Faster Two-Party Computation Secure Against Malicious Adversaries in the Single-Execution Setting

Xiao Wang
University of Maryland
wangxiao@cs.umd.edu

Alex J. Malozemoff
University of Maryland
amaloz@cs.umd.edu

Jonathan Katz
University of Maryland
jkatz@cs.umd.edu

Abstract

We propose a new protocol for two-party computation, secure against malicious adversaries, that is significantly faster than prior work in the single-execution (i.e., non-amortized) setting. In particular, our protocol requires only $O(\rho)$ public key operations and ρ garbled circuits, where ρ is the *statistical* security parameter, whereas previous work with the same number of garbled circuits required either $O(\rho \cdot n)$ public-key operations (where n is the input/output length) or another execution of a separate malicious two-party protocol.

We implement our protocol to evaluate its performance. Our prototype is able to securely compute AES in only 65 ms over a local-area network using a single thread without any pre-computation, only 3× slower than a semi-honest execution of the same functionality, and 22× faster than the best prior work in the single-execution setting. On a local-area network, our protocol requires around 20 μs to process each input/output bit and around 4 μs to process each AND gate, along with a fixed cost of around 23 ms to compute the base oblivious transfers.

1 Introduction

Secure multi-party computation (MPC) allows multiple parties with private inputs to compute some agreed-upon function such that all parties learn the output while keeping their inputs private. Introduced in the 1980s [Yao82], MPC has become more practical in recent years, with several companies using the technology: e.g., Dyadic [dya] uses MPC to help secure cryptographic keys; Sharemind [sha] uses MPC to process financial data [BKK+16], among other things; and Partisia [par] uses MPC for privacy-preserving auctions. A particularly important subfield of MPC is that of *two-party computation* (2PC), which is the focus of this work.

Many existing applications and implementations of 2PC assume that all participants are *semi-honest*, that is, they follow the protocol but can try to learn sensitive information from the protocol transcript. However, in real-world applications this assumption may not be justified. Although protocols with stronger security guarantees exist, most 2PC protocols secure against malicious adversaries are far from practical, especially when compared to protocols in the semi-honest setting. For example, one recent construction of malicious 2PC reports about six seconds for two parties to securely compute an AES circuit [AMPR14]. In the offline/online setting (where pre-processing is used) and timings are amortized over 1024 executions, it is possible to achieve an *online* time of 9 ms [LR15] per execution, however the total amortized time per execution (i.e., including the offline time) is still around 74 ms, with a latency of around 75,000 ms before the first 2PC can be executed. By way of comparison, in the semi-honest setting AES can be securely computed in just 22 ms.

On a related note, almost all existing 2PC schemes with security against malicious adversaries perform poorly on even moderate-sized inputs or very large circuits. For example, the schemes of Lindell [Lin13] and Afshar et al. [AMPR14] require a number of public-key operations at least proportional to the statistical security parameter *times* the sum of one party's input length and the output length. Furthermore, the most efficient scheme [LR15], which is tailored for an offline/online setting, does not scale to large circuits due to

Protocol	ρ	Time	Notes
PSSW09 [PSSW09]	40	1,114 s	
sS11 [sS11]	40	192 s	
NNOB12 [NNOB12]	55	4,000 ms	
KsS12 [KSS12]	80	1,400 ms	256 nodes/party
FN13 [FN13]	39	1,082 ms	GPU
AMPR14 [AMPR14]	40	5,860 ms	
FJN14 [FJN14]	40	455 ms	GPU
LR15 [LR15]	40	1,442 ms	
Here	40	65 ms	

Table 1: Two-party computation of AES, with security against malicious adversaries, in the single-execution setting. The statistical security parameter is ρ. All numbers except for LR15 are taken directly from the relevant paper, and are based on different hardware/network configurations. The numbers for LR15 are from our own experiments. See §6 for more details.

memory constraints: the garbled circuits created during the offline phase need to either be stored in memory, in which case evaluating very large circuits is almost impossible, or else must be written/read from disk, in which case the online time incurs a huge penalty due to disk I/O (see §6), not included in the number reported above or in [LR15].

Motivated by these issues, we design a new 2PC protocol for the malicious setting. Our protocol uses the cut-and-choose paradigm [LP07] and the input-recovery approach introduced by Lindell [Lin13], but does not need an extra 2PC protocol or a large number of public-key operations. We make the following contributions.

- Our protocol is more efficient, and often much more efficient, than the previous best protocol with malicious security in the single-execution setting (see Table 1). More concretely, our protocol takes only **65 ms** to evaluate an AES circuit over a local-area network, better than the most efficient prior work in the same setting, and even better than the amortized total time per execution in [LR15].
- We identify and fix various bottlenecks in 2PC building blocks that may prove useful in subsequent work. As an example, we use Streaming SIMD Extensions (SSE) instructions to improve the performance of oblivious-transfer extension, and improve the efficiency of the XOR-tree technique to avoid high (non-cryptographic) complexity when operating over large inputs. Our new construction reduces the cost of processing the circuit evaluator's input by 1000× for a 65,536-bit input.
- Our implementation is open sourced as a part of the `EMP-toolkit`[1], with the aim of providing a benchmark for secure computation and allowing other researchers to experiment with and extend our code.

1.1 High-Level Approach

Our protocol is based on the cut-and-choose paradigm. Let f be the circuit the parties want to compute. One party, denoted as the *garbler*, begins by generating s garbled circuits of f and sending these to the other party, the *evaluator*. Some portion of those circuits are checked for correctness by the evaluator, and the remaining circuits are evaluated in order to learn the output.

Input recovery. To achieve statistical security $2^{-\rho}$, early cut-and-choose protocols [sS13, sS11, LP11] required $s \approx 3\rho$. Lindell [Lin13] introduced the *input-recovery* technique and demonstrated a protocol requiring only $s = \rho$ garbled circuits. At a high level, the input-recovery technique allows P_2 to obtain P_1's input x if P_1 cheats; in that case, P_2 can compute the function itself to learn the output. For example, in

[1] https://github.com/emp-toolkit

one way of instantiating this approach [Lin13] all the garbled circuits use output-wire labels with the same XOR difference Δ. That is, for output wire i in every garbled circuit, the output-wire label corresponding to '0,' say $Z_{i,0}$, is random whereas the output-wire label corresponding to '1,' $Z_{i,1}$ is set to $Z_{i,1} := Z_{i,0} \oplus \Delta$. If P_2 learns different outputs for output wire i in two different gabled circuits—which means that P_1 cheated—then P_2 recovers Δ. (The protocol is set up so that Δ is not revealed by the check circuits.) The parties then run a *second* 2PC protocol in which P_2 learns x if it knows Δ; here, *input-consistency checks* are used to enforce that P_1 uses the same input x as before. Although this input-consistency protocol was further optimized by Lindell and Riva [LR15], the fact remains that two phases of secure computation are required, adding additional communication rounds and complexity.

Afshar et al. [AMPR14] addressed this issue by designing an input-recovery mechanism that does not require a secondary 2PC protocol.[2] In their scheme, P_1 first commits to its input bit-by-bit using ElGamal encryption; that is, for each bit $x[i]$ of x, P_1 sends $(g^r, h^r g^{x[i]})$ to P_2, where $h := g^\omega$ for some ω known only to P_1. Note that if P_2 learns ω then it can decrypt everything and thus learn x. Now, for each output wire P_1 secret shares ω as ω_0 and ω_1 with $\omega = \omega_0 + \omega_1$, and sends $\{Z_{i,b} + \omega_b\}_{b\in\{0,1\}}$ to P_2 for each output-wire label $Z_{i,b}$. Thus, if P_2 learns two different output-wire labels, P_2 can recover ω. Afshar et al. are able to avoid an extra 2PC protocol by using homomorphic properties of the ElGamal encryption scheme to efficiently check that the encrypted values are valid. As this needs to be done *per* input bit, this incurs a multiplicative overhead of ρ in terms of the number of public-key operations required.

Our construction uses this general idea, but the key innovation is that we are able to replace most of the public-key operations required by Afshar et al. with symmetric-key operations; see §3 for details.

Input consistency. We also need a way to enforce that P_1 uses the same input x across the different garbled circuits. Afshar et al. address this issue by using efficient zero-knowledge proofs to prove that the ElGamal ciphertexts sent by P_1 all commit to the same bit across all the evaluation circuits. However, this approach again requires many public-key operations.

We observe that it is not actually necessary to ensure that P_1 uses the same input x across all evaluation circuits and the input-recovery protocol. Rather, we only need to enforce that x is used in the input-recovery protocol and *at least one* of the evaluation circuits. This results in much better efficiency; see §3 for details.

Preventing a selective-failure attack. One other standard attack that must be prevented is a selective-failure attack whereby a malicious P_1 uses one valid input-wire label and one invalid input-wire label (for P_2's inputs) in the oblivious-transfer step. If care is not taken, P_1 could potentially use this to learn a bit of P_2's input by observing whether P_2 aborts or not. Lindell and Pinkas [LP07] proposed to deal with this using the *XOR-tree approach* in which P_2 replaces each bit its input by ρ random bits that XOR to the actual bit. By doing so, it can be shown that the probability with which P_2 aborts is *independent* of its actual input. The XOR-tree approach increases the number of oblivious transfers needed by a factor of ρ. However, this can be improved by using a ρ-probe matrix [LP07, sS13], which only increases the number of bits by a constant factor.

Nevertheless, this constant-factor blow-up in the number of input bits corresponds to a *quadratic* blow-up in the number of XOR operations required. Somewhat surprisingly (given that these XORs are non-cryptographic operations), this can become quite prohibitive. For example, for inputs as small as 4096 bits, we find that the time to simply compute all the XORs required for the XOR-tree is more than 3 seconds. We resolve this bottleneck by breaking P_2's input into small chunks and constructing smaller ρ-probe matrices for each chunk. See §5 for details.

Results. Combining the above solutions, as well as other optimizations identified in §5, we present a new 2PC protocol with provable security against malicious adversaries; see §4 for the full description. Implementing this protocol, we find that it outperforms prior work by up to several orders of magnitude; see §6.

[2] While other approaches exist for avoiding this extra 2PC protocol [Bra13, FJN14], they require additional circuits to be garbled, and thus we focus on the work of Afshar et al. here.

1.2 Related Work

Secure two-party computation based on garbled circuits has seen much progress in recent years. In the semi-honest setting, AES can be securely computed in 22 ms, whereas in the malicious setting AES can be computed in 5.8 s [AMPR14] on a standard machine, 460 ms[3] when using GPUs [FJN14], and 74 ms in the offline/online setting [LR15]. Here we focus on malicious 2PC protocols based on the *cut-and-choose* paradigm. We note that while other approaches exist for achieving malicious security (e.g., [FJN+13, NNOB12, DLT14]), they often come at the cost of a high number of rounds and are not as efficient as cut-and-choose protocols in the single-execution setting when preprocessing is not used.

The cut-and-choose approach. Lindell and Pinkas [LP07] first adopted the cut-and-choose technique to garbled circuits to achieve malicious security. Their construction requires 680 garbled circuits for statistical security 2^{-40}, which has been improved over a sequence of works [sS11, LP11, HKE13, Lin13, AMPR14] to the point where now only 40 circuits are required.

The first implementation of a 2PC protocol with malicious security was by Lindell et al. [LPS08]. Since then, multiple systems [PSSW09, NNOB12, KSS12, sS13] have been proposed that further improved efficiency, including using GPUs to accelerate the computation [FJN14, FN13].

Building blocks. One building block in our protocol is the garbling scheme formally defined by Bellare et al. [BHR12]. It is improved by a series of works [PSSW09, KMR14, KS08, BHKR13], and the best garbling scheme is by Zahur et al. [ZRE15]. Oblivious transfer (OT) is another important building block. Efficient protocols have been proposed, some of which are by Peikert et al. [PVW08] and by Chou and Orlandi [CO15]. Ishai et al. [IKNP03] showed how to extend OT efficiently from smaller number of OTs, which has been improved by some following works [NNOB12, ALSZ15, KOS15].

2 Preliminaries

Let κ be the security parameter, and let ρ be the statistical security parameter. For bitstring x, let $x[i]$ denote the ith bit of x. We use the notation $a := f(\cdots)$ to denote the output of a deterministic function, $a \leftarrow f(\cdots)$ to denote the output of a randomized function, and $a \in_R S$ to denote choosing a uniform value from set S. Let $[n] = \{1, \ldots, n\}$. We use the notation $(\mathsf{c}, \mathsf{d}) \leftarrow \mathsf{Com}(x)$ for a commitment scheme, where c and d are the commitment and decommitment of x, respectively. Due to space constraints, we refer the reader to related papers [BHR12, LP09] for the basics of garbled circuits. Throughout this paper, we use P_1 and P_2 to denote the circuit garbler and circuit evaluator, respectively. We let n_1, n_2, and n_3 denote P_1's input length, P_2's input length, and the output length, respectively.

In Figure 2, we include the functionality of $\mathcal{F}_{\mathsf{OT}}$ and $\mathcal{F}_{\mathsf{cOT}}$. $\mathcal{F}_{\mathsf{cOT}}$ is the weaker flavor of committing OT, which is also used by Jawurek et al. [JKO13]: in addition to the OT functionality, it also allows sender to open all the messages sent to the functionality before. As also Jawurek et al. [JKO13], $\mathcal{F}_{\mathsf{cOT}}$ can be made compatible with OT extension by letting sender commit to a seed and run a secure OT protocol using the seed PRG as the random tape.

Two-party computation. We use a (standard) ideal functionality for two-party computation in which the output is only given to P_2; this can be extended to deliver (possibly different) outputs to both parties using known techniques [LP07, sS11].

ρ-probe matrix. A ρ-probe matrix, used to prevent selective-failure attacks, is a binary matrix $M \in \{0,1\}^{n_2 \times m}$ such that for any $L \subseteq [n_2]$, the Hamming weight of $\bigoplus_{i \in L} M_i$ (where M_i is the ith row of M) is at least ρ. If P_2's actual input is y, then P_2 computes its effective input by sampling a random y' such that $y := My'$.

The original ρ-probe matrix proposed by Lindell and Pinkas [LP07] requires $m := \max\{4n_2, 8\rho\}$. Shelat and Shen [sS13] improved this to $m := n_2 + O(\rho + \log(n_2))$. Lindell and Riva [LR15] proposed to append an identity matrix to M to ensure that M is full rank, and to make it easier to find y' such that $y := My'$.

[3] This does not include base OTs and GPU initialization.

Functionality $\mathcal{F}_{\mathsf{OT}}$

Private inputs: P_1 has input $x \in \{0,1\}^n$ and P_2 has input $\{X_{i,b}\}_{i\in[n],b\in\{0,1\}}$.

1. Receive x from P_1 and $\{X_{i,b}\}_{i\in[n],b\in\{0,1\}}$ from P_2, and send $\{X_{i,x[i]}\}_{i\in[n]}$ to P_1.

Figure 1: Functionality $\mathcal{F}_{\mathsf{OT}}$ for oblivious transfer.

Functionality $\mathcal{F}_{\mathsf{cOT}}$

Private inputs: P_1 has input $x \in \{0,1\}^n$ and P_2 has input $\{X_{i,b}\}_{i\in[n],b\in\{0,1\}}$.

1. Receive x from P_1 and $\{X_{i,b}\}_{i\in[n],b\in\{0,1\}}$ from P_2, and send $\{X_{i,x[i]}\}_{i\in[n]}$ to P_1.
2. If receive open from P_2, send $\{X_{i,b}\}_{i\in[n],b\in\{0,1\}}$ to P_1, if receive abort from P_2, send $\perp$ to P_1.

Figure 2: Functionality $\mathcal{F}_{\mathsf{cOT}}$ for committing oblivious transfer.

3 Protocol Intuition

We already presented a high-level description of the cut-and-choose approach in §1.1; here, we describe in more detail the intuition behind the changes we introduce.

In our protocol, the two parties first run ρ instances of oblivious transfer (OT), where in instance j P_1 sends a random key key_j and random seed seed_j, while P_2 chooses whether to learn key_j (thereby putting j in the evaluation set) or seed_j (thereby putting j in the check set). The protocol is designed such that key_j can be used to recover the input-wire labels associated with P_1's input, whereas seed_j can be used to recover all the randomness used to generate the jth garbled circuit. Thus far, the structure of our protocol is similar to that of Afshar et al. [AMPR14]. However, we differ in how we recover P_1's input if P_1 is caught cheating and how we ensure P_1's consistency.

Input recovery. Recall that we want to ensure that if P_2 detects cheating by P_1, then P_2 can recover P_1's input. This is done by encoding some trapdoor in the output-wire labels of the garbled circuit such that if P_2 learns *both* labels on some output wire then it can recover the trapdoor and thus learn P_1's input. In more detail, input recovery consists of the following steps:

1. P_1 commits to its input x using some trapdoor.
2. P_1 sends garbled circuits and the input-wire labels associated with x, using an input-consistency protocol (discussed below) to enforce that consistent input-wire labels for x are used.
3. P_1 and P_2 run some protocol such that if P_2 detects cheating by P_1, then P_2 gets the trapdoor without P_1 learning this fact.
4. P_2 either (1) detects cheating, recovers x using the trapdoor, and outputs $f(x,y)$, or (2) outputs the (consistent) output of all the garbled circuits, which is $f(x,y)$.

In Afshar et al. [AMPR14] the above is done using ElGamal commitments and efficient zero-knowledge checks to enforce input consistency. However, this approach leads to $O(\rho(n_1+n_3))$ public-key operations. In contrast, our protocol achieves the same functionality with only $O(\rho)$ public-key operations.

Our scheme works as follows. Assume for ease of presentation that P_1's input x is a single bit and the output of the function is also a single bit. The parties run an OT protocol in which P_1 inputs x and P_2 inputs two random labels X_0, X_1, with P_1 receiving X_x. Then, for each j, P_1 "commits" to x by computing $R_{j,x} := \mathrm{PRF}_{\mathsf{seed}_j}(\text{“}R\text{”}) \oplus X_x$ and sending an encryption of $R_{j,x}$ under key_j to P_2. Note that P_1 cannot "commit" to $1-x$ unless P_1 can guess X_{1-x}. Also, x remains hidden from P_2 because P_2 knows either key_j

Notation	Meaning
$\mathcal{E}$	cut-and-choose set
E	ρ-probe matrix
GC_j	jth garbled circuit
$A_{j,i,b}$	ith input-wire labels for P_1 in GC_j
$B_{j,i,b}$	ith input-wire labels for P_2 in GC_j
$Z_{j,i,b}$	ith output-wire labels in GC_j
$T_{j,i,b}$	ith output mapping table for GC_j
$R_{j,i,b}$	P_1's input commitment
C_j, D_j	Input recovery elements

Table 2: Notation used in our protocol.

or seed_j for each j, but not both. The trapdoor to recover x is seed_j for any evaluation circuit j, which (in conjunction with key_j that P_2 already has) allows P_2 to recompute X_x (and hence x) .

The next step is to devise a way for P_2 to recover seed_j if it learns *inconsistent* output-wire labels in two different evaluation circuits. We do this as follows. Let $Z_{j,0}, Z_{j,1}$ be the two output-wire labels of the jth circuit. P_1 chooses some random value Δ and secret shares this value as Δ_0, Δ_1 such that $\Delta = \Delta_0 \oplus \Delta_1$. Next, it encrypts each share under the appropriate output-wire label and sends these encryptions to P_2. Thus, if P_2 learns $Z_{j,0}$ it can recover Δ_0 and if it learns $Z_{j,1}$ it can recover Δ_1. If it learns *both* output-wire labels, it can then of course recover Δ.

P_1 and P_2 then run a protocol which guarantees that if P_2 knows Δ then it recovers seed_j, and otherwise it learns nothing. This is done as follows. P_2 sets $\Omega = \Delta$ if it learned Δ, and sets $\Omega := 1$ otherwise. P_2 then computes $(h, g_1, g_2) := (g^\omega, g^r, h^r\Omega)$, for random ω and r, and sends (h, g_1, g_2) to P_1. Then, for each index j, party P_1 then computes $C_j := g^{s_j} h^{t_j}$ and $D_j := g^{s_j}(h/\Delta)^{t_j}$, for random s_j and t_j, and sends C_j along with an encryption of seed_j under D_j. Note that if $\Omega = \Delta$, then $C_j^r = D_j$ and thus P_2 can recover seed_j, whereas if $\Omega \neq \Delta$ then P_2 learns nothing.

Of course, the protocol as described does not account for the fact that P_1 can send invalid messages or otherwise try to cheat. However, by carefully integrating appropriate correctness checks as part of the cut-and-choose, we can guarantee that if P_1 tries to cheat then P_2 either aborts (due to detected cheating) or learns P_1's input.

Input consistency. As discussed in §1.1, prior schemes enforce that P_1 uses the same input x in all garbled circuits and the input-recovery mechanism. However, we observe that this is not necessary. Intuitively, we only need to ensure that P_1 uses the same input in the input-recovery mechanism and *at least one* evaluated garbled circuit. Even if P_1 cheats by using different inputs in two different evaluated garbled circuits, P_2 can always obtain the correct output: if P_2 learns only one output then this is the correct output; if P_2 learns multiple outputs, then the input-recovery procedure helps P_2 learn the correct output.

We ensure such consistency by integrating the consistency check with cut-and-choose as follows. Recall that in our input-recovery scheme, P_1 sends to P_2 "commitments" $R_{j,x} := \mathrm{PRF}_{\mathsf{seed}_j}(\text{“}R\text{”}) \oplus X_x$ for each index j. After these values are sent, P_2 sends $X_0 \oplus X_1$ to P_1, allowing P_1 to learn both X_0 and X_1. P_1 then sends a random permutation of $\mathsf{Com}(A_{j,0}, \mathrm{PRF}_{\mathsf{seed}_j}(\text{“}R\text{”}) \oplus X_0)$ and $\mathsf{Com}(A_{j,1}, \mathrm{PRF}_{\mathsf{seed}_j}(\text{“}R\text{”}) \oplus X_1)$, where $A_{j,0}, A_{j,1}$ are P_1's input-wire labels to the jth garbled circuit. P_1 also sends $\mathsf{Enc}_{\mathsf{key}_j}(\mathsf{Decom}(\mathsf{Com}(A_{j,x}, R_{j,x})))$. Note that (1) if P_2 chooses j as a check circuit then it can check the correctness of the commitment pair, since everything is computed from seed_j, and (2) if P_2 choose j as an evaluation circuit then it can recover P_1's input-wire label $A_{j,x}$ and check if the X_x received before is the same as the label decommitted.

4 Scheme

We present the full details of our protocol in Figure 3. To aid in understanding the protocol, we also present a graphical depiction in Figure 4. We summarize some important notations in Table 2 for reference.

Protocol Π_{2pc}

Private inputs: P_1 has input $x \in \{0,1\}^{n_1}$ and P_2 has input $y \in \{0,1\}^{n_2}$.

Common inputs: ρ-probe matrix $E \in \{0,1\}^{n_2 \times m}$, where $m = O(n_2)$; circuit $f : \{0,1\}^{n_1} \times \{0,1\}^{n_2} \to \{0,1\}^{n_3}$; and circuit $f' : \{0,1\}^{n_1} \times \{0,1\}^{m} \to \{0,1\}^{n_3}$ such that $f'(x, y') = f(x, Ey')$

Protocol:

1. P_1 picks random κ-bit strings $\{\mathsf{key}_j, \mathsf{seed}_j\}_{j\in[\rho]}$, and sends them to $\mathcal{F}_{\mathsf{OT}}$. P_2 picks $\mathcal{E} \in_R \{0,1\}^{\rho}$ and sends $\mathcal{E}$ to $\mathcal{F}_{\mathsf{OT}}$ and receives $\{\mathsf{seed}_j\}_{j\notin\mathcal{E}}$ and $\{\mathsf{key}_j\}_{j\in\mathcal{E}}$.
2. P_1 computes $\{B_{j,i,b} := \mathrm{PRF}_{\mathsf{seed}_j}(i, b, \text{“}B\text{”})\}_{j\in[\rho],i\in[m],b\in\{0,1\}}$ and sends $\{B_{1,i,b} \| \cdots \| B_{\rho,i,b}\}_{i\in[m],b\in\{0,1\}}$ to $\mathcal{F}_{\mathsf{OT}}$. P_2 chooses random $y' \in_R \{0,1\}^m$ such that $y = Ey'$, sends y' to $\mathcal{F}_{\mathsf{OT}}$, and receives $\{B_{1,i,y'[i]} \| \cdots \| B_{\rho,i,y'[i]}\}_{i\in[m]}$.
3. P_2 sends random labels $\{X_{i,b}\}_{i\in[n_1],b\in\{0,1\}}$ to $\mathcal{F}_{\mathsf{cOT}}$. P_1 sends x to $\mathcal{F}_{\mathsf{cOT}}$ and receives $\{X_{i,x[i]}\}_{i\in[n_1]}$. For $j \in [\rho], i \in [n_1]$, P_1 computes $R_{j,i,x[i]} := \mathrm{PRF}_{\mathsf{seed}_j}(i, \text{“}R\text{”}) \oplus X_{i,x[i]}$, and sends $\mathsf{Enc}_{\mathsf{key}_j}(\{R_{j,i,x[i]}\}_{i\in[n_1]})$ to P_2. P_2 sends open to $\mathcal{F}_{\mathsf{cOT}}$, which sends $\{X_{i,0}, X_{i,1}\}_{i\in[n_1]}$ to P_1, and for $j \in \mathcal{E}$ uses key_j to decrypt and learn $R_{j,i,x[i]}$.
4. For $j \in [\rho], i \in [n_1]$, P_1 computes $R_{j,i,1-x[i]} := R_{j,i,x[i]} \oplus X_{i,0} \oplus X_{i,1}$, $\{A_{j,i,b} := \mathrm{PRF}_{\mathsf{seed}_j}(i, b, \text{“}A\text{”})\}_{b\in\{0,1\}}$, $\{(\mathsf{c}^R_{j,i,b}, \mathsf{d}^R_{j,i,b}) \leftarrow \mathsf{Com}(R_{j,i,b}, A_{j,i,b})\}_{b\in\{0,1\}}$ using randomness derived from seed_j, and sends $\{(\mathsf{c}^R_{j,i,0}, \mathsf{c}^R_{j,i,1})\}$ (in random permuted order) and $\mathsf{Enc}_{\mathsf{key}_j}(\{\mathsf{d}^R_{j,i,x[i]}\}_{i\in[n_1]})$ to P_2. For $j \in \mathcal{E}, i \in [n_1]$, P_2 opens $\mathsf{c}^R_{j,i,x[i]}$ to obtain $R_{j,i,x[i]}$ and $A_{j,i,x[i]}$, and checks that $R_{j,i,x[i]}$ equals the value from Step 3. If any decommitment is invalid or any check fails, P_2 aborts.
5. P_1 picks random κ-bit labels Δ, $\{\Delta_{i,0}\}_{i\in[n_3]}$, sets $\{\Delta_{i,1} := \Delta_{i,0} \oplus \Delta\}_{i\in[n_3]}$, and sends $\{H(\Delta_{i,b})\}_{i\in[n_3],b\in\{0,1\}}$ to P_2. For $j \in [\rho]$, P_1 computes garbled circuit GC_j for function f' using $A_{j,i,b}, B_{j,i,b}$ as the input-wire labels and randomness derived from seed_j for internal wire labels. Let $Z_{j,i,b}$ denote the output-wire labels. P_1 computes $\{T_{j,i,b} := \mathsf{Enc}_{Z_{j,i,b}}(\Delta_{i,b})\}_{i\in[n_3],b\in\{0,1\}}$ and $(\mathsf{c}^T_j, \mathsf{d}^T_j) \leftarrow \mathsf{Com}(\{T_{j,i,b}\}_{i\in[n_3],b\in\{0,1\}})$ using randomness derived from seed_j, and sends GC_j, c^T_j, and $\mathsf{Enc}_{\mathsf{key}_j}(\mathsf{d}^T_j)$ to P_2.
6. For $j \in \mathcal{E}$, P_2 decrypts to learn d^T_j and opens c^T_j to learn $\{T_{j,i,b}\}_{i\in[n_3],b\in\{0,1\}}$; if any decommitment is invalid, P_2 aborts. P_2 evaluates GC_j using labels $\{A_{j,i,x[i]}\}_{i\in[n_1]}$ and $\{B_{j,i,y'[i]}\}_{i\in[m]}$, and obtains output-wire label $Z_{j,i}$. P_2 checks the validity of these labels by checking if $H(\mathsf{Dec}_{Z_{j,i}}(T_{j,i,b}))$ matches $H(\Delta_{i,b})$ for some $b \in \{0,1\}$, and if so sets $z'_j[i] = b$; else it sets $z'_j[i] = \perp$.
   - *Invalid Circuits.* If, for every $j \in \mathcal{E}$, there is some i with $z'_j[i] = \perp$, then P_2 sets $\Omega := 1, z := \perp$.
   - *Inconsistent Output Labels.* Else if, for some $i \in [n_3], j_1, j_2 \in \mathcal{E}$, P_2 obtains $z'_{j_1}[i] = 0$ and $z'_{j_2}[i] = 1$, then P_2 sets $\Omega := \mathsf{Dec}_{Z_{j_1,i}}(T_{j_1,i,0}) \oplus \mathsf{Dec}_{Z_{j_2,i}}(T_{j_2,i,1})$. If different Ωs are obtained, P_2 sets $z := \perp$.
   - *Consistent Output Labels.* Else, for all i, set $z[i] = z'_j[i]$ for the first index j such that $z'_j[i] \neq \perp$, and set $\Omega := 1$.
7. P_2 picks $\omega, r \in_R \mathbb{F}_q$, and sends $(h, g_1, h_1) := (g^\omega, g^r, h^r\Omega)$ to P_1. P_1 sends Δ and $\{\Delta_{i,b}\}_{i\in[n_3],b\in\{0,1\}}$ to P_2, who checks that $\{\Delta = \Delta_{i,0} \oplus \Delta_{i,1}\}_{i\in[n_3]}$ and that $H(\Delta_{i,b})$ matches the values P_1 sent in Step 5; if any check fails, P_2 aborts. For $j \in [\rho]$, P_1 picks $s_j, t_j \in_R \mathbb{F}_q$ using randomness derived from seed_j, computes $C_j := g^{s_j} h^{t_j}, D_j := g_1^{s_j} \left(\frac{h_1}{\Delta}\right)^{t_j}$, and sends C_j and $\mathsf{Enc}_{D_j}(\mathsf{seed}_j)$ to P_2. For $j \in \mathcal{E}$, P_2 uses C^r_j to decrypt and obtains some seed'_j.
8. If $\Omega \neq 1$, P_2 recovers x as follows: For $j \in \mathcal{E}, i \in [n_1]$, if $R_{j,i,x[i]} = \mathrm{PRF}_{\mathsf{seed}'_j}(i, \text{“}R\text{”}) \oplus X_{i,0}$, P_2 sets $x_j[i] := 0$; if $R_{j,i,x[i]} = \mathrm{PRF}_{\mathsf{seed}'_j}(i, \text{“}R\text{”}) \oplus X_{i,1}$, P_2 sets $x_j[i] := 1$; and otherwise, P_2 sets $x_j[i] := \perp$. If no valid x_j is obtained, or more than two different x_j are obtained, P_2 sets $z := \perp$; otherwise P_2 sets $z := f(x_j, y)$.
9. If any of the following checks fail for any $j \notin \mathcal{E}$, P_2 aborts; otherwise P_2 outputs z.
   (a) For $i \in [m]$, the $B_{j,i,y'[i]}$ value received in Step 2 equals $\mathrm{PRF}_{\mathsf{seed}_j}(i, y'[i], \text{“}B\text{”})$.
   (b) GC_j is computed correctly using $A_{j,i,b} := \mathrm{PRF}_{\mathsf{seed}_j}(i, b, \text{“}A\text{”})$ and $B_{j,i,b} := \mathrm{PRF}_{\mathsf{seed}_j}(i, b, \text{“}B\text{”})$ as input-wire labels and randomness derived from seed_j.
   (c) Compute $T_{j,i,b}$ using $Z_{j,i,b}$ from GC_j and $\Delta_{i,b}$ sent by P_1, and check that c^T_j is computed correctly with randomness derived from seed_j.
   (d) The $C_j, \mathsf{Enc}_{D_j}(\mathsf{seed}_j)$ values in Step 7 are correctly computed, using Δ and seed_j.
   (e) For $i \in [n_1], b \in \{0,1\}$, $\mathsf{c}^R_{j,i,b}$ is correctly computed using seed_j, $A_{j,i,b}$, and $R_{j,i,b}$, which are also computed from seed_j.

Figure 3: The full description of our malicious 2PC protocol.

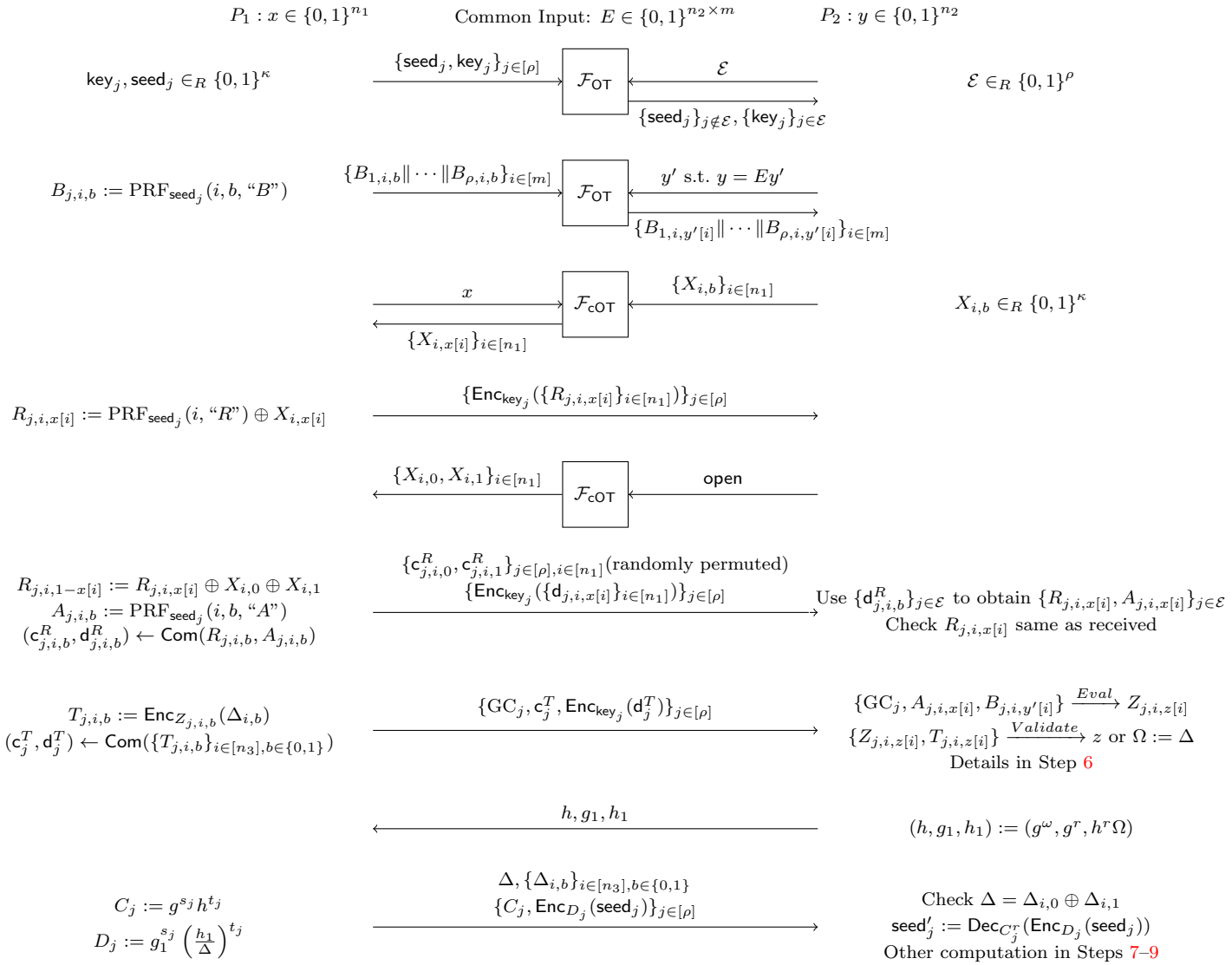

Figure 4: Graphical depiction of Π_{2pc}.

Our protocol, including the optimizations detailed in §5, requires $O(\rho(n_1 + n_2 + n_3 + |C|))$ symmetric operations and $O(\rho)$ group element operations. Furthermore, most of the symmetric operations, including circuit garbling and computing the PRFs, can be accelerated using hardware AES.

Theorem 4.1. *Let* $(\mathsf{Com}, \mathsf{Open})$ *be a computational hiding and binding commitment scheme, let the garbling scheme satisfy authenticity, privacy, and obliviousness, let* H *be a one-way, collision-resistant hash function, and assume the decisional Diffie-Hellman assumption holds. Then the protocol in Figure 3 securely computes* f *in the* $(\mathcal{F}_{\mathsf{OT}}, \mathcal{F}_{\mathsf{cOT}})$*-hybrid model with security* $2^{-\rho} + neg(\kappa)$.

Proof. We consider separately the case where P_1 or P_2 is malicious.

Malicious P_1. Our proof is based on the fact that with all but negligible probability, P_2 either aborts or learns the output $f(x, y)$, where x is the input P_1 sent to $\mathcal{F}_{\mathsf{cOT}}$ in Step 3, and y is P_2's input. Given this, the simulator essentially acts as an honest P_2 using $y = 0$, extracts P_1's input x from the call to $\mathcal{F}_{\mathsf{cOT}}$, and outputs $f(x, y)$ if no party aborts.

We now proceed to the formal details. Let $\mathcal{A}$ be an adversary corrupting P_1; we construct a simulator $\mathcal{S}$ interacting with an ideal functionality $\mathcal{F}$ evaluating f, defined as follows.

1–2 $\mathcal{S}$ acts as an honest P_2, using input $y := 0$.

3 $\mathcal{S}$ acts as an honest P_2, and obtains the input x that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{cOT}}$. It sends x to $\mathcal{F}$.

4–6 $\mathcal{S}$ acts as an honest P_2, where if P_2 would abort then $\mathcal{S}$ sends abort to $\mathcal{F}$ and halts, outputting whatever $\mathcal{A}$ outputs.

7–8 $\mathcal{S}$ acts as an honest P_2 using $\Omega := 1$, where if P_2 would abort then $\mathcal{S}$ sends abort to $\mathcal{F}$ and halts, outputting whatever $\mathcal{A}$ outputs.

9 $\mathcal{S}$ acts as an honest P_2, except that after the check in Step 9a, $\mathcal{S}$ also checks if $\{B_{j,i,b}\}_{j\notin\mathcal{E},i\in[m],b\in\{0,1\}}$ are correctly computed and aborts if, for at least ρ different $i \in [m]$, $\{B_{j,i,b}\}_{j\notin\mathcal{E},b\in\{0,1\}}$ contains incorrect values. If P_2 would abort then $\mathcal{S}$ sends abort to $\mathcal{F}$ and halts, outputting whatever $\mathcal{A}$ outputs; otherwise, $\mathcal{S}$ sends continue to $\mathcal{F}$.

We now show that the views in the hybrid and ideal worlds are indistinguishable.

$\mathbf{H_1}$. Same as the hybrid-world protocol, where $\mathcal{S}$ plays the role of an honest P_2 using the actual input y.

$\mathbf{H_2}$. Same as $\mathbf{H_1}$, except that $\mathcal{S}$ extracts the input x that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{cOT}}$, uses $y := 0$ throughout the protocol, except as explained below, and sends x to $\mathcal{F}$ if no party aborts. $\mathcal{S}$ also performs the additional checks as described above in Step 9 of the simulator. Moreover, $\mathcal{S}$ sets the value of Ω to what it would obtain if it was using its actual input y (rather than what it would obtain using $y := 0$).

There are two cases on how $\mathcal{A}$ would cheat here, and we address each in turn. For simplicity, we denote $I \subset [m]$ as the set of indices used in the selective failure attack, that is I is the set of indices i such such that $B_{j,i,b}$ is not honestly computed.

1. $\mathcal{A}$ launches a selective-failure attack with $|I| < \rho$. Lemma 4.4 ensures that in $\mathbf{H_1}$ $\mathcal{S}$ either aborts or learns $f(x, y)$ with probability at least $1 - 2^{-\rho}$. In $\mathbf{H_2}$, $\mathcal{S}$ either aborts or learns $f(x, y)$ with probability 1. Further, since less than ρ wires are corrupted, the probability of abort due to the selective-failure attack is exactly the same in both hybrids. Therefore the distribution between $\mathbf{H_1}$ and $\mathbf{H_2}$ is different by at most $2^{-\rho}$.
2. $\mathcal{A}$ launches a selective-failure attack with $|I| \geq \rho$. By the security of the ρ-probe matrix [LR15], $\mathcal{S}$ aborts in $\mathbf{H_1}$ with probability at least $1 - 2^{-\rho}$. If $\mathcal{A}$ cheats elsewhere, the probability of abort would be even higher than $1 - 2^{-\rho}$.
   In $\mathbf{H_2}$, $\mathcal{S}$ aborts with probability 1, therefore there is at most $2^{-\rho}$ difference between $\mathbf{H_1}$ and $\mathbf{H_2}$.

Thus, in any case the $\mathbf{H_1}$ and $\mathbf{H_2}$ can be distinguished by at most $2^{-\rho}$ probability.

$\mathbf{H_3}$. Same as $\mathbf{H_2}$, except $\mathcal{S}$ always sets $\Omega := 1$ in Step 7.

In $\mathbf{H_3}$, P_2 sends $(h, g_1, h_1) := (g^\omega, g^r, g^{\omega r})$, which is indistinguishable from $(g^\omega, g^r, g^{\omega r}\Omega)$ by the decisional Diffie-Hellman problem. Thus, the views in $\mathbf{H_2}$ and $\mathbf{H_3}$ are computationally indistinguishable.

As $\mathbf{H_3}$ is the same as the ideal world protocol, the proof is complete.

Malicious P_2. Here, we need to simulate the correct output $f(x, y)$ towards P_2. Rather than simulate the actual garbled circuit, as is done in most prior work, we modify the output mapping tables $\{T_{j,i,b}\}$ to encode the correct output. At a high level, the simulator acts as an honest P_1 with $x = 0$, which lets P_2 learn $f(0, y)$ when evaluating the garbled circuits. The simulator then "tweaks" the output mapping tables $\{T_{j,i,b}\}$ such that if the ith bit of $f(0, y)$ and $f(x, y)$ are different then P_2 learns the opposite value.

We now proceed to the formal details. Suppose there exists an adversary $\mathcal{A}$ corrupting P_2; we construct a simulator $\mathcal{S}$ as follows.

1 $\mathcal{S}$ acts as an honest P_1 and obtains the set $\mathcal{E}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{OT}}$.

2 $\mathcal{S}$ acts as an honest P_1, and obtains the input y' that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{OT}}$. $\mathcal{S}$ computes y from y', sends (input, y) to $\mathcal{F}$, which sends back $z := f(x, y)$ to $\mathcal{S}$.

3–4 $\mathcal{S}$ acts as an honest P_1 with input $x := 0$. If $\mathcal{A}$ send abort to $\mathcal{F}_{\mathsf{cOT}}$, $\mathcal{S}$ aborts, outputting whatever $\mathcal{A}$ outputs.

5 $\mathcal{S}$ acts as an honest P_1, except as follows. $\mathcal{S}$ computes $z' := f(0, y)$ and for $j \in \mathcal{E}$, $i \in [n_3]$, and $b \in \{0, 1\}$, sets $T_{j,i,b} := \mathsf{Enc}_{Z_{j,i,b}}(\Delta_{1-b})$ if $z[i] \neq z'[i]$.

6–7 $\mathcal{S}$ acts as an honest P_1.

We now show that the views in the hybrid and ideal worlds are indistinguishable.

$\mathbf{H_1}$. Same as the hybrid-world protocol, where $\mathcal{S}$ plays the role of an honest P_1.

$\mathbf{H_2}$. Same as $\mathbf{H_1}$, except as follows. $\mathcal{S}$ extracts P_2's input y from $\mathcal{F}_{\mathsf{OT}}$ and sends (input, y) to $\mathcal{F}$, receiving back z. $\mathcal{S}$ uses $x := 0$ throughout the simulation and modifies the "tweaks" $\{T_{j,i,b}\}$ as is done by the simulator using knowledge of z.

The view between $\mathbf{H_1}$ and $\mathbf{H_2}$ are the same expect that in $\mathbf{H_1}$, P_2 gets $Z_{j,i,z[i]}$ and $T_{j,i,z[i]} := \mathsf{Enc}_{Z_{j,i,z[i]}}(\Delta_{i,z[i]})$, while in $\mathbf{H_2}$, if $z[i] \neq z'[i]$, P_2 gets $Z_{j,i,1-z[i]}$ and $T_{j,i,1-z[i]} := \mathsf{Enc}_{Z_{j,i,1-z[i]}}(\Delta_{i,z[i]})$ instead. In both hybrids, P_2 cannot learn the other output label due to authenticity of the garbled circuit.

By the obliviousness property of the garbling scheme, $Z_{j,i,0}$ and $Z_{j,i,1}$ are indistinguishable. Likewise, by the security of the encryption scheme the values $T_{j,i,0}$ and $T_{j,i,1}$ are indistinguishable.

As $\mathbf{H_2}$ is the same as the ideal world protocol, the proof is complete. □

4.1 Additional Lemmas

We now build a series of lemmas towards proving Lemma 4.4, which we use above to prove security for a malicious P_1. We begin with a definition of what it means for an index $j \in [\rho]$ to be "good".

Definition 4.1. *Consider an adversary $\mathcal{A}$ corrupting P_1, and denote $\{\mathsf{seed}_j\}$ as the labels $\mathcal{A}$ sent to $\mathcal{F}_{\mathsf{OT}}$. An index $j \in [\rho]$ is good if and only if all of the following hold.*

1. *The $B_{j,i,y'[i]}$ values $\mathcal{A}$ sent to $\mathcal{F}_{\mathsf{OT}}$ in Step 2 are computed honestly using seed_j.*
2. *The commitments $\{\mathsf{c}^R_{j,i,b}\}_{i\in[n_1],b\in\{0,1\}}$ that $\mathcal{A}$ sent to P_2 in Step 4 are computed honestly using seed_j.*
3. *GC_j is computed honestly using $\{A_{j,i,b}\}$ and $\{B_{j,i,b}\}$ as the input-wire labels and seed_j.*
4. *The values C_j and $\mathsf{Enc}_{D_j}(\mathsf{seed}_j)$ are computed honestly using seed_j and the Δ value sent by $\mathcal{A}$ in Step 7.*
5. *The commitment c^T_j is computed honestly using $\Delta_{i,b}$ and seed_j.*

It is easy to see the following.

Fact 4.1. *If an index $j \in [\rho]$ is not a good index then it cannot pass all the checks in Step 9.*

Our first Lemma shows that P_2 is able to recover the correct output-wire labels for a good index.

Lemma 4.1. *Consider an adversary $\mathcal{A}$ corrupting P_1, and denote x as the input $\mathcal{A}$ sent to $\mathcal{F}_{\mathsf{cOT}}$. If an index $j \in \mathcal{E}$ is a good index and P_2 does not abort, then P_2 learns output labels $Z_{j,i,z[i]}$, where $z = f(x, y)$, with all but negligible probability.*

Proof. Since j is good, we know that P_2 receives an honestly computed GC_j and $T_{j,i,b}$ from $\mathcal{A}$ and honest $B_{j,i,y'[i]}$ from $\mathcal{F}_{\mathsf{OT}}$. However, it is still possible that P_2 does not receive correct input labels for P_1's input that corresponds to the input x that $\mathcal{A}$ sent to $\mathcal{F}_{\mathsf{cOT}}$. We will show that this can only happen with negligible probability.

Note that if j is good, then the commitments $\{\mathsf{c}^R_{j,i,b}\}$ are computed correctly. Since P_2 obtains the $A_{j,i,x[i]}$ labels by decommitting one of these commitments, the labels P_2 gets are valid input-wire labels, although they may not be consistent with the input x that $\mathcal{A}$ sent to $\mathcal{F}_{\mathsf{cOT}}$.

Assume that for some $i \in [n_1]$, P_2 receives $A_{j,i,1-x[i]}$. This means P_2 also receives $R_{j,i,1-x[i]}$ from the same decommitment, since $\mathsf{c}_{j,i,b}$ is computed honestly. However, if P_2 does not abort, then we know that P_2 receives the same label $R_{j,i,1-x[i]}$ in Step 3 since the checks pass. We also know that

$$R_{j,i,1-x[i]} = \mathrm{PRF}_{\mathsf{seed}_j}(i, \text{“}R\text{”}) \oplus X_{i,1-x[i]}.$$

Therefore $\mathcal{A}$ needs to guess $X_{i,1-x[i]}$ correctly before P_2 sends both labels, which happens with probability at most $2^{-\kappa}$. □

Our next lemma shows that P_2 can recover x if P_1 tries to cheat on a good index.

Lemma 4.2. *Consider an adversary $\mathcal{A}$ corrupting P_1, and denote x as the input $\mathcal{A}$ sent to $\mathcal{F}_{\mathsf{cOT}}$. If an index $j \in \mathcal{E}$ is a good index, and P_2 learns $\Omega = \Delta$, then P_2 can recover $x_j = x$ in Step 8 if no party aborts.*

Proof. Since j is a good index, we know that C_j and $\mathsf{Enc}_{D_j}(\mathsf{seed}_j)$ are constructed correctly, where seed_j is the one P_1 sent to $\mathcal{F}_{\mathsf{OT}}$ in Step 1. Therefore, P_2 can recompute seed_j from them. We just need to show that P_2 is able to recover x from a good index using seed_j.

Using a similar argument as the previous proof, we can show that the label $R_{j,i,x[i]}$ that P_2 learns in Step 4 is a correctly computed label using x that P_1 sent to $\mathcal{F}_{\mathsf{cOT}}$ in Step 3: Since j is good, the $\mathsf{c}^R_{j,i,b}$ values are all good, which means that the $R_{j,i,x[i]}$ labels P_2 learns are valid. However, P_1 cannot "flip" the wire label unless P_1 guesses a random label correctly, which happens with negligible probability.

In conclusion, P_2 has the correct $R_{j,i,x[i]} = \mathrm{PRF}_{\mathsf{seed}_j}(i, \text{“}R\text{”}) \oplus X_{i,x[i]}$ and the seed_j used in the computation. Further P_2 has $X_{i,0}, X_{i,1}$. Therefore P_2 can recover x that P_1 sent to $\mathcal{F}_{\mathsf{cOT}}$ if P_2 has $\Omega = \Delta$. □

Note that given the above lemma, it may still be possible that a malicious P_1 acts in such a way that P_2 recovers different x's from different indices. In the following we show this only happens with negligible probability.

Lemma 4.3. *Consider an adversary $\mathcal{A}$ corrupting P_1 and denote x as the input P_1 sends to $\mathcal{F}_{\mathsf{cOT}}$ in Step 3. If P_2 does not abort, then P_2 recovers some $x' \neq x$ with at most negligible probability.*

Proof. We do a proof by contradiction and start by assuming that P_2 does not abort while P_2 recovers some $x' \neq x$ for some $j \in \mathcal{E}$. Let i be an index at which $x'[i] \neq x[i]$.

Since P_2 does not abort at Step 4, we can denote $R_{j,i,x[i]}$ as the label P_1 learns in Step 3, which also equals the one decommitted to in Step 4. P_2 recovering some x' means that

$$R_{j,i,x[i]} = \mathrm{PRF}_{\mathsf{seed}'_j}(i, \text{“}R\text{”}) \oplus X_{i,x'[i]},$$

where seed'_j is the seed P_2 recovers in Step 7. Therefore we conclude that

$$\begin{aligned}\mathrm{PRF}_{\mathsf{seed}'_j}(i, \text{“}R\text{”}) &= R_{j,i,x[i]} \oplus X_{i,x'[i]} \\ &= R_{j,i,x[i]} \oplus X_{i,1-x[i]}.\end{aligned}$$

Although $\mathcal{A}$ receives $X_{i,x[i]}$ in Step 3, $X_{i,1-x[i]}$ remains completely random before $\mathcal{A}$ sends $R_{j,i,x[i]}$. Further, $\mathcal{A}$ receives $X_{i,b}$ only after sending $R_{j,i,x[i]}$. Therefore, the value of $R_{j,i,x[i]} \oplus X_{i,1-x[i]}$ is completely random to $\mathcal{A}$. If $\mathcal{A}$ wants to "flip" a bit in x, $\mathcal{A}$ needs to find some seed'_j such that $\{\mathrm{PRF}_{\mathsf{seed}'_j}(i, \text{“}R\text{”})\}_{i\in[n_1]}$ equals a randomly chosen string, which is information theoretically infeasible if $n_1 > 1$. □

Finally, the last lemma shows that P_2 either aborts or learns $f(x, y)$, regardless of P_1's behavior.

Lemma 4.4. *Consider an adversary $\mathcal{A}$ corrupting P_1 and denote x as the input P_1 sends to $\mathcal{F}_{\mathsf{cOT}}$ in Step 3. With probability at least $1 - (2^{-\rho} + \epsilon(\kappa))$, P_2 either aborts or learns $f(x, y)$ for some negligible function $\epsilon(\cdot)$.*

Proof. Denote the set of P_1's good circuits as $\mathcal{E}'$ and consider the following three cases:

- $\bar{\mathcal{E}} \cap \bar{\mathcal{E}}' \neq \emptyset$. In this case P_2 aborts because P_2 checks some $j \notin \mathcal{E}'$ which is not a good index.
- $\mathcal{E} \cap \mathcal{E}' \neq \emptyset$. In this case, there is some $j \in \mathcal{E} \cap \mathcal{E}'$, which means P_2 learns $z := f(x, y)$ and $Z_{j,i,z[i]}$ from the jth garbled circuit (by Lemma 4.1). However, it is still possible that P_2 learns more than one valid z. If this is the case, P_2 learns Δ. Lemma 4.2 ensures that P_2 obtains x; Lemma 4.3 ensures that P_2 cannot recover any other valid x' even from bad indices.
- $\mathcal{E} = \mathcal{E}'$. This only happens when $\mathcal{A}$ guesses $\mathcal{E}$ correctly, which happens with at most $2^{-\rho}$ probability.

This completes the proof. □

5 Optimizations

We now discuss several protocol optimizations we discovered in the course of implementing our protocol, some of which may be applicable to other malicious 2PC implementations.

5.1 Optimizing the XOR-tree

We noticed that when using a ρ-probe matrix to reduce the number of OTs needed for the XOR-tree, we incurred a large performance hit when P_2's input was large. In particular, processing the XOR gates introduced by the XOR-tree, which are always assumed to be free due to the free-XOR technique [KS08], takes a significant amount of time. The naive XOR-tree [LP07] requires ρn OTs and ρn XOR gates; on the other hand, using a ρ-probe matrix of dimension $n \times cn$, with $c \ll \rho$, requires cn OTs but cn^2 XOR gates. We observe that this quadratic blowup becomes prohibitive as P_2's input size increases: for a 4096-bit input, it takes more than 3 seconds to compute *just* the XORs in the ρ-probe matrix of Lindell and Riva [LR15] across all circuits. Further, it also introduces a large memory usage: it takes gigabytes of memory just to store the matrix for 65,536-bit inputs.

In the following we introduce two new techniques to both asymptotically reduce the number of XOR gates required and the hidden constant factor in the ρ-probe matrix.

A general transformation to a sparse matrix. We first reduce the number of XORs needed. Assuming a ρ-probe matrix with dimensions $n \times cn$, we need $c\rho n^2$ XOR gates to process the ρ-probe matrices across all ρ circuits. Our idea to avoid this quadratic growth in n is to break P_2's input into small chunks, each of size k. When computing the random input y', or recovering y in the garbled circuits, we process each chunk individually. By doing so, we reduce the complexity to $\rho \cdot \frac{n}{k} c(k)^2 = ck\rho n$. By choosing $k = 2\rho$, this equates to a 51× decrease in computation even for just 4096-bit inputs. This also eliminates the memory issue, since we only need a very small matrix for any input size.

A better ρ-probe matrix. After applying the above technique, our problem is reduced to finding an efficient ρ-probe matrix for k-bit inputs for some small k, while maintaining a small blowup c. We show that a combination of the previous solutions [LP07, LR15] with a new tighter analysis results in a better solution, especially for small k. Our solution can be written as $A = [M \| I_k]$, where $M \in \{0,1\}^{k \times (c-1)k}$ is a random matrix and I_k as an identity matrix of dimension k. The use of I_k makes it easy to find a random y' such that $y = Ay'$ for any y, and ensures that A is full rank [LR15]. However, we show that it also helps to reduce c. The key idea is that the XOR of any i rows of A has Hamming weight at least i, contributed by I_k, so we do not need as much Hamming weight from the random matrix as the prior work [LP07].

k	40	65	80	103	143	229	520
LP07 [LP07]	6.66	4.1	4	4	4	4	4
sS13 [sS13]	7.95	5.2	4.5	4.1	3.2	2.4	1.6
This work	5.675	4	3.5	3	2.5	2	1.5

Table 3: Choices for c for ρ-probe matrix for $\rho = 40$, where k is the chunk size after applying the sparse matrix transformation.

In more detail, for each $S \subseteq [k]$, denote $M_S := \bigoplus_{i \in S} M_i$ and use random variable X_S to denote the number of ones in M_S. In order to make A a ρ-probe matrix, we need to ensure that $X_S + |S| \geq \rho$ for any $S \subseteq [k]$, because XORing any $|S|$ rows from I_k gives us a Hamming weight of $|S|$.

Note that X_S is a random variable following a binomial distribution $\mathrm{Bin}(ck - k, \frac{1}{2})$. Therefore, we can compute the probability that A is not a ρ-probe matrix as follows:

$$\begin{aligned}\Pr[A \text{ is bad}] &= \Pr\left[\bigcup_{S \subseteq [k]} X_S < \rho - |S|\right] \\ &\leq \sum_{S \subseteq [k]} \Pr[X_S < \rho - |S|] \\ &= \sum_{S \subseteq [k]} cdf(\rho - |S| - 1) = \sum_{i=1}^{k} \binom{k}{i} cdf(\rho - i - 1),\end{aligned}$$

where $cdf()$ is the cumulative distribution function for $\mathrm{Bin}(ck - k, \frac{1}{2})$. Now, for each k we can find the smallest c numerically such that $\Pr[A \text{ is bad}] \leq 2^{-\rho}$. We include some results in Table 3, and can see that our new probe matrix has smaller c than prior work [LP07, sS13]. Note that number of XOR to perform is $c\rho kn$ and number of OT needed is cn. Therefore we need to have a trade off between them, since a smaller c requires a larger k. In our implementation we use $k = 232$ and $c = 2$ to achieve the maximum overall efficiency.

Performance results. See Figure 5 for a comparison between our approach and the best previous scheme [sS13]. When the input is large, the cost of computing the ρ-probe matrix over all circuits dominates the overall cost. As we can see, our design is about 10× better for just 1,024-bit inputs and can be 1000× better for 65,536-bit inputs. We are not able to compare beyond this point, because just storing the ρ-probe matrix for 262,144 bits takes at least 8.59 GB of memory for the prior work.

5.2 Other Optimizations

Oblivious transfer with hardware acceleration. As observed by Asharov et al. [ALSZ13], matrix transposition takes a significant amount of the time during the execution of OT extension. Rather than adopting their solution using cache-friendly matrix transposition, we found that a better speedup can be obtained by using matrix transposition routines based on Streaming SIMD Extensions (SSE) instructions [mis]. The use of SSE-based matrix transposition in the OT extension protocol is also independently studied in a concurrent work by Keller et al. [KOS16] in multi-party setting.

Given a 128-bit vector of the form $a[0], \ldots, a[15]$ where each $a[i]$ is an 8-bit number, the instruction _mm_movemask_epi8 returns the concatenation of the highest bits from all $a[i]$s. This makes it possible to transpose a matrix of dimension 8×16 very efficiently in 15 instructions (8 instructions to "assemble the matrix" and 7 instructions to shift the vector left by one bit). By composing such an approach, we achieve very efficient matrix transposition, which leads to highly efficient OT extension protocols; see §6.1 for performance results.

Reducing OT cost. Although our protocol requires three instantiations of OT, we only need to construct the base OTs once. The OTs in Steps 1 and 2 can be done together, and further, by applying the observation

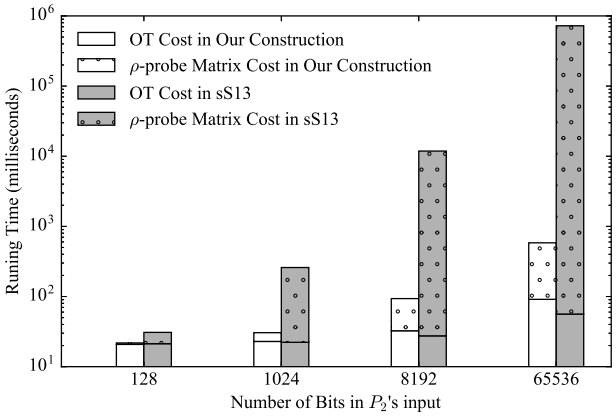

Figure 5: Comparing the cost of our ρ-probe matrix design with the prior best scheme [sS13]. When used in a malicious 2PC protocol, computing the ρ-probe matrix needs to be done ρ times, and OT extension needs to process a cn-bit input because of the blowup of the input caused by the ρ-probe matrix.

	n_1	n_2	n_3	$\|C\|$	localhost			LAN			WAN		
					SE	Offline	Online	SE	Offline	Online	SE	Offline	Online
ADD	32	32	33	127	29	60	6 (0.2)	39	27	12 (0.2)	1060	474	697 (0.2)
AES	128	128	128	6,800	50	82	14 (2)	65	62	21 (3)	1513	867	736 (2)
SHA1	256	256	160	37,300	136	156	48 (32)	200	206	52 (27)	3439	2705	820 (20)
SHA256	256	256	256	90,825	277	356	85 (144)	438	497	92 (128)	6716	5990	856 (99)

Table 4: Performance of common functions over various networks. SE stands for "single execution". All numbers are in milliseconds. Offline time includes disk I/O. For online time, disk I/O is shown separately in the parentheses.

by Asharov et al. [ALSZ13] that the "extension phase" can be iterated, we can perform more random OTs along with the OTs for Steps 1 and 2 to be used in the OTs of Step 3.

Pipelining. Pipelining garbled circuits was first introduced by Huang et al. [HEKM11] to reduce memory usage and hence improve efficiency. We adopt a similar idea for our protocol. While as written we have P_2 conduct most of the correctness checks at the end of the protocol, we note that P_2 can do most of the checks much earlier. In our implementation, we "synchronize" P_1 and P_2's computation such that P_2's checking is pipelined with P_1's computation. Pipelining also enables us to evaluate virtually any sized circuit (as long as the width of the circuit is not too large). As shown in §6.4, we are able to evaluate a 4.3 billion-gate circuit without any memory issue, something that offline/online protocols [LR15] cannot do without using lots of memory or disk I/O.

Pushing computation offline. One desired property for secure computation protocols is the ability of pre-compute before knowing the inputs. Our protocol can be modified such that all garbled circuits and most of the group element operations are done in an offline stage as follows:

1. In addition to base OT, most of the remaining public key operations can also be done offline. P_2 can send $(h, g_1) := (g^\omega, g^r)$ before knowing the input to P_1, who can compute the C_j values and half of the D_j values. During the online phase, P_1 and P_2 only need to do ρ exponentiations.

2. All garbled circuits can be computed and sent in an offline stage, with all check circuits checked. P_2 can also decommit c_j^T to learn the output translation tables for the evaluation circuits.

Building block	localhost	LAN	WAN
ρ-probe matrix for 2^{15}-bit input	5.8 ms	—	—
Garble 10^4 AES circuits	3.42 s	—	—
Garble and send 10^4 AES circuits	4.83 s	7.53 s	87.4s
2^{10} malicious base OTs	113 ms	133 ms	249 ms
8 mil semi-honest OT extension	1.52 s	2.56 s	18.1 s
8 mil malicious OT extension	4.99 s	5.64 s	25.6s
10 mil malicious OT extension	6.20 s	7.02 s	31.5 s

Table 5: Performance of our building blocks. The first row gives the running time of P_2 recovering its input when using a ρ-probe matrix. The second row gives the running time of garbling, and the third row gives the running time when both garbling and sending. The remaining rows give the performance of OT and both semi-honest and malicious OT extension.

	Our Protocol	LR15 [LR15]	AMPR [AMPR14]
ADD	39	1034	—
AES	65	1442	5860
SHA1	200	2007	—
SHA256	438	2621	7580

Table 6: Single execution performance. All numbers are in milliseconds. Numbers for LR15 [LR15] were obtained by tuning their implementation to work for single execution, using the same hardware as our results. Numbers for AMPR [AMPR14] were for single execution, not including any I/O time, and taken from their paper.

6 Implementation and Evaluation

We implemented the protocol in C++, using `RELIC` [AG] for group element operations, `libssl` for instantiating the hash function, and `libgarble` [Mal] for garbling. We adopted most of the recent advances in the field [BHKR13, ZRE15, ALSZ15, CO15, LR15] as well as various optimizations introduced in §5. The implementation is open sourced at `EMP-toolkit`[4].

We instantiate the commitment scheme as $(\text{SHA-1}(x, r), r) \leftarrow \mathsf{Com}(x)$. When x is random and has enough entropy we use the hash of x as both the commitment and decommitment.

Evaluation setup. All evaluations were performed with a single-threaded program with computational security parameter $\kappa = 128$ and statistical security parameter $\rho = 40$. We evaluated our system over three different network settings:

1. **localhost.** Experiments were run over the same machine using the loopback network interface.
2. **LAN.** Experiments were run over two `c4.2xlarge` Amazon EC2 instances with 2.32 Gbps bandwidth measured by `iperf` and less than 1 ms latency measured by `ping`.
3. **WAN.** Experiments were run over two `c4.2xlarge` Amazon EC2 instances with 200 Mbps bandwidth and 75 ms latency.

All numbers are average results of 10 runs. We observed very small variance between multiple executions which we believe is due to Amazon EC2 instances having dedicated bandwidth.

6.1 Subprotocol Performance

Because of the various optimizations mentioned in §5, as well as a carefully engineered implementation, many parts of our system perform better than previously reported implementations. We summarized these results

[4] https://github.com/emp-toolkit

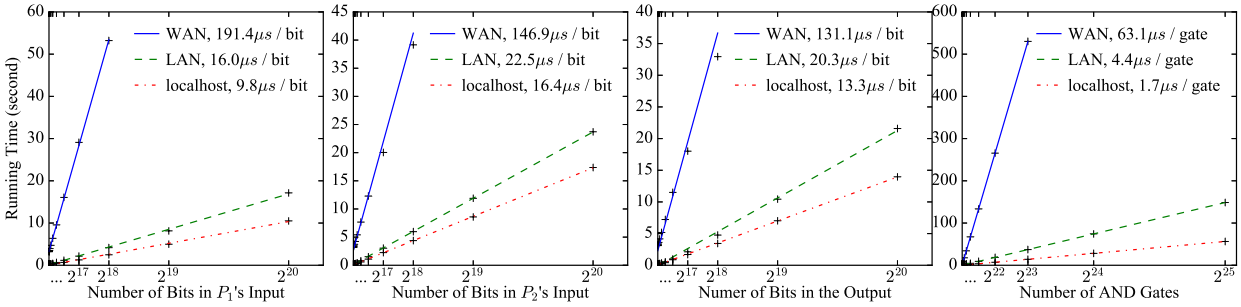

Figure 6: The performance of our protocol while modifying the input lengths, output length and the circuit size. Numbers in the figure show the slope of the lines, namely the cost to process an additional bit or gate.

Example	n_1	n_2	n_3	$\|C\|$	Running Time	Projected Time	Total Comm.	Non-GC Comm.
16384-bit cmp	16,384	16,384	1	16,383	0.67 s	0.72 s	128 MB	84%
128-bit sum	128	128	128	127	0.04 s	0.03 s	1.8 MB	91%
256-bit sum	256	256	256	255	0.05 s	0.04 s	3.4 MB	90%
1024-bit sum	1024	1,024	1,024	1,023	0.08 s	0.09 s	11.2 MB	88%
128-bit mult	128	128	128	16,257	0.13 s	0.1 s	22.4 MB	7%
256-bit mult	256	256	256	65,281	0.4 s	0.37 s	86.6 MB	3%
Sort 1024 32-bit ints	32,768	32,768	32,768	1,802,240	9.43 s	9.8 s	2.6 GB	11.5%
Sort 4096 32-bit ints	131,072	131,072	131,072	10,223,616	53.7 s	52.7 s	14.2 GB	7.7%
1024-bit modular exp	1,024	1,024	1,024	4,305,443,839	5.3 h	5.26 h	5.5 TB	0.0002%

Table 7: Performance of our implementation on additional examples. *Running Time* reports the performance of our single execution over LAN; *Projected Time* is calculated using the formula in §6.3; *Total Comm.* is the total communication as measured by our implementation; and *Non-GC Comm.* is the percentage of communication not used for garbled circuits.

in Table 5.

The `libgarble` library is able to garble about 20 million AND gates per second. When both garbling and sending through localhost, this reduces to 14 million AND gates per second due to the overhead of sending all the data through the loopback interface. Over LAN the speed is roughly 9.03 million gates per second, reaching the theoretical upper bound of $2.32 \cdot 10^9/256 = 9.06 \cdot 10^6$ gates per second.

For oblivious transfer, Asharov et al. [ALSZ15] report a running time of 11.9 seconds to compute around 8 million *random* OTs, whereas our implementation requires only around 5.64 seconds for standard OTs on 128-bit inputs. Keller et al. [KOS15] present a more efficient OT extension protocol, achieving 10 million OTs in around 9.5 seconds; even though we implement the scheme of Asharov et al., our performance is still better than that reported by Keller et al. Our semi-honest OT extension also reports the best number we are aware of: Asharov et al. [ALSZ13] report 11.4 seconds for 8 million OTs over a Gigabit LAN, about 4 times slower than the running time of our semi-honest implementation.

6.2 General Performance

We now discuss the overall performance of our protocol. Table 4 presents the running time of our protocol on several standard 2PC benchmark circuits for various network settings. For each network condition, we report a *single execution* running time, which includes all computation for one 2PC invocation, and an *offline/online* running time. In order to be comparable with Lindell and Riva [LR15], the offline time includes disk I/O and the online time does not; the time to preload all garbled circuits before the online stage starts is reported separately in parentheses.

Comparing single execution implementations. In Table 6, we compare the performance of our protocol

	Our Protocol		LR15 (1024 executions)		
	Offline	Online	Total Offline	Online	Amortized
ADD	27	12 (0.2)	89,492	5 (2.7)	92
AES	62	21 (3)	132,276	9 (5)	138
SHA1	206	52 (27)	226,570	20 (11)	241
SHA256	497	92 (128)	338,927	44 (22)	374

Table 8: Offline/online performance. All numbers are in milliseconds. Numbers for LR15 [LR15] are obtained by running their implementation on `c4.8xlarge` instances over LAN using a single thread. *Online* denote the running time of a single execution, and *Amortized* denotes the combined offline and online running time of a single execution. The disk I/O time is given in parentheses, and is *not* included in the total online time.

	localhost	LAN	WAN
Time per P_1's input bit	9.8	16	191.4
Time per P_2's input bit	16.4	22.5	146.9
Time per output bit	13.3	20.3	131.1
Time per AND gate	1.7	4.4	63.1

Table 9: Scalability of our protocol. All numbers are in microseconds per bit or microseconds per gate.

with the existing state-of-the-art implementations. The most efficient implementation for single execution of malicious 2PC without massive parallelization or GPUs we are aware of is by Afshar et al. [AMPR14]. They reported 5860 ms of computation time for AES and 7580 ms for SHA256, with disk and network I/O excluded, whereas we achieve 65 ms and 438 ms, respectively, with all I/O included. Thus our result is 17× to 90× better than their result, although ours includes network cost while theirs does not.

We also evaluated the performance of the implementation by Lindell and Riva [LR15] using the same hardware with one thread and parameters tuned for single execution, i.e., 40 main circuits and 132 circuits for input recovery. Their implementation is about 3× to 4× better than Afshar et al., but still 6× to 26× slower than our LAN results.

Comparing offline/online implementations. Although our protocol is not designed for an offline/online setting, it is possible to perform a pre-computation stage as described in §5.2. We compare such a protocol with the best-known offline/online implementation [LR15]. The detailed comparison can be found in Table 8. Note that the numbers for the comparison work were obtained by running their protocol using `c4.8xlarge` instances with a single thread and much higher bandwidth, due to their protocol requiring a significant amount of memory. Our online time is about twice as slow as the prior work, mainly due to the need to evaluate 20 garbled circuits whereas the prior work only needs to evaluate 4 circuits. However, our protocol does not require an expensive offline phase, which means we have a much smaller latency. Furthermore, even amortized over 1024 executions, the *amortized* cost per 2PC invocation (i.e., the combined offline/online cost) is worse than ours for AES. That said, we view it as an interesting direction to modify our protocol to support amortization.

6.3 Scalability

In order to understand the cost of each component of our construction, we investigated the scalability as one modifies the input lengths, output length, and circuit size. We set input and output lengths to 128 bits and circuit size as 16,384 AND gates and increase each the variables separately. In Figure 6, we show how the performance is related to these parameters.

Not surprisingly, the cost increases linearly for each parameter. We can thus provide a realistic estimate of the running time (in μs) of a given circuit of size $|C|$ with input lengths n_1 and n_2 and output length n_3 through the following formula (which is specific to the LAN setting):

$$T = 16n_1 + 22.5n_2 + 20.3n_3 + 4.4|C| + 23,000.$$

The coefficients for other network settings can be found in Table 9, with the some constant cost of the base OTs.

6.4 More Examples

Finally, in Table 7 we report the performance of our implementation in the LAN setting on several additional examples. We also show the projected time calculated based on the formula in the previous section. We observe that over different combinations of input, output and circuit sizes, the projected time calculated using the formula mentioned previously matches closely to the real results we get.

We further report the total communication and the percentage of the communication not spent on garbled circuits. We can see the percentage stays low except when the circuit is linear to the input lengths.

Acknowledgments

Work of Alex J. Malozemoff conducted in part with Government support through the National Defense Science and Engineering Graduate (NDSEG) Fellowship, 32 CFG 168a, awarded by DoD, Air Force Office of Scientific Research. Work of Xiao Wang and Jonathan Katz was supported in part by NSF awards #1111599 and #1563722.

References

[AG] D. F. Aranha and C. P. L. Gouvêa. RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic.

[ALSZ13] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS*, pages 535–548. ACM Press, November 2013.

[ALSZ15] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer extensions with security for malicious adversaries. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT*, volume 9056 of *LNCS*, pages 673–701. Springer, Heidelberg, April 2015.

[AMPR14] Arash Afshar, Payman Mohassel, Benny Pinkas, and Ben Riva. Non-interactive secure computation based on cut-and-choose. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT*, volume 8441 of *LNCS*, pages 387–404. Springer, Heidelberg, May 2014.

[BHKR13] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *IEEE S&P*, pages 478–492. IEEE Computer Society Press, May 2013.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS*, pages 784–796. ACM Press, October 2012.

[BKK+16] Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. Students and taxes: a privacy-preserving social study using secure computation. In *PETS*, 2016.

[Bra13] Luís T. A. N. Brandão. Secure two-party computation with reusable bit-commitments, via a cut-and-choose with forge-and-lose technique - (extended abstract). In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT*, volume 8270 of *LNCS*, pages 441–463. Springer, Heidelberg, December 2013.

[CO15] Tung Chou and Claudio Orlandi. The simplest protocol for oblivious transfer. In Kristin E. Lauter and Francisco Rodríguez-Henríquez, editors, *LATINCRYPT*, volume 9230 of *LNCS*, pages 40–58. Springer, Heidelberg, August 2015.

[DLT14] Ivan Damgård, Rasmus Lauritsen, and Tomas Toft. An empirical study and some improvements of the MiniMac protocol for secure computation. In Michel Abdalla and Roberto De Prisco, editors, *SCN*, volume 8642 of *LNCS*, pages 398–415. Springer, Heidelberg, September 2014.

[dya] Dyadic Security. https://www.dyadicsec.com/.

[FJN+13] Tore Kasper Frederiksen, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Peter Sebastian Nordholt, and Claudio Orlandi. MiniLEGO: Efficient secure two-party computation from general assumptions. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *LNCS*, pages 537–556. Springer, Heidelberg, May 2013.

[FJN14] Tore Kasper Frederiksen, Thomas P. Jakobsen, and Jesper Buus Nielsen. Faster maliciously secure two-party computation using the GPU. In Michel Abdalla and Roberto De Prisco, editors, *SCN*, volume 8642 of *LNCS*, pages 358–379. Springer, Heidelberg, September 2014.

[FN13] Tore Kasper Frederiksen and Jesper Buus Nielsen. Fast and maliciously secure two-party computation using the GPU. In Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *ACNS*, volume 7954 of *LNCS*, pages 339–356. Springer, Heidelberg, June 2013.

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

[HKE13] Yan Huang, Jonathan Katz, and David Evans. Efficient secure two-party computation using symmetric cut-and-choose. In Ran Canetti and Juan A. Garay, editors, *CRYPTO*, volume 8043 of *LNCS*, pages 18–35. Springer, Heidelberg, August 2013.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO*, volume 2729 of *LNCS*, pages 145–161. Springer, Heidelberg, August 2003.

[JKO13] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS*, pages 955–966. ACM Press, November 2013.

[KMR14] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. FleXOR: Flexible garbling for XOR gates that beats free-XOR. In *CRYPTO*, 2014.

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO*, volume 9215 of *LNCS*, pages 724–741. Springer, Heidelberg, August 2015.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Mascot: Faster malicious arithmetic secure computation with oblivious transfer. Cryptology ePrint Archive, Report 2016/505, 2016. http://eprint.iacr.org/2016/505.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP*, volume 5126 of *LNCS*, pages 486–498. Springer, Heidelberg, July 2008.

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

[Lin13] Yehuda Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In Ran Canetti and Juan A. Garay, editors, *CRYPTO*, volume 8043 of *LNCS*, pages 1–17. Springer, Heidelberg, August 2013.

[LP07] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In Moni Naor, editor, *EUROCRYPT*, volume 4515 of *LNCS*, pages 52–78. Springer, Heidelberg, May 2007.

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

[LP11] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In Yuval Ishai, editor, *TCC*, volume 6597 of *LNCS*, pages 329–346. Springer, Heidelberg, March 2011.

[LPS08] Yehuda Lindell, Benny Pinkas, and Nigel P. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In Rafail Ostrovsky, Roberto De Prisco, and Ivan Visconti, editors, *SCN*, volume 5229 of *LNCS*, pages 2–20. Springer, Heidelberg, September 2008.

[LR15] Yehuda Lindell and Ben Riva. Blazing fast 2PC in the offline/online setting with security for malicious adversaries. In *ACM CCS*, 2015.

[Mal] Alex J. Malozemoff. `libgarble`: Garbling library based on `JustGarble`. https://github.com/amaloz/libgarble.

[mis] mischasan. What is SSE good for? Transposing a bit matrix. https://mischasan.wordpress.com/2011/07/24/what-is-sse-good-for-transposing-a-bit-matrix/. Accessed: 2015-12-10.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO*, volume 7417 of *LNCS*, pages 681–700. Springer, Heidelberg, August 2012.

[par] Partisia Market Design. http://www.partisia.dk/.

[PSSW09] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *ASIACRYPT*, volume 5912 of *LNCS*, pages 250–267. Springer, Heidelberg, December 2009.

[PVW08] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In David Wagner, editor, *CRYPTO*, volume 5157 of *LNCS*, pages 554–571. Springer, Heidelberg, August 2008.

[sha] Sharemind. https://sharemind.cyber.ee/.

[sS11] abhi shelat and Chih-Hao Shen. Two-output secure computation with malicious adversaries. In Kenneth G. Paterson, editor, *EUROCRYPT*, volume 6632 of *LNCS*, pages 386–405. Springer, Heidelberg, May 2011.

[sS13] abhi shelat and Chih-Hao Shen. Fast two-party secure computation with minimal assumptions. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS*, pages 523–534. ACM Press, November 2013.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE Computer Society Press, November 1982.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT*, volume 9057 of *LNCS*, pages 220–250. Springer, Heidelberg, April 2015.