choose game in terms of balls and bins. From here on, a ball refers to a garbled circuit, and a cracked ball is an incorrect garbled circuit that was maliciously generated; a single execution in the online phase uses a full bucket of unchecked "balls". Recall that in the cheating recovery method of [22] there are actually two garbled-circuit evaluations: the main circuit for computing the function is evaluated, and a very small auxiliary circuit is computed that is used for P_2 to learn P_1's input in case P_1 cheated. The balls and bins game is such that for the main circuit the adversary can cheat if there exists a bucket where *all* the balls are cracked, and for the small cheating recovery circuit the adversary can cheat if there is a bucket where a *majority* of the balls are cracked.

P_2 chooses three parameters p, N and B, and sets $M = \left\lceil \frac{NB}{p} \right\rceil$ and $m = NB$. (Note that $p < 1$ and so $M > m$.) 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. Denote the balls that are not checked by $1, \ldots, m$. Then, P_2 chooses a random mapping function $\pi : [m] \to [N]$ that places the unchecked balls in buckets of size B.

In [26], bounds are proven for the probabilities that P_2 does not abort and **(1)** there exists a fully-cracked bucket (i.e., all balls in some bucket are cracked), or **(2)** there exists a majority-cracked bucket (i.e., at least $B/2$ balls in some bucket are cracked).

When considering multiple executions of protocols (as in [26]), the *overall* cheating probability is the natural soundness one should work with. However, we believe that is it preferable to focus on the cheating probability in a *single 2PC execution* since this enables a direct comparison to single-execution implementations. We will therefore be interested in the probabilities that P_2 does not abort and *some specific* bucket is fully cracked, or, that *some specific* bucket is majority-cracked.

In addition to the bounds shown in [26], a few concrete examples are presented there to exemplify that those bounds are not tight. Since we care here about concrete efficiency, we implemented a program that finds the parameters analytically, based on the following tighter computations that are derived from the analysis of [26] but are not informative within themselves (in previous versions of this paper, the multiplicative factor N was mistakenly omitted from the bounds).

Lemma 2.3 *Let N, B, p, M be parameters as described above. The probability that a bucket is* fully-cracked *is at most*

$$\max\left[\frac{\binom{M-t}{NB-t}}{\binom{M}{NB}}\cdot N\cdot\binom{t}{B}\binom{NB}{B}^{-1}\right]_{t=B}^{NB}.$$

Lemma 2.4 *Let N, B, p, M parameters as described above. The probability that a bucket is* majority-cracked *is at most*

$$\max\left[\frac{\binom{M-t}{NB-t}}{\binom{M}{NB}}\cdot N\cdot 2^{B-1}\left(\frac{t}{NB}\right)^{\lceil B/2\rceil}\right]_{t=B}^{NB}.$$

See Tables 1 and 2 for concrete examples of parameters computed according to Lemma 2.3.

3 Commitment with ZK Proof of Difference

The aim of this section is to construct a commitment scheme with an efficient zero-knowledge proof of difference. Given $\mathsf{commit}(x_1)$, $\mathsf{commit}(x_2)$ and $\Delta = x_1 \oplus x_2$, the aim is to efficiently prove that the XOR of the decommitments is indeed Δ. Formally, one party inputs (x_1, x_2), and the other party chooses to either learn $x_1 \oplus x_2$ or the pair (x_1, x_2) itself. (Thus, the first party is essentially committed to the pair, and must either decommit or prove their difference, depending on P_2's choice.) We will show later how it is used to prove P_1's input consistency Our constructions are based on ideas of [32].

We start by describing a basic functionality (presented in Figure 3.1), prove its correctness, and then describe how to extend it to work with many commitments so we can use it for input-consistency checks.

FIGURE 3.1 (The Simple Commit-and-Difference Proof Functionality $\mathcal{F}_{\mathsf{Com\Delta ZK}}$)

$\mathcal{F}_{\mathsf{Com\Delta ZK}}$ runs with parties P_1 and P_2, as follows:

Input: $\mathcal{F}_{\mathsf{Com\Delta ZK}}$ receives a pair of messages (x_1, x_2) from P_1, and a bit b from P_2.

Output:

- $\mathcal{F}_{\mathsf{Com\Delta ZK}}$ sends $\Delta = x_1 \oplus x_2$ to P_2 if $b = 0$; otherwise (if $b = 1$), it sends (x_1, x_2) to P_2.
- $\mathcal{F}_{\mathsf{Com\Delta ZK}}$ sends b to P_1.

3.1 A Warm-Up – Only Two Messages

In this section, we show how to securely realize the functionality from Figure 3.1. The basic idea for the construction is as follows. We define a split commitment of a value x to be a pair of commitments to random values whose XOR equals x; i.e., $[\mathsf{commit}(x \oplus r), \mathsf{commit}(r)]$. Party P_1 sends P_2 a set of s split commitments to x_1 and s split commitments to x_2. If P_2 asks to decommit (i.e., $b = 1$) then P_1 simply decommits using the standard (canonical) decommitment and P_2 checks that there exist x_1, x_2 such that all the split commitments are as expected. In contrast, if $b = 0$, then P_1 sends P_2 the XORs of the split commitment values. Specifically, let $[\mathsf{commit}(x_1 \oplus r_i), \mathsf{commit}(r_i)]$ and $[\mathsf{commit}(x_2 \oplus \rho_i), \mathsf{commit}(\rho_i)]$ be the ith split commitment of x_1 and x_2, respectively. Then, P_1 sends $\delta_i^0 = x_1 \oplus r_i \oplus x_2 \oplus \rho_i$ and $\delta_i^1 = r_i \oplus \rho_i$ to P_2, for every $i = 1, \ldots s$, as well as $\Delta = x_1 \oplus x_2$. Observe that for every i it holds that $\delta_i^0 \oplus \delta_i^1 = x_1 \oplus x_2 = \Delta$, and so P_2 checks that for every i it indeed holds that $\delta_i^0 \oplus \delta_i^1 = \Delta$. Then, given these values, P_2 sends a random s-bit "challenge string" W to P_1, indicating to P_1 which value in each split commitment to open. Letting $W = W_1, \ldots, W_s$, party P_1 decommits to both *left* commitments in the ith split commitments of x_1 and x_2 if $W_i = 0$; otherwise it decommits to both *right* commitments in the ith split commitments of x_1 and x_2. Observe that if $W_i = 0$ then P_2 receives $x_1 \oplus r_i$ and $x_2 \oplus \rho_i$ and so can verify that δ_i^0 was correctly constructed. In contrast, if $W_i = 1$ then P_2 receives r_i and ρ_i and so can check that δ_i^1 was correctly constructed. Thus, if $x_1 \oplus x_2 \neq \delta$, then P_1 must cheat on at least one side of every split commitment, and so will be caught with probability $1 - 2^{-s}$. Observe that this check is very simple and very efficient; when using a hash function to commit it requires $2s$ hash computations only per value.

Despite its simplicity, we remark that in order to *simulate* this protocol (in the sense of securely computing Functionality $\mathcal{F}_{\mathsf{Com\Delta ZK}}$ in the ideal/real model paradigm), we need to have P_2 commit to its challenges b and W before the protocol begins. If an *extractable* commitment is used, then

the simulator can learn the challenges ahead of time and therefore appropriately prepare the commitment, even before Δ is known. Fortunately, this comes at very little overhead, as can be seen in the full protocol. See Figure 3.2 for the detailed protocol.

FIGURE 3.2 (Two Commitments with Proof of Difference)

Inputs: P_1 has a pair (x_1, x_2) and P_2 has a bit b.

Commit to Challenge: P_2 chooses a random $W \in \{0,1\}^s$. Then, P_1 and P_2 run a *perfectly (or statistically) hiding extractable commitment* scheme $\mathsf{ExtractCom}$ (see Section 2.1), in which P_2 commits to b and W. (A Pedersen commitment with a zero-knowledge proof of knowledge of the committed value suffices, or a simple hash with a random string in the random oracle model.)

Commit to x_1, x_2: Define the split commitment $\mathsf{SCom}(x, r) = [\mathsf{commit}(x \oplus r), \mathsf{commit}(r)]$. Then:

1. For $i = 1, \ldots, s$, P_1 chooses $r_i, \rho_i \leftarrow \{0,1\}^n$ and computes $\left[c_i^0, c_i^1\right] = \mathsf{SCom}(x_1, r_i)$ and $\left[d_i^0, d_i^1\right] = \mathsf{SCom}(x_2, \rho_i)$.
2. Denote $c_1 = SC(x_1) = \langle [c_1^0, c_1^1], \ldots, [c_s^0, c_s^1] \rangle = \langle \mathsf{SCom}(x_1, r_1), \ldots, \mathsf{SCom}(x_1, r_s) \rangle$, and
   $c_2 = SC(x_2) = \langle [d_1^0, d_1^1], \ldots, [d_s^0, d_s^1] \rangle = \langle \mathsf{SCom}(x_2, \rho_1), \ldots, \mathsf{SCom}(x_2, \rho_s) \rangle$

P_1 sends $(\mathsf{commit}, c_1, c_2)$ to P_2.

Decommit to b: P_2 decommits to the value b using the Decom procedure of $\mathsf{ExtractCom}$. (At the end of the protocol, P_1 outputs b.)

If $b = 1$, party P_1 decommits to x_1, x_2: P_1 sends x_1, x_2 and all of the randomness used to generate the commitments c_1, c_2. P_2 verifies that all commitments were correctly constructed, and if yes it outputs (x_1, x_2).

If $b = 0$, party P_1 provides a proof of difference:

1. For every $i = 1, \ldots, s$, party P_1 defines $\delta_i^0 = x_1 \oplus r_i \oplus x_2 \oplus \rho_i$ and $\delta_i^1 = r_i \oplus \rho_i$ (note that $\delta_i^0 \oplus \delta_i^1 = x_1 \oplus x_2$).
2. P_1 sends $\left\{(\delta_i^0, \delta_i^1)\right\}_{i=1}^s$ and $\Delta = x_1 \oplus x_2$ to P_2, who checks that for every i it holds that $\delta_i^0 \oplus \delta_i^1 = \Delta$.
3. P_2 decommits to W using the Decom procedure of $\mathsf{ExtractCom}$. Denote $W = W_1, \ldots, W_s$.
4. For $i = 1, \ldots, s$, party P_1 decommits to $c_i^{W_i}, d_i^{W_i}$ to P_2.
5. For $i = 1, \ldots, s$, P_2 verifies that $\mathsf{Decom}\left(c_i^{W_i}\right) \oplus \mathsf{Decom}\left(d_i^{W_i}\right) = \delta_i^{W_i}$, where Decom denotes the standard decommitment.
6. If all checks pass, then P_2 outputs Δ.

Proving consistency. Before proceeding to prove security, we explain how this functionality can be used to force P_1 to use the same input in two different garbled circuits. The functionality $\mathcal{F}_{\mathsf{Com}\Delta\mathsf{ZK}}$ is used for P_1 to commit to the *strings* of the signal bits associated with *all* P_1's input bits. (Recall that the signal bit determines whether the keys on the wire are given in the "correct" order or reversed order. In some works this value is also called the permutation bit.) In this warm-up

case with two commitments, P_1 uses $\mathcal{F}_{\mathsf{Com\Delta ZK}}$ to commit to the string of signal bits in the first garbled circuit and in the second garbled circuit (note that these are independent random strings, since each garbled circuit is independently generated). Now, P_1 provides (standard) commitments to the garbled values on these input wires (this is standard in all cut-and-choose protocols); we call them wire-commitments. However, P_1 provides the wire-commitments in the order determined by the signal bit (i.e., if the signal bit on a wire equals 0 then the commitment to the 0-key is placed before the commitment to the 1-key, and if the signal bit is 1 then they are reversed). Now, when two circuits are opened to be checked, then P_2 provides input $b = 1$ to $\mathcal{F}_{\mathsf{Com\Delta ZK}}$, and so all values are decommitted. This enables P_2 to check that the split commitment was constructed correctly and that the wire-commitments were indeed given in the order determined by the signal bits. In contrast, when two circuits are to be evaluated, then P_2 provides input $b = 0$ to $\mathcal{F}_{\mathsf{Com\Delta ZK}}$. As a result, P_2 will receive the string which is the XOR of the signal bits of all P_1's inputs in the two circuits. If the first bit in this XOR equals 0, then P_2 knows that for the wire associated with P_1's first input bit, P_1 must either decommit to the first wire-commitment in both circuits or decommit to the second wire-commitment in both circuits. (Since P_2 knows that the signal bit is the *same* in both cases – without knowing its value – this ensures that the same input bit is used by P_1 in both.) In contrast, if the XOR equals 1, then P_2 knows that P_1 must either decommit to the first wire-commitment in the first circuit and the second wire-commitment in the second circuit, or vice versa. (Once again, since P_2 knows that the signal bit is *different* in both cases, this ensures that the same input bit is used by P_1 in both.) The same holds for all of the wires associated with P_1's input (each bit of the XOR that is revealed is associated with a different wire associated with P_1's input).

3.1.1 Proof of Security

Let $x \in \{0,1\}^n$. Denote $c \in SC(x)$ if there exists randomness so that $c = SC(x)$; otherwise denote $c \notin SC(x)$. Denote $\mathcal{SC} = \{c \mid \exists x : c = SC(x)\}$; i.e., the set of all valid split commitments. We will be interested in commitments $c \in \mathcal{SC}$ versus series of $2s$ commitments that are not valid split commitments. Note that since commit is perfectly binding, the question of whether a series of $2s$ commitments are in $\mathcal{SC}$ is well defined.

We will prove that the proof in Protocol 3.2 (in the case of $b = 0$) is an interactive proof system for the *promise problem* (P, Q), where

$$P = \left\{(c, d, \Delta) \mid \exists x_1, x_2 \text{ s.t. } c \in SC(x_1) \wedge d \in SC(x_2)\right\},$$

and

$$Q = \left\{(c, d, \Delta) \mid \exists x_1, x_2 \text{ s.t. } c \in SC(x_1) \wedge d \in SC(x_2) \wedge x_1 \oplus x_2 = \Delta\right\}.$$

The promise problem (P, Q) considers the question of whether an input $(c, d, \Delta) \in Q$, under the promise that $(c, d, \Delta) \in P$. In words, we are given an input (c, d, Δ) and we are guaranteed that there exist x_1, x_2 such that $c \in SC(x_1)$ and $d \in SC(x_2)$. The "aim" is then just to determine if $x_1 \oplus x_2 = \Delta$ or $x_1 \oplus x_2 \neq \Delta$. (Note that if c and d are such that they are not valid commitments at all, then this will be detected in the checks carried out in the cut-and-choose protocol; i.e., when $b = 1$.)

We follow the definition of [13] regarding interactive proofs for promise problems. Informally, completeness must hold for every $(c, d, \Delta) \in P \cap Q$, soundness guarantees that the verifier will reject for any $(c, d, \Delta) \in P \setminus Q$, and nothing is required for $(c, d, \Delta) \notin P$.

In addition to the above, we prove that Protocol 3.2 securely computes the functionality $\mathcal{F}_{\mathsf{Com\Delta ZK}}$, defined in Figure 3.1, in the presence of a corrupt P_2. We stress that in the case that P_1 is corrupted we rely on the soundness property of the proof (since Protocol 3.2 does *not* securely compute $\mathcal{F}_{\mathsf{Com\Delta ZK}}$ in the presence of a corrupt P_1).

We prove the following:

Theorem 3.3 *If* commit *is a perfectly-binding commitment scheme and* ExtractCom *is a perfectly-hiding extractable commitment scheme, then the commitment phase of Protocol 3.2 is a perfectly-binding commitment scheme, and the proof phase is an interactive proof system for the promise problem (P, Q) defined above. In addition, Protocol 3.2 securely computes $\mathcal{F}_{\mathsf{Com\Delta ZK}}$ in the presence of a corrupt P_2.*

Proof: The fact that SC is a perfectly-binding commitment scheme follows immediately from the fact that commit is perfectly binding. We proceed to prove that the proof is an interactive proof system for the promise problem (P, Q). Completeness for the case that $(c, d, \Delta) \in P \cap Q$ is immediate. We next prove soundness. Let $(c, d, \Delta) \in P \setminus Q$. Then, there exist x_1, x_2 such that $c \in SC(x_1)$ and $d \in SC(x_2)$. However, $x_1 \oplus x_2 \neq \Delta$. We show that P_2 accepts the proof with probability at most $2^{-s} + \mu(n)$, for some negligible function $\mu(\cdot)$. Let $i \in [s]$. Since c and d are valid commitments, it holds that there exist r_i and ρ_i such that the ith element in c is

$$\mathsf{SCom}(x_1, r) = [c_i^0, c_i^1] = [\mathsf{commit}(x_1 \oplus r), \mathsf{commit}(r)]$$

and the ith element in d is

$$\mathsf{SCom}(x_2, \rho) = [d_i^0, d_i^1] = [\mathsf{commit}(x_2 \oplus \rho), \mathsf{commit}(\rho)].$$

Let (δ_i^0, δ_i^1) be the pair sent by P_1 in the proof. If $\delta_i^0 \oplus \delta_i^1 \neq \Delta$ then P_2 rejects the proof. Thus, it must be that $\delta_i^0 \oplus \delta_i^1 = \Delta$. Observe that if $\mathsf{Decom}(c_i^0) \oplus \mathsf{Decom}(d_i^0) = \delta_i^0$ *and* $\mathsf{Decom}(c_i^1) \oplus \mathsf{Decom}(d_i^1) = \delta_i^1$ then $x_1 \oplus r \oplus x_2 \oplus \rho = \delta_i^0$ and $r \oplus \rho = \delta_i^1$ and thus $\delta_i^0 \oplus \delta_i^1 = x_1 \oplus x_2 \neq \Delta$, in contradiction to the assumption above. Thus, we conclude that either $\mathsf{Decom}(c_i^0) \oplus \mathsf{Decom}(d_i^0) \neq \delta_i^0$ or $\mathsf{Decom}(c_i^1) \oplus \mathsf{Decom}(d_i^1) \neq \delta_i^1$. This holds for every $i = 1, \ldots, s$.

Now, let $W = W_1, \ldots, W_s$ be the random string chosen and committed to in the preprocess phase. Then, W is revealed only *after* $\{(\delta_i^0, \delta_i^1)\}_{i=1}^s$ was sent by P_1 to P_2. Furthermore, the commitment is perfectly hiding. Thus, the commitment to W received by P_1 in the preprocessing phase reveals no information about W, and so this is equivalent to W being chosen at random after the $\{(\delta_i^0, \delta_i^1)\}_{i=1}^s$ were sent. By what we have shown, for every i, it holds that either $\mathsf{Decom}(c_i^0) \oplus \mathsf{Decom}(d_i^0) \neq \delta_i^0$ or $\mathsf{Decom}(c_i^1) \oplus \mathsf{Decom}(d_i^1) \neq \delta_i^1$. Thus, for every $i = 1, \ldots, s$ we have with probablity at least $1/2$ (depending on where $W_i = 0$ or $W_i = 1$) party P_2 catches P_1. Thus, the probability that P_2 aborts is at least 2^{-s}.

Secure computation of $\mathcal{F}_{\mathsf{Com\Delta ZK}}$: It remains to prove that Protocol 3.2 securely computes $\mathcal{F}_{\mathsf{Com\Delta ZK}}$ in the presence of a corrupted P_2. Let $\mathcal{A}$ be an adversary who has corrupted P_2. We begin by defining the simulator $\mathcal{S}$:

1. $\mathcal{S}$ interacts with $\mathcal{A}$ in the "Commit to Challenge" phase and obtains the trapdoor td that enables it to extract any committed value in the extractable commitment; see Section 2.1.

2. Using the trapdoor td, $\mathcal{S}$ extracts b and W.

3. $\mathcal{S}$ sends b to $\mathcal{F}_{\mathsf{Com}\Delta\mathsf{ZK}}$ (on P_2's behalf).

4. If $b = 0$,

   - Upon receiving Δ from $\mathcal{F}_{\mathsf{Com}\Delta\mathsf{ZK}}$, for $k = 1, 2$ the simulator $\mathcal{S}$ chooses $2s$ independent random strings $r^0_{k,1}, r^1_{k,1}, \ldots, r^0_{k,s}, r^1_{k,s}$, computes the commitments $\{(c^0_{k,i} = \mathsf{commit}(r^0_{k,i}), c^1_{k,i} = \mathsf{commit}(r^1_{k,i}))\}_{i=1}^{s}$ and defines $c_k = \langle c^0_{k,1}, c^1_{k,1}, \ldots, c^0_{k,s}, c^1_{k,s}\rangle$. $\mathcal{S}$ hands $\mathcal{A}$ the message $(\mathsf{commit}, c_1, c_2)$.
   - For every $i = 1, \ldots, s$, $\mathcal{S}$ chooses $\delta^{W_i}_{k,i} = r^{W_i}_{k,i} \oplus r^{W_i}_{k+1,i}$ and sets $\delta^{1-w_i}_{k,i} = \Delta \oplus \delta^{W_i}_{k,i}$. $\mathcal{S}$ hands $\{(\delta^0_{k,i}, \delta^1_{k,i})\}_{i=1}^{s}$ to $\mathcal{A}$.
   - $\mathcal{S}$ receives the decommitment of W from $\mathcal{A}$. If the decommitment is not valid, then $\mathcal{S}$ simulates P_1 halting, outputs whatever $\mathcal{A}$ outputs, and halts. If the decommitment is to a string $W' \neq W$, then $\mathcal{S}$ outputs fail. Otherwise, $\mathcal{S}$ proceeds to the next step. Similar steps are executed with respect to the bit b.
   - For every $i = 1, \ldots, s$, simulator $\mathcal{S}$ sends $\mathsf{Decom}(c^{W_i}_{k,i}), \mathsf{Decom}(d^{W_i}_{k,i})$; it can do this since it generated these commitments.

5. If $b = 1$, upon receiving x_1, x_2 from $\mathcal{F}_{\mathsf{Com}\Delta\mathsf{ZK}}$, the simulator $\mathcal{S}$ emulates an honest P_1 with inputs x_1, x_2. If $\mathcal{A}$ decommits to $b' \neq b$, $\mathcal{S}$ outputs fail.

6. $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs and halts.

We now show that the view of $\mathcal{A}$ in the simulation with $\mathcal{S}$ is indistinguishable from its view in a real execution. First, note that $\mathcal{S}$ outputs fail with negligible probability since the commitments to b and W are *computationally binding*. In order to prove the simulation, we first consider a hybrid simulator $\mathcal{S}_1$ who works in an ideal model in which it always receives x_1, x_2 from the trusted party. $\mathcal{S}_1$ works in exactly the same way as $\mathcal{S}$ regarding the proof, but defines the commitment c by choosing $r^{W_1}_1, \ldots, r^{W_s}_s$ at random and setting $r^{1-W_i}_i = r^{W_i}_i \oplus x_k$, for every $i = 1, \ldots, s$. Thus, unlike in the simulation with $\mathcal{S}$, here we have that $c \in SC(x)$. In the proof, $\mathcal{S}_1$ works exactly like $\mathcal{S}$.

We claim that the view of $\mathcal{A}$ when interacting with $\mathcal{S}_1$ is computationally indistinguishable from its view when interacting with $\mathcal{S}$. This follows from a straightforward reduction to the hiding property of commit. Briefly, consider a commitment experiment where a bit $b \in_R \{0, 1\}$ is chosen at the onset. Then, the adversary interacts with a commitment oracle that receives pairs of length-n values, and returns a commitment to the first value if $b = 0$, and a commitmentent to the second value if $b = 1$. (This is an LR-oracle type formulation [4] and can be shown to be equivalent to standard computational hiding of commitments in a straightforward way.) The commitment adversary runs $\mathcal{S}/\mathcal{S}_1$ and the trusted party (together with the commitment values). Upon receiving x_1, x_2, it chooses $r^0_1, r^1_i \ldots, r^0_s, r^1_s$ at random. Then, it computes the commitments $c^{W_i}_i = \mathsf{commit}\left(r^{W_i}_i\right)$ for every $i = 1, \ldots, s$. In addition, it queries its commitment LR-oracle with the pair $\left(r^{1-W_i}_i, r^{W_i}_i \oplus x_k\right)$ and defines $c^{1-W_i}_i$ to be the response from the oracle. The commitment adversary proceeds exactly like $\mathcal{S}/\mathcal{S}_1$ for the remainder of the simulation. Observe that it can carry out the proof since it needs to decrypt only the commitments $c^{W_i}_i$ and $d^{W_i}_i$, and it computed these itself. This completes the proof. ■

3.1.2 Replacing the Perfectly-Binding Commitment

Note that so far we have assumed that $\mathsf{commit}(\cdot)$ is a perfectly-binding commitment. In practice, perfectly-binding commitments are less efficient than computationally binding ones. For example, with an appropriate assumption on the cryptographic hash function, $\mathsf{commit}(x) = \mathsf{H}(x; r)$ is a computationally binding and computationally hiding commitment. If we model H as a random oracle, then commit is still only computationally binding. However, it is extractable, and thus we can prove the interactive proof of Protocol 3.2 to be a proof of knowledge. This achieves the same effect as soundness. (Note that once we model H as a random oracle, we can also use commit as the statistically-hiding extractable commitment $\mathsf{ExtractCom}$.)

In order to use any computationally-binding commitments, including like that above but without resorting to the random oracle model, the following change can be made to Protocol 3.2. Let σ be a seed to a pseudorandom generator G, and define $SC(x)$ as before (i.e., $SC(x) = \langle \mathsf{SCom}(x, r_1), \ldots, \mathsf{SCom}(x, r_s) \rangle$) where the underlying commitment uses H, but all of the randomness in generating SC is taken from $G(\sigma)$, and a perfectly-binding extractable commitment is given to σ alone. This has the advantage that a single perfectly-binding commitment to a short string suffices to define all of SC as perfectly binding. The promise problem used to model the interactive proof, and the proof of soundness then remains the same (with the additional requirement that P_1 is polynomial time and cannot efficiently open any of the individual commitments to anything else). This adds one extractable commitment per circuit (which can be implemented via El Gamal and so costs 2 exponentiations per circuit), plus a single zero-knowledge proof of knowledge of the El Gamal private key generated by P_1 (that is done only once for all circuits and costs just 9 exponentiations).

3.2 Extending to Many Messages

The functionality of Figure 3.1 works with only two messages from P_1 (and so only for two circuits). We would like to use it for a larger number of messages, where P_2 can choose any subset of them to be revealed and learn the XOR differences between the remaining ones (as in the cut-and-choose case where a random subset of the circuits are evaluated and consistency must be proved for them). In addition, for our online/offline 2PC protocol we would like P_2 to be able to pick different subsets of the unrevealed messages, and learn the XOR differences for all the messages in each subset (since in the online/offline setting, the evaluated circuits are randomly thrown into buckets and each bucket is used for a different execution; thus the XOR differences are needed inside each bucket).

FIGURE 3.4 (The Extended Commit-and-Difference Proof Functionality $\mathcal{F}_{\mathsf{ExComΔZK}}$)

$\mathcal{F}_{\mathsf{ExComΔZK}}$ runs with parties P_1 and P_2, a public index M (saying how many inputs there are), a public constant N (saying how many subsets there are), and a public constant B (saying how big each subset is), as follows:

Input: $\mathcal{F}_{\mathsf{ExComΔZK}}$ receives M messages $(m_1, \ldots, m_M)$ from P_1, and a series of subsets $I_1, \ldots, I_N \subset [M]$ from P_2.

Output: For $j \in [N]$, let $I_j = \{i_j^1, \ldots, i_j^B\}$ and let Δ_j be the set $\left\{ m_{i_j^k} \oplus m_{i_j^{k+1}} \right\}_{k=1}^{B-1}$.

- $\mathcal{F}_{\mathsf{ExComΔZK}}$ sends $\Delta_1, \ldots, \Delta_N$ to P_2. In addition, $\mathcal{F}_{\mathsf{ExComΔZK}}$ sends P_2 the value m_i, for every $i \notin \left(\bigcup_{j=1}^{N} I_j \right)$.
- $\mathcal{F}_{\mathsf{ExComΔZK}}$ sends $I_1, \ldots, I_N$ to P_1.

The extended functionality is defined in Figure 3.4. The subsets $I_1, \ldots, I_N$ are the buckets of circuits to be evaluated in the online phase (each bucket is of size B). Thus, P_2 learns the XOR differences between every pair in each bucket; this enables it to verify consistency as described above. Observe that the indices of values not in any of $I_1, \ldots, I_N$ are circuits that are checked; thus, the values corresponding with these indices are revealed.

FIGURE 3.5 (Extended Commit with Proof of Difference – for a Single Subset I)

Commit to Indices and Challenge:

1. P_1 and P_2 run a *perfectly-hiding (or statistically-hiding) extractable commitment* scheme $\mathsf{ExtractCom}$, in which P_2 commits to a key K for a PRG and to an s-bit random string W. (Pedersen commit with a ZK proof of knowledge of the committed value suffices.)
2. P_2 sends P_1 an encryption c of the indices $i_1, \ldots, i_B$ of the single subset I by XORing them with the output of the PRF with key K.

Commit to $m_1, \ldots, m_M$: For every m_k $(k = 1, \ldots, M)$:

1. Define the split commitment $\mathsf{SCom}(m_k, r) = [\mathsf{commit}(m_k \oplus r), \mathsf{commit}(r)]$
2. For $i = 1, \ldots, s$, P_1 chooses $r_i \leftarrow \{0,1\}^n$ and computes $[c_i^0, c_i^1] = \mathsf{SCom}(m_k, r_i)$
3. Denote the commitment to m_k by $c_k = SC(m_k) = \langle \mathsf{SCom}(m_k, r_1), \ldots, \mathsf{SCom}(m_k, r_s) \rangle$.

P_1 sends $(\mathsf{commit}, c_1, \ldots, c_M)$ to P_2.

Decommit to m_k:

1. P_1 sends $(m_k, r_1, \ldots, r_s, R_1^0, R_1^1, \ldots, R_s^0, R_s^1)$, where R_i^b is the randomness used to generate the commitment c_i^b in $SC(m_k)$.
2. Upon receiving the above, P_2 verifies that $c_k = SC(m_k)$ is correctly constructed.

Proofs of Difference:

1. P_2 decommits to K, and P_1 decrypts c obtaining $i_1, \ldots, i_B$.
2. For $k = 1, \ldots, B-1$:
   (a) Let $r_1, \ldots, r_s$ be s.t. $SC(m_k) = \langle \mathsf{SCom}(m_1, r_1), \ldots, \mathsf{SCom}(m_1, r_s) \rangle$, and let $\rho_1, \ldots, \rho_s$ be s.t. $SC(m_{k+1}) = \langle \mathsf{SCom}(m_k, \rho_1), \ldots, \mathsf{SCom}(m_k, \rho_s) \rangle$. Let $\Delta_k^I = m_k \oplus m_{k+1}$.
   (b) For every $i = 1, \ldots, s$, party P_1 defines $\delta_i^0 = m_k \oplus r_i \oplus m_{k+1} \oplus \rho_i$ and $\delta_i^1 = r_i \oplus \rho_i$ (note that $\delta_i^0 \oplus \delta_i^1 = m_k \oplus m_{k+1} = \Delta_k^I$)
   (c) P_1 sends $\{(\delta_i^0, \delta_i^1)\}_{i=1}^s$ and Δ_k^I to P_2, who checks that for every i, $\delta_i^0 \oplus \delta_i^1 = \Delta_k^I$.
3. P_2 decommits to W.
4. For $k = 1, \ldots, B-1$:
   (a) For $i = 1, \ldots, s$, party P_1 sends $\mathsf{Decom}(c_i^{W_i}), \mathsf{Decom}(d_i^{W_i})$ to P_2, where $[c_i^0, c_i^1]$ is the ith commitment pair in $SC(x)$, and $[d_i^0, d_i^1]$ is the ith commitment pair in $SC(y)$,
   (b) For $i = 1, \ldots, s$, party P_2 verifies that $\mathsf{Decom}(c_i^{W_i}) \oplus \mathsf{Decom}(d_i^{W_i}) = \delta_i^{W_i}$.

The main difference between the protocol that securely computes the extended functionality in Figure 3.4 and the protocol in Figure 3.2 is that in the general case, P_2 commits to all of the subsets $I_1, \ldots, I_N$ initially (and not just a single bit b). The detailed protocol appears in Figure 3.5, described for a single subset I input by P_2. Extending the protocol in Figure 3.5 to support multiple sets is straightforward; P_2 just sends a longer list of indices (instead of only B), ordered by the buckets they belong to. The proof of security follows directly from the proof of Theorem 3.3.

3.3 Using $\mathcal{F}_{\mathsf{ExComΔZK}}$ in Cut-and-Choose

As we have mentioned, for every circuit in the cut-and-choose, P_1 commits to the string m which contains the "signal" bits σ on *all* of its input wires (this requires $2s$ basic commitments commit). In addition, the input garbled labels are committed; if $\sigma = 0$ then the commitments are in the correct order (with the 0 label first), and if $\sigma = 1$ they are in the opposite order (with the 1 label first). When checking a circuit, these commitments are also verified, including their order according to the signal bits as committed in $\mathcal{F}_{\mathsf{ExComΔZK}}$. For the evaluation circuits, let $gc_1, \ldots, gc_B$ be the circuits to be evaluated, and let $m_1, \ldots, m_B$ be their committed signal bit labels. Then, for every $i = 1, \ldots, B$, party P_1 sends P_2 the string $\hat{x}_i = m_i \oplus x$, where x is its input to the secure computation, along with the decommitment to the labels pointed to by $\hat{x}_i$ (the 0-label if the bit of $\hat{x}_i = 0$, and the 1-label otherwise). In addition, for every $i = 1, \ldots, B-1$, it defines $\Delta_i = \hat{x}_i \oplus \hat{x}_{i+1}$ and proves that $m_i \oplus m_{i+1} = \Delta_i$ (using $\mathcal{F}_{\mathsf{ExComΔZK}}$). Party P_2 uses the labels decommitted, and this ensures that the same input is used in all circuits in the bucket. This is because the functionality provides the XOR differences $\Delta_1 = m_1 \oplus m_2, \ldots, \Delta_{B-1} = m_{B-1} \oplus m_B$, and these differences are preserved even after XORing with the input. This means that the *same* x was XORed with $m_1, \ldots, m_B$ and so the same value is used throughout. The overall cost is $2s$ basic commitments commit per circuit plus two extractable commitments, which is very cheap (especially since the number of circuits per bucket is very small).

Advantages Over Previous Input Consistency Proofs. Note that the number of commitments in Protocol 3.2 is only $2s$ for every circuit. In the online/offline setting, the number of circuits is very small (typically 5-10, depending on the parameters) and thus this costs significantly less than a single commitment per input bit (unless the input is tiny). When commit is implemented as described above using a computationally-binding commitment, the resulting protocol is more efficient than those of [28, 34], and significantly more simple to understand and implement. From a theoretical standpoint, our protocol can also be based on very standard assumptions (though with the additional negligible overhead of the two exponentiations needed by the El Gamal encryption used to implement a perfectly-binding commitment), whereas [28] requires correlation robustness and [34] requires Free-XOR. Our protocol yields significant improvements over [17, 26] who both use inefficient solutions for the input-consistency issue (i.e., using discrete-log ZK proofs).

4 Optimized 2PC in the Online/Offline Setting

We base our protocol on [26] who present a protocol for multiple 2PCs in the online/offline model. First, we adapt it to use the technique of [23, 28, 34] for protecting against selective-OT attacks, instead of using cut-and-choose OT which is much less efficient. Next, we plug in our new technique for checking P_1's input consistency as discussed in Section 3.3. These two modifications essentially replace all the exponentiations required by the protocol of [26] for the input wires with cheaper cryptographic operations (and a small number of exponentiations that is independent of the input size).

Since our goal is to minimize the cost of the online stage, we chose to work in the random-oracle model, so we could construct adaptively secure garbled circuits in an efficient way. We therefore further utilize the power of the random-oracle model and optimize other parts of the protocol. For example, we replace all extractable commitments (that require exponentiations) with random-oracle

based commitments (that requires only hash function calls), and reduce the number of encryptions needed for the cheating recovery method presented in [26].

The protocol is described in Appendix B in Figures B.1 and B.4 and uses the sub-protocols of Figure B.2. Based on the proof of security from [26], we prove the following:[1]

Theorem 4.1 *Let* commit *and* ExtractCom *be implemented using a random oracle* H *(defined by* $\mathsf{commit}(m,r) = \mathsf{ExtractCom}(m,r) = \mathsf{H}(m;r)$*), let* PRF *be a pseudorandom function, and let the garbling scheme be adaptively secure. Then, for any function* f*, Protocols B.1-B.4 securely compute* f *with multiple executions (according to the definition of [26]).*

Proof Sketch: We follow the proof of [26] and present a proof sketch of Theorem 4.1. For each corruption case we provide some intuition about the simulation and then show indistinguishability of the real and the simulated executions via a sequence of hybrid executions.

P_1 is corrupted. The simulator emulates an honest P_2 with the following modifications. In the offline stage, it first extracts the seeds from and λ-s from $\mathsf{ExtractCom}(seed_1), \ldots, \mathsf{ExtractCom}(seed_M)$ and $\mathsf{ExtractCom}(\lambda_1), \ldots, \mathsf{ExtractCom}(\lambda_M)$. Second, it sees Adv′ messages to $\mathcal{F}_{\mathsf{Com}\Delta\mathsf{ZK}}$. At the end of the offline stage, the simulator can determine if there exists a bucket in which none (or most) of the bundles is good (e.g., the garbled circuit is invalid). If so, the simulator aborts. (Note that this event could happen with only a negligible probability because of the cut-and-choose protocol.)

In the online stage, the simulator emulates an honest P_1 with a random input y and learns Adv's input to the good bundles. (Recall that the simulator saw Adv's messages to $\mathcal{F}_{\mathsf{Com}\Delta\mathsf{ZK}}$, thus it knows the signal bits of the good bundles.) It sends this input to the ideal functionality as P_1's input. It emulates the protocol until its end (emulating abort if needed) and outputs whatever Adv outputs.

Before we describe in more details the hybrids, we reduce the security of the protocol, denoted by Π, to the security of protocol Π' in which P_2 generates the random mapping functions with truly random coins (instead of pseudo-random ones) and commits to the functions instead of the PRF seeds. Let Π'' be a protocol in which P_2 uses a PRF to derive the coins needed for generating the random mapping functions, but then commits to the functions instead of the PRF seeds. Since those commitments are done using ExtractCom, any adversary that can cheat in Π can also cheat in Π''. On the other hand, any adversary that can cheat in in Π'' can also cheat in Π' or else we get a distinguisher for the PRF security game. Therefore, any adversary in Π can also cheat in Π' except for a negligible probability.

We proceed to proving that Π' is secure. We show indistinguishability of the real execution and the simulated one using the following hybrid executions:

- The real execution.
- The simulator Sim emulates an honest P_2 with the real inputs and realizes the ideal functionalities for Adv. It see all Adv's messages to the OTs, $\mathcal{F}_{\mathsf{Com}\Delta\mathsf{ZK}}$, and the random-oracle. Indistinguishability follows the composition theorem.

[1] We are aware that the proof is far from satisfactory, even as a proof sketch, and that the protocol specification needs to be significantly improved. We hope to carry out this work in the near future.

- Sim aborts if at the end of the offline stage there exists a bucket that is fully-cracked with respect to C or majority-cracked with respect to C'. (This abort is in addition to the abort that an honest P_2 might do in case it detects cheating.) Following Lemmas 2.3 and 2.4, this abort happens with a negligible probability.

- Sim emulates P_2 with a random input in the online stage, learns Adv's input to the good bundles, and sends it to the ideal functionality. (Observe that there exists exactly one input in all good bundles.) It sends abort if the emulated P_2 aborts, and otherwise outputs whatever Adv outputs. Note that this execution differ only in P_2's input. However, since P_2's input is encoded using an s-resistance matrix, Adv has no information about the value $y^{(1)}$ that P_2 has used in the offline stage, so the masking of y with $y^{(1)}$ completely hides y (except for the negligible probability for which Adv has learned s input bits of $y^{(1)}$ using selective-OT attack). Additionally, for the good bundles, P_2 evaluations with the real input should return the same output as the ideal functionality, thus, the output of P_2 in this execution would be the same as in the previous one.

The last hybrid execution is essentially the simulated execution since the simulator does not use the real input y, so in conclusion, the real and the simulated executions are indistinguishable.

P_2 is corrupted. In the offline stage, the simulator emulates an honest P_1 and extracts the values of σ and $seed_\pi$ of the cut-and-choose for C and for C' (by utilizing the extractability of ExtractCom/random-oracle). It constructs good garbled-circuit bundles for the ones that are chosen to be checked, but uses fake ones for the bundles that are chosen to be evaluated. In addition, the simulator sees Adv's inputs to the OTs, and by that, learns the values of $y_1^{(1)}, \ldots, y_N^{(1)}$, and $d_1^{(1)}, \ldots, d_N^{(1)}$

In the online stage, for the current bucket, the simulator gets $y^{(2)}$ and uses that to recover y. It sends it to the ideal functionality and receives the output z. Then, it causes the garbled circuits to always output z (using the simulator of the adaptive garbled circuit). It continues the execution with a random input x and outputs what Adv outputs.

We show indistinguishability of the real execution and the simulated one using the following hybrid executions:

- The real execution.

- The simulator Sim emulates an honest P_1 with the real inputs and realizes the ideal functionalities for Adv. It see all Adv's messages to the OTs, $\mathcal{F}_{\mathsf{Com}\Delta\mathsf{ZK}}$, and the random-oracle. Indistinguishability follows the composition theorem.

- Once Adv sends $y^{(2)}$ in the online stage, Sim computes y, sends it to the ideal functionality and receives the output z. Then, Sim programs the commitments of the input wire label that Adv has not learnt so that they will be commitments to zeros, and in addition, it sets the garbled circuits to be fake garbled circuits that always output z. Because of the adaptive security of the garbling scheme, and since Adv might catch the modification of the random-oracle with only a negligible probability, this execution is indistinguishable from the previous one.

- Sim aborts if Adv sends $d^{(2)}$ such that $d^{(2)} = D \oplus d^{(1)}E'$ (meaning it knows D). Note that at this stage, the only information about D comes from $\mathsf{H}(D)$, so if Adv knows it, we can use it

to break the one-wayness of H (i.e., we receive $\mathsf{H}(D)$ in the one-wayness game, and use Adv to recover it.) Therefore, this abort happens with only a negligible probability.

- Sim modifies the garbled circuits of the cheating recovery circuits, causing them to always output 0. Since at this stage, Adv does not input the correct value D to the circuit, it should always learn 0, and thus this execution is indistinguishable from the previous one following the security of the garbling scheme.

- Sim uses the a random input on P_1's behalf. Note that Adv knows only the XORs of the signal bits, but does not know their actual values for the evaluated circuits. Therefore, x_1 is distributed exactly the same as in the previous execution. (Since we use the random-oracle for the commitments of P_1's input wire labels, we can program them so that their decommitments are distributed the same as well.)

The last hybrid execution is essentially the simulated execution since the simulator does not use the real input y, so in conclusion, the real and the simulated executions are indistinguishable. ■

We remark that the protocol can be slightly modified to be secure given an adaptively secure garbling scheme and without utilizing the random oracle for commit and $\mathsf{ExtractCom}$. The only modification needed is to commit ro all the commitments of the input labels using a trapdoor (equivocal) commitment, so that in the simulation in case P_1 is corrupted, the simulator can change the commitments on the input labels after it learns P_1's input. The cost of this modification is small, requiring only one additional trapdoor commitment per garbled circuit.

5 Prototype Implementation

Our goal is to provide an end-to-end system for multiple 2PC executions in the online/offline setting. First, the system includes a tool for providing the user with very good sets of parameters (in the offline/online setting, finding the parameters of how many circuits to use and how many to open for a given security parameter, is complex). Next, the system is optimized both crypto-wise (e.g., using the random-oracle where suitable) and engineering-wise (e.g., using parallelism where possible). Some key parts of the system are the following:

Additional optimizations in the random-oracle model. First, recall that everywhere we use an extractable commitment, we actually use $\mathsf{commit}(x;r) = \mathsf{H}(x;r)$. Second, since the labels of P_2's input wires are random, we can use a second random-oracle H_2, and ask P_1 to commit to label W by just sending $\mathsf{H}_2(W)$. This reduces the length of P_1's inputs to the OTs by a factor of two (since the decommitment includes only W and not additional randomnes), thereby reducing bandwidth. This still preserves security as $\mathsf{H}_2(W)$ does not reveal any information about W if W has enough entropy (which happens in our case, as W is at least 128 bits long random string).

Finding good parameters. We implemented a program that is given the values of s (statistical security parameter) and N (the overall number of executions desired), and calculates the parameters based on Lemmas 2.3 and 2.4 that minimize the overall number of circuits (to minimize the run-time of the offline stage) or the number of evaluation circuits per bucket (to minimize the run-time of the online stage). Similarly to what was observed by [26], the parameters we obtain by directly computing the equations found in Lemmas 2.3 and 2.4 are much better than the (clean) upper

bounds proven in [26]. See Tables 1 and 2 for several example sets of generated parameters. We also implemented a program that receives a circuit C and calculates the encoding matrices E and E', used for protecting P_2's input from selective-OT attacks (as described in Section 2.3).

N	Total number of circuits overall	Number of eval circuits per 2PC (online phase)	Number of circuits per 2PC
8	136	10	16.95
	165	8	20.51
32	362	7	11.29
	437	6	13.63
128	998	6	7.79
	1143	5	8.92
1024	5627	5	5.49
	5689	4	5.55
4096	18005	4	4.39
	25600	3	6.25

Table 1: Several sets of parameters for Lemma 2.3 with $s = 40$. Note the tradeoff between the total number of circuits (which affects the offline stage efficiency) and the number of evaluation circuits per bucket (which affects the online stage efficiency).

N	Total number of circuits overall	Number of eval circuits per 2PC (online phase)	Number of circuits per 2PC
8	277	19	34.54
	296	17	36.95
32	706	15	22.05
	771	13	24.07
128	1995	12	15.58
	2246	10	17.54
1024	10843	9	10.58
4096	36294	7	8.86

Table 2: Several sets of parameters for Lemma 2.3 with $s = 80$.

In contrast to [26], we have set the statistical security parameter s to be such that the probability that an adversary cheats in a *single* 2PC execution is 2^{-s}. (In [26], they set 2^{-s} to be the probability that an adversary cheats in at least one of the many executions overall). Indeed, this is merely a different way of looking at the parameters, but we believe that for most users, considering security of a single execution is more natural.

Handling large inputs. Calculating a probe-resistant matrix according to the algorithm of [34] is a very computationally intensive task when the input is large (e.g., 1000-bit long). Instead, when dealing with long inputs, our system constructs the probe-resistant matrix using a composition of smaller probe-resistant matrices that can each be generated very efficiently. While this method

results in a slightly larger matrix used in the protocol (and, thus, more OTs), it dramatically reduces the time needed for generating the probe-resistant matrix (from hours to seconds).

Architecture. We use the SCAPI library [10, 1] for implementing the high-level steps of the protocols, while using more optimized C/C++ code for steps that are more computationally intensive (e.g., computing the large amount of XORs of the probe-resistant matrix). We use the OT-extension implementation of [3], a new SCAPI garbling library that uses fixed-key AES for garbling, as suggested by [5], and the SCAPI wrapper of OpenSSL for AES and SHA-1.

The prototype is able to generate and evaluate many garbled circuits (and carry out other operations) in parallel, using multiple threads. In addition, before the online stage begins, all relevant files are loaded to memory so that no I/O delays occur once the interaction starts. (We do not include disk I/O time in our measurements as in practice loading to memory should always occur before actual inputs are received.)

The code for the prototype is freely available from the SCAPI project git [1].

6 Performance Evaluation

Setup. We ran the prototype implementation on two types of Amazon AWS instances: c4.8xlarge (with 64GB RAM and 36 virtual 2.9GHz CPUs) and c4.2xlarge (with 15GB RAM and 8 virtual 2.9GHz CPUs). On both instances, garbling 1000 AES circuits in isolation took about 470 ms. Unless stated otherwise, all the tests in this section were ran on the c4.8xlarge instances. We ran tests with *LAN configuration*, where both parties were in the same AWS region and the roundtrip is less than 1 ms, and with a *WAN configuration*, where the parties were in different regions (specifically, EU-west and US-east) and the roundtrip is 75 ms.

Test functions. We tested the prototype with the following circuits:

1. ADD: receives two 32-bit integers and outputs their sum (the circuit has 127 AND gates)
2. AES: receives two 128-bit inputs and outputs the encryption of the first input using the second input as the key (the circuit has 6800 AND gates)
3. SHA-1: receives two 256-bit inputs and outputs the SHA-1 hash digest of the XOR of the two inputs (the circuit has 37300 AND gates)
4. SHA-256: receives two 256-bit inputs and outputs the SHA-256 hash digest of the XOR of the two inputs (the circuit has 90825 AND gates)

We remark that smaller circuits exist for some of these tasks. However, our goal was not the computation of these functions per se, but rather measurements over different sized circuits.

Results. In the following, all experiments use the sets of parameters from Tables 1 and 2, and unless said otherwise, $s = 40$. See Table 3 for the results of the implementation on these circuits; the online time given is the average over all executions. We can see that, for example, the *total amortized time* it takes to evaluate a single AES (i.e.. the sum of the online and offline stages timings) ranges from around 210ms (for $N = 32$) to around 80ms (for $N = 1024$). See Table 4 for results with $s = 80$; the cost with $s = 80$ is 2-3 times the cost of $s = 40$ and so is also very fast.

Circuit	Number of buckets	Offline total	Offline per bucket	Online time per bucket		
				1 thread	5 threads	9 threads
ADD	32	4266	133	10	6	8
	128	9735	76	7	5	4
	1024	49590	48	5	4	4
AES	32	6310	197	18	13	12
	128	14539	114	13	10	10
	1024	75879	74	9	7	7
SHA-1	32	10042	314	40	29	26
	128	24201	189	31	24	22
	1024	127555	125	20	15	15
SHA-256	32	14699	459	75	62	50
	128	35243	275	62	50	40
	1024	210935	206	44	33	33

Table 3: Running times of the different circuits in LAN configuration (in ms). For $N = 32$ we use buckets of 7 circuits of C and 20 of C'; for $N = 128$ we use buckets of 6 circuits of C and 14 of C'; for $N = 1024$ we use buckets of 4 circuits of C and 10 of C' (C is the main circuit and C' is the auxiliary cheating-recovery circuit). Offline times are for execution with 9 threads.

Circuit	Number of buckets	Offline total	Offline per bucket	Online time per bucket		
				1 thread	5 threads	9 threads
AES	32	13901	434	52	29	30
	128	35031	274	34	17	20
	1024	164937	161	24	13	14
SHA-256	32	29041	908	176	123	129
	128	74120	579	129	87	96
	1024	662640	647	100	79	76

Table 4: Running times for AES circuit in LAN configuration for $s = 80$. For $N = 32$ we use buckets of 15 circuits of C and 46 of C'; for $N = 128$ we use buckets of 12 circuits of C and 28 of C'; for $N = 1024$ we use buckets of 9 circuits of C and 20 of C'.

Number of buckets	Offline total		
	1 thread	5 threads	9 threads
32	10129	6905	6310
128	23961	15819	14539
1024	125993	82476	75879

Table 5: Running times of the offline stage for the AES circuit in LAN configuration (in ms) for $s = 40$. (For $s = 80$, the times are 2-2.5 larger.) The number of circuits per bucket is as in Table 3.

In Table 5 we show an example of the effect of the number of threads on the *offline* stage performance. Even though performance is far from linear in the number of threads, it is clear that parallelism helps, and we expect that further optimizations utilizing multithreading will further improve performance. We also ran these experiments for other settings and verified that this effect is consistent.

As discussed earlier, there is a tradeoff between the total number of circuits and the number of evaluation circuits per online stage. This affects the performance of the two stages. See Tables 1 and 2 for examples of those tradeoffs. In addition to the tests described in Table 3, we also tested how this tradeoff is reflected in practice: Instead of running AES with $s = 80$, $N = 128$ and bucket size 12, we ran it with bucket size 10 which increases the total number of circuits from 1995 to 2246. The result was that the offline running time was 310 ms per 2PC (with 9 threads) and the online running time was 31/16/17 ms for 1/5/9 threads (respectively). This is about 13% slower in the offline phase and around 10% faster in the online phase, which roughly matches the differences in the numbers of circuits and so is as expected. Thus, it is possible to obtain different tradeoffs, depending on whether it is more important to reduce the overall cost or the online latency.

We also tested the prototype in the WAN configuration, since in many real-world scenarios the participating parties may be far apart. Note that in these scenarios, the Yao-based approach has a significant advantage over TinyOT [29] and SPDZ [7] who have a number of rounds that depends on the circuit depth. See Tables 6, 7 and 8 for the results of those tests.

Circuit	Number of buckets	Offline total	Offline per bucket	Online time per bucket		
				1 thread	5 threads	9 threads
AES	32	36039	1126	171	164	163
	128	117650	919	166	163	164
	1024	778235	759	162	160	160
SHA-1	32	52463	1639	194	185	176
	128	152509	1191	194	182	180
	1024	2936775	2867	184	173	175

Table 6: Running times of AES and SHA-1 in WAN configuration using the parameters of Table 3 (in ms) for $s = 40$. The roundtrip between the parties was 75 ms.

Circuit	Number of buckets	Offline total	Offline per bucket	Online time per bucket		
				1 thread	5 threads	9 threads
AES	32	54467	1702	204	180	180
	128	165239	1291	190	172	176
	1024	1102191	1076	178	167	169
SHA-1	32	89860	2808	268	227	234
	128	314239	2455	236	210	211
	1024	1412681	1380	213	196	196

Table 7: Running times of AES and SHA-1 in WAN configuration using the parameters of Table 4 (in ms) for $s = 80$. The roundtrip between the parties was 75 ms.

Number of buckets	Offline total		
	1 thread	5 threads	9 threads
32	36483	35725	36039
128	113352	111586	117650
1024	815313	778010	778235

Table 8: Running times of the offline stage for the AES circuit in WAN configuration (in ms) for $s = 40$. (For $s = 80$, the costs at most twice larger.)

We note that our online phase requires four rounds of interaction (two messages in each direction), and since the roundtrip in our WAN configuration is 75 ms, the cost of our online stage cannot go below 150 ms. Our tests show that in this case, the majority of the time spent is on communication, and the cost of the actual steps of our protocol (i.e., excluding communication) is very low. We remark that protocols which require a round of communication for every level of the circuit in the online phase will perform poorly in this scenario. For example, the best AES circuit has depth 50 and thus the online time will not be able to be less than 3750 ms in this setting, even if each step in the protocol has almost zero cost and only a few bits are sent.

Last, we tested the prototype on a weaker AWS instance, i.e., c4.2xlarge, for computation of AES. See Table 9. As expected, performance is mostly worse than on the stronger instance but for some parameters they are still very close. This is mainly because of memory issues, and thus is mostly reflected in the offline stage (our current implementation of the offline stage stores many garbled circuits in memory). Indeed, the online times are almost the same.

Number of buckets	Offline total	Offline per bucket	Online time per bucket		
			1 thread	5 threads	9 threads
32	6915	216	17	12	12
128	18367	143	12	10	10
1024	93613	91	8	6	6

Table 9: Running times of AES (in ms) in LAN configuration using the parameters of Table 3 on c4.2xlarge instances. (The costs for $s = 80$ are about 2-3 times larger.)

Communication in the offline stage consists mostly of the garbled circuits, whereas the communication in the online stage is very small. For example, for AES computation with $s = 40$ and $N = 32$, about 260MB are transmitted in the offline stage and only about 312KB per online execution; for $N = 128$, about 698MB are transmitted in the offline stage and only about 238 KB per online execution; for $N = 1024$, about 3850MB are transmitted in the offline stage and less than 170KB in the online stage. (Recall that the number of circuits per bucket is larger for $N = 32$ than for $N = 128, 1024$, and thus the communication in the online stage is larger.) For $s = 80$, these numbers are about double, as expected since the number of circuits is about double.

Comparison with related work. We focus here on comparing our results with the ones reported by previous works. We leave the comprehensive benchmarking of all relevant protocols using similar hardware, network configuration, and so on to future work.

As discussed in Section 1, the fastest previously published implementation of cut-and-choose based 2PC on standard machines (without massive parallelism) is of [2] which requires more than

6 seconds for a single secure computation of AES. In, [34, 11], it is shown how to reduce costs drastically using massive parallelism, requiring only several tens of ms per 2PC invoked. Our protocol works in the online/offline setting, while [2, 11] work in the single-execution setting and [34] works in the batch setting. The online/offline setting has great applicability. Our work demonstrates that in this setting, it is possible to carry out secure two-party computation with security for malicious adversaries with efficiency that is several orders of magnitude less than previous non-massively parallel implementations, and has the potential to cost much less in the massively parallel setting (since all the expensive steps of the protocol can be done in parallel). We stress that the offline/online setting is preferable to the batch setting since online executions can be run in isolation.

Different 2PC protocols, that are not based on the cut-and-choose technique for garbled circuits, are presented in [29] and [7]. Both protocols have an offline stage in which the parties work independently of their inputs, and a much shorter online stage in which the players use their inputs and compute the function of interest. The cryptographic work required by these protocols during the online stage is very small (if any). However, both protocols require a number of interaction rounds that depends on the depth of the circuit being evaluated.

The overall online stage of [29] costs 4 seconds (for a single computation) for computing AES, while the amortized offline time is at least 1 second (even for many computations). For many computations (135), the total online time is 15 seconds. This gives an amortized time of just 111ms, but *high latency*. The average total running time (i.e., the sum of the offline and online timings for a single AES computation) is at least 1.6 seconds (for all numbers tested in [29]). In [8], optimizations and improvements were made to [29] that enables running many AES executions in parallel. The best results obtained there provide an online time of 9962 ms for 680 AES operations in parallel. This yields a low average cost (about 14 ms per AES), but a high latency. The online time for a single execution is expected to be similar to [29]. Regarding [7], the cost of the offline stage for computing AES is about 156 seconds and the cost of the online stage (with 50 rounds of communication) is about 20 ms [35]. However, both [29] and [7] have many rounds of communication; thus in slower networks (e.g., between different Amazon regions) they will perform poorly. (All measurements reported in these works are for networks with very low latency.)

We note that in [29] and [7], the offline stage is independent of the circuit being evaluated in the online stage, whereas in our protocol, a single circuit is fixed for all computations. Thus, they are better suited for settings in which the function to be computed is not known ahead of times. In addition, [7] has two significant advantages over our protocol: (1) it can work with more than two parties, and (2) it can work with arithmetic circuits, which for some types of computations is much more efficient.

Overall, it is clear that when considering *two parties*, *Boolean circuits* and the *offline/online setting*, our protocol and prototype implementation outperform all existing solutions. In high latency networks, the improvement is by orders of magnitude.

7 Conclusion

The first implementation of cut-and-choose based 2PC was presented in [31] in 2009. It required 1114 seconds for a single computation of AES. Since then, many algorithmic and engineering improvements have been presented, gradually reducing the cost of AES computation to 264 seconds [33], to 6 seconds [2], and to even 1.4 seconds [20] and 0.46 seconds [11] when using massive parallelism. This work continues this line of work and shows how the costs can be further reduced

using recent and new algorithmic improvements (though in a slightly different, yet very natural setting). When preprocessing 1024 executions, the average online time is less than 10 ms and the amortized offline time is only 74 ms. As we use most state-of-the-art techniques (e.g., the protocol of [26], the garbling scheme of [5], and the OT extension of [3]), these timings are the result of incredible work carried out by the community on all aspects of the protocol, together with our new consistency check. We find these results to be exciting as they are more than four orders of magnitude better than the one of [31], carried out just 6 years ago.

Acknowledgements. We would like to thank Moriya Farbstein, Meital Levy and Asaf Cohen for the implementation of the protocol and its extensive performance evaluation.

References

[1] SCAPI, 2015. http://crypto.biu.ac.il/scapi and https://github.com/cryptobiu/scapi.

[2] A. Afshar, P. Mohassel, B. Pinkas, and B. Riva. Non-interactive secure computation based on cut-and-choose. In *EUROCRYPT 2014*, Springer (LNCS 8441), pages 387–404, 2014.

[3] G. Asharov, Y. Lindell, T. Schneier, and M. Zohner. More efficient oblivious transfer extensions with security for malicious adversaries. In *EUROCRYPT 2015*, Springer (LNCS 9056), pages 673–701, 2015.

[4] M. Bellare, A. Boldyreva, and S. Micali. Public-key encryption in a multi-user setting: Security proofs and improvements. In *EUROCRYPT 2000*, pages 259–274. Springer, 2000.

[5] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *IEEE Symposium of Security and Privacy*, 2013.

[6] M. Bellare, V. T. Hoang, and P. Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In *ASIACRYPT 2012*, Springer (LNCS 7658), pages 134–153, 2012.

[7] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO 2012*, Springer (LNCS 7417), pages 643–662, 2012.

[8] I. Damgård, R. Lauritsen, and T. Toft. An empirical study and some improvements of the minimac protocol for secure computation. In *SCN 2014*, Springer (LNCS 8642), pages 398–415, 2014.

[9] I. Damgård and J. B. Nielsen. Perfect hiding and perfect binding universally composable commitment schemes with constant expansion factor. In *CRYPTO 2002*, Springer (LNCS 2442), pages 581–596, 2002.

[10] Y. Ejgenberg, M. Farbstein, M. Levy, and Y. Lindell. SCAPI: The secure computation application programming interface. *Cryptology ePrint Archive,* Report 2012/629, 2012. http://eprint.iacr.org/.

[11] T. K. Frederiksen, T. P. Jakobsen, and J. B. Nielsen. Faster maliciously secure two-party computation using the GPU. *Cryptology ePrint Archive*, Report 2014/270, 2014. http://eprint.iacr.org/.

[12] T. K. Frederiksen and J. B. Nielsen. Fast and maliciously secure two-party computation using the GPU. In *ACNS 2013*, Springer (LNCS 7954), pages 339–356, 2013.

[13] O. Goldreich and E. Kushilevitz. A perfect zero-knowledge proof system for a problem equivalent to the discrete logarithm. In *Journal of Cryptology*, 6(2):97–116, 1993.

[14] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In the 19*th ACM STOC*, pages 218–229, 1987.

[15] C. Hazay, J. Katz, C.-Y. Koo, and Y. Lindell. Concurrently-secure blind signatures without random oracles or setup assumptions. In *TCC 2007*, Springer (LNCS 4392) pages 323–341, 2007.

[16] Y. Huang, J. Katz, and D. Evans. Efficient secure two-party computation using symmetric cut-and-choose. In *CRYPTO 2013*, Springer (LNCS 8043), pages 18–35, 2013.

[17] Y. Huang, J. Katz, V. Kolesnikov, R. Kumaresan, and A. J. Malozemoff. Amortizing garbled circuits. In *CRYPTO 2014*, Springer (LNCS 8617), pages 458–475, 2014.

[18] G. Kol and M. Naor. Cryptography and game theory: Designing protocols for exchanging information. In R. Canetti, editor, *TCC*, pages 320–339. Springer, 2008.

[19] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free xor gates and applications. In *ICALP 2008*, Springer (LNCS 5126), pages 486–498, 2008.

[20] B. Kreuter, A. Shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security*, pages 14–14, 2012.

[21] Y. Lindell. Highly-efficient universally-composable commitments based on the DDH assumption. In *EUROCRYPT 2011*, Springer (LNCS 6632), pages 446–466, 2011.

[22] Y. Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In *CRYPTO 2013*, Springer (LNCS 8043), pages 1–17, 2013.

[23] Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *EUROCRYPT 2007*, Springer (LNCS 4515), pages 52–78, 2007.

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

[25] Y. Lindell and B. Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In the 8*th TCC*, Springer (LNCS 6597), pages 329–346, 2011.

[26] Y. Lindell and B. Riva. Cut-and-choose yao-based secure computation in the online/offline and batch settings. In *CRYPTO 2014*, Springer (LNCS 8617), pages 476–494, 2014.

[27] P. Mohassel and M. K. Franklin. Efficiency tradeoffs for malicious two-party computation. In *PKC 2006*, Springer (LNCS 3958), pages 458–473, 2006.

[28] P. Mohassel and B. Riva. Garbled circuits checking garbled circuits: More efficient and secure two-party computation. In *CRYPTO 2013*, Springer (LNCS 8043), pages 36–53, 2013.

[29] J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. In *CRYPTO 2012*, Springer (LNCS 7417), pages 681–700, 2012.

[30] C. Peikert, V. Vaikuntanathan, and B. Waters. A framework for efficient and composable oblivious transfer. In *CRYPTO 2008*, Springer (LNCS 5157) ,pages 554–571, 2008.

[31] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In *ASIACRYPT 2009*, Springer (LNCS 5912), pages 250–267, 2009.

[32] M. O. Rabin, Y. Mansour, S. Muthukrishnan, and M. Yung. Strictly-black-box zero-knowledge and efficient validation of financial transactions. In *ICALP 2012*, Springer (LNCS 7391), pages 738–749, 2012.

[33] A. Shelat and C.-H. Shen. Two-output secure computation with malicious adversaries. In *EUROCRYPT 2011*, Springer (LNCS 6632), pages 386–405. Springer, 2011.

[34] A. Shelat and C.-h. Shen. Fast two-party secure computation with minimal assumptions. In *ACM CCS*, pages 523–534, 2013.

[35] N. Smart. Personal communication, February 2015.

[36] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, pages 162–167, 1986.

A Efficient Perfectly-Hiding Extractable Commitment

Perfect hiding together with extractability sounds like a contradiction in terms. This problem is solved by ensuring perfect hiding when the preprocess phase is run between an honest P_2 and a potentially malicious P_1, but is perfectly binding and even extractable in a simulation with P_2.

In the standard model, we can use dual-mode commitments. We use the exact construction of [21] that is based on [9, 15, 18, 30], and is secure under the DDH assumption. Let $(\mathbb{G}, q, g, h)$ be the description of a group of order q with two distinct generators g_1, g_2.

- **Preprocessing:**
  1. The receiver R chooses a random $\rho_1 \in_R \mathbb{Z}_q$ and sets $\rho_2 = \rho_1 + 1$. It then computes $h_1 = g_1^{\rho_1}$ and $h_2 = g_2^{\rho_2}$.
  2. The receiver R and committer C engage in a zero-knowledge proof of knowledge of ρ_1, where R plays the prover and C plays the verifier. The statement proven is that $(g_1, g_2, h_1, \frac{h_2}{g_2})$ is a Diffie-Hellman tuple.
- **Commitment:** To commit to $x \in \mathbb{G}$, choose random $R, S \in \mathbb{Z}_q$ and compute $u = g_1^R \cdot g_2^S$ and $v = h_1^R \cdot h_2^S \cdot x$. The commitment value is is $c_x = (u, v)$.
- **Decommitment:** The decommitment is canonical; the committer sends R, S, x and the receiver verifies that (u, v) is correctly constructed with these values.

We begin by showing that the commitment is perfectly hiding, assuming that h_1, h_2 are correctly constructed (by the soundness of the zero-knowledge proof, this holds in a real interaction except with negligible probability). In order to see this, observe that $u = g_1^R \cdot g_2^S$ and $v = g_1^{R\rho_1} \cdot g_2^{S\rho_2} \cdot x$. Letting $g_2 = g_1^w$ for some w, we have that $u = g_1^{R+wS}$ and $v = g_1^{R\rho_1 + wS\rho_2}$. Considering the matrix $\begin{pmatrix} R & wS \\ R\rho_1 & wS\rho_2 \end{pmatrix}$ we have that its determinant equals $RwS\rho_2 - RwS\rho_1 = RwS(\rho_2 - \rho_1) = RwS \neq 0$ (recall that $\rho_2 = \rho_1 + 1$). Thus the equations are linearly independent. This implies that u and v are uniformly distributed in $\mathbb{G}$, over the choice of R and S. Thus, for every $x \in G$ there exist $R, S \in \mathbb{Z}_q$ such that (u, v) is a commitment to m with randomness R and S. This implies that x is perfectly hidden.

We now show the simulation, for which the commitment is extractable (and perfectly binding). The simulator $\mathcal{S}$ for a corrupted committer C works as follows:

- **Simulator preprocessing:**
  1. The simulator $\mathcal{S}$ chooses a random $\rho \in_R \mathbb{Z}_q$ and computes $h_1 = g_1^\rho$ and $h_2 = g_2^\rho$. $\mathcal{S}$ hands $\mathcal{A}$ the pair (h_1, h_2). $\mathcal{S}$ defines the trapdoor $\mathsf{td} = \rho$.
  2. $\mathcal{S}$ runs the zero-knowledge simulator with $\mathcal{A}$ as the verifier, for the statement $(g_1, g_2, h_1, \frac{h_2}{g_2})$. (Note that the statement is *not* in the language. Nevertheless, the simulator can be run.)
- **Simulator extraction:** Upon receiving a commitment $c_x = (u, v)$, simulator $\mathcal{S}$ computes $c = v/u^\rho$.

First, note that the view of the committer is indistinguishable from a real execution, by the DDH assumption and the zero-knowledge property of the proof. Second, note that extraction works since $h_1 = (g_1)^\rho$ and $h_2 = (g_2)^\rho$; therefore:

$$\frac{v}{u^\rho} = \frac{h_1^R \cdot h_2^S \cdot x}{(g_1^R \cdot g_2^S)^\rho} = \frac{h_1^R \cdot h_2^S \cdot x}{h_1^R \cdot h_2^S} = x.$$

B The Full Protocol Specification

FIGURE B.1 (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 Lemmas 2.3 and 2.4.
- The parties decide on two circuits: **(1)** A circuit $C\left(x, y^{(1)}, y^{(2)}\right)$ that computes $f\left(x, (Ey^{(1)}) \oplus y^{(2)}\right)$, with $y^{(2)}$ being public-input wires. **(2)** A cheating-recovery circuit $C'(x, D, d^{(1)}, d^{(2)})$ that outputs x if $E'd^{(1)} \oplus d^{(2)} = D$, and outputs 0 otherwise, where $d^{(2)}$ and D being public-input wires. E and E' are probe-resistance matrices, generated by P_2 as shown in Section 2.3.

  We require that both circuits are constructed as described in Section 2.3 so that the values of $Ey^{(1)}$ and $E'd^{(1)}$ remain private even if $s-1$ bits of $y^{(1)}$ or $d^{(1)}$ are revealed.

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

1. The parties run the cut-and-choose sub protocol from Figure B.2 with the circuit C and parameters p, N and B.
2. The parties run the cut-and-choose sub protocol from Figure B.2 with the circuit C' and parameters p', N and B'. (Note that the same N is used in both cut-and-choose, so both result in the same number of buckets.)

We chose to simplify the description by using the cut-and-choose as a sub-protocol. However, the calls to $\mathcal{F}_{\mathsf{ExComΔZK}}$ for the masks of C and C' *must be made together* since P_2 should learn the XORs of the masks for the circuits that are placed in the same bucket. In the proof, we assume that the steps of the two cut-and-choose sub protocols are done in parallel, and thus the calls to $\mathcal{F}_{\mathsf{ExComΔZK}}$ can be made together.

From now on, we refer to the elements of the second cut-and-choose with prime. E.g. π' is the mapping function of the second execution from above (while π is of the first one). Also, 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 $j = 1, \ldots, N$ and $i = 1, \ldots, B$).

Running OTs for C and for C':

1. P_2 chooses $y_1^{(1)}, \ldots, y_N^{(1)} \in_R \{0,1\}^{|y^{(1)}|}$, and $d_1^{(1)}, \ldots, d_N^{(1)} \in_R \{0,1\}^{|d^{(1)}|}$.
2. P_1 acts as the sender in $\mathcal{F}_{\mathsf{ot}}$ and P_2 as the receiver. For bucket $j = 1, \ldots, N$, the parties execute $|y^{(1)}|$ OTs, where in the ith OT P_2 inputs the ith bit of $y_j^{(1)}$, and P_1 inputs the set of labels in the entire bucket that correspond to 0 in the ith bit, and the set of labels in the entire bucket that correspond to 1, both concatenated with their decommitments related to lc's. (Recall that the labels are XORed with λ_j.)

   The parties do the same for circuit C', where P_2 inputs the bits of $d_{j,i}^{(1)}$.

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

1. P_1 stores $(seed_{j,i}, m_{j,i}, lc_{j,i}, ld_{j,i}, \lambda_{j,i})$ for $i = 1, \ldots, B$, and similarly for all bundles of C'.
2. P_2 stores $y_j^{(1)}$, $lc_{j,i}$ and $gc_{j,i}$ for $i = 1, \ldots, B$. In addition, it stores the labels it received for its input $y_j^{(1)}$ from the OTs, the values of Δ_j, and similarly for all the bundles of C'.

FIGURE B.2 (Creating a Garbled-Circuit Bundle)

Public Parameters:

- A circuit $C(x, y^{(1)}, y^{(2)})$ with $y^{(2)}$ being public-input wires, or, a circuit $C(x, D, y^{(1)}, y^{(2)})$ with $D, y^{(2)}$ being public-input wires

Constructing the jth bundle:

1. Choose a random seed $seed_j \in_R \{0,1\}^k$. All the randomness needed in the next steps is derived from $\mathrm{PRF}_{seed}(\cdot)$.
2. Choose a random $m_j \in_R \{0,1\}^{|x|}$ and $\lambda_j \in_R \{0,1\}^k$.
3. Construct a garbled circuit gc_j in which 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 $l \in_R \{0,1\}^{k-1}$). We use an adaptively-secure garbling scheme as described in Section 2.2, in which all input-wire labels are XORed with λ.
4. Commit to input-wire labels associated with x, permuted according to m, by
$$\left\{\left(i, \mathsf{commit}(\lambda_j \oplus W_{j,i}^{m_{j,i}}), \mathsf{commit}(\lambda_j \oplus W_{j,i}^{1-m_{j,i}})\right)\right\}_{i \in \mathsf{In}(C,x)}.$$
where $\mathsf{In}(C,x)$ denotes the set of input wires in the circuit C associated with input x.
5. Commit to input-wire labels associated with $y^{(1)}$ and $y^{(2)}$ by
$$\left\{(i, \mathsf{commit}(\lambda_j \oplus W_{j,i}^0), \mathsf{commit}(\lambda_j \oplus W_{j,i}^1))\right\}_{i \in \mathsf{In}(C,y^{(1)}) \cup \mathsf{In}(C,y^{(2)})}.$$
6. If D is an input to the circuit, commit to all input-wire labels of D by
$$\left\{(i, \mathsf{commit}(\lambda_j \oplus W_{j,i}^0), \mathsf{commit}(\lambda_j \oplus W_{j,i}^1))\right\}_{i \in \mathsf{In}(C,D)}.$$
7. Commit to all output-wire labels by $\mathsf{commit}\left(\{j, i, W_{j,i}^0, W_{j,i}^1\}_{i \in \mathsf{Out}(C)}\right)$.[a]
8. Let lc_j be the union of the above sets of label commitments, and let ld be the set of all the corresponding decommitments.
9. Output $(gc_j, lc_j; seed_j, ld_j, m_j, \lambda_j)$.

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

FIGURE B.3 (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.

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

1. *The cut:* P_2 sets σ to be a random string of length M that has exactly NB ones.
2. *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.

   Define $\pi_\sigma : [M] \to [N]$ to be the function that maps the NB non-zero bits in σ according to π, and maps the zero bits all to -1. Let B_i be the set $\{j | \pi_\sigma(j) = i\}$ for $i = 1, \ldots, N$.
3. P_2 commits to σ and $seed_\pi$ using $\mathsf{ExtractCom}(\cdot)$.

The cut-and-choose:

1. For $j = 1, \ldots, M$, P_1 runs the garbled-circuit bundle construction procedure in Figure B.2 with the circuit C, and receives $(gc_j, lc_j; seed_j, ld_j, m_j, \lambda_j)$.
2. P_1 sends $gc_1, \ldots, gc_M$ and $lc_1, \ldots, lc_M$ to P_2, and commits to their seeds and λ_j's by $\mathsf{ExtractCom}(seed_1), \ldots, \mathsf{ExtractCom}(seed_M)$ and $\mathsf{ExtractCom}(\lambda_1), \ldots, \mathsf{ExtractCom}(\lambda_M)$.
3. P_2 inputs to $\mathcal{F}_{\mathsf{ExComAZK}}$ the sets B_i, for $i = 1, \ldots, N$, while P_1 inputs the values $m_1, \ldots, m_M$. P_2 learns the sets Δ_i for each bucket, whereas P_1 learns $B_1, \ldots, B_N$.
4. P_2 decommits to σ and $seed_\pi$. P_1 verifies that they are consistent with J and the B_i's received in the last step.
5. Let J be the set of indices that did not appear in any B_i. For $j \in J$, P_2 receives m_j from $\mathcal{F}_{\mathsf{ExComAZK}}$. In addition, P_1 decommits to $seed_j$ and λ_j to P_2.
6. P_2 computes the set $\{gc_j, lc_j\}_{j \in J}$ using the seeds it received and verifies that everything is correct.

FIGURE B.4 (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:

1. P_2 sends $y^{(2)} = y \oplus y_j^{(1)} E$ to P_1.
2. P_1 sends $x_1 = x \oplus m_1$ to P_2.
3. For $i = 1, \ldots, B$,
   (a) P_1 decommits λ_i.
   (b) P_1 sends the input-wire labels for $y^{(2)}$ and for $x_i = x_1 \oplus m_1 \oplus m_i = x \oplus m_i$ in gc_i, and the decommitments of those labels for the corresponding commitments in lc_i. (Recall that P_2 knows $m_1 \oplus m_i$ from Δ_j, thus, can compute the value of x_i by itself.)

Continued in Figure B.5.

FIGURE B.5 (The Online Stage – Continued)

Evaluating C – continued:

4. P_2 checks that all of the decommitments are valid, and that it received all of the input-wire labels indeed associated with x_i. If no, it aborts.
5. P_1 chooses a random $D \in_R \{0,1\}^k$.
6. For $v \in \mathsf{Out}(C)$,
   (a) P_1 chooses a random $R_v \in_R \{0,1\}^k$ (for masking the output wires)
   (b) Let $W^b_{i,v}$ be the bth label of output wire v of gc_i, where $v \in \mathsf{Out}(C)$. P_1 sends $\left(W^0_{i,v} \oplus R_v,\ W^1_{i,v} \oplus R_v \oplus D\right)$ for $i = 1, \ldots, B$.
7. P_1 sends $\mathsf{H}(D)$.
8. 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 $\mathsf{H}(\cdot)$ and comparing with the value that P_1 has sent). If so, it sets d to D. Otherwise, it sets $d \in \{0,1\}^s$.
9. If all evaluations (that ended) returned the same output, set z to be that output (but do not halt).

Evaluating C':

1. Let $d^{(1)}$ the input that P_2 used in the OTs for circuit C' in bucket j. P_2 sends $d^{(2)} = d \oplus d^{(1)}E'$.
2. P_1 sends D, and for $i = 1, \ldots B'$, and:
   (a) P_1 decommits to λ'_i.
   (b) P_1 sends the labels that correspond to D and $d^{(2)}$ in gc'_i, and decommits to the corresponding commitments from lc'_i.
   (c) P_1 sends the input-wire labels for $x'_i = x_1 \oplus m_1 \oplus m'_i = x \oplus m'_i$ in gc'_i, and the decommitments of those labels for the corresponding commitments in lc'_i. (Again, recall that P_2 knows $m_1 \oplus m'_i$ from Δ_j.)
3. P_1 decommits to the output labels of gc_i, for $i = 1, \ldots B$ (i.e. revealing all output wire labels of the garbled circuits for C).
4. P_2 verifies all decommitments, all the values $\left(W^0_{i,v} \oplus R_v,\ W^1_{i,v} \oplus R_v \oplus D\right)$, for $i = 1, \ldots, B$ and $v \in \mathsf{Out}(C')$, and the hash $\mathsf{H}(D)$, and aborts if there is a problem.
5. P_2 evaluates gc'_i, for $i = 1, \ldots B'$, and takes the majority output to be $\hat{x}$.

P_2's output:

1. If all evaluation circuits of C returned the same output z, then P_2 outputs z.
2. Else, if P_2 has learned earlier d such that $\mathsf{H}(d) = \mathsf{H}(D)$, then it outputs $f(\hat{x}, y)$.
3. 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.