Faster Secure Two-Party Computation in the Single-Execution Setting

Xiao Wang[1]*, Alex J. Malozemoff[2]**, and Jonathan Katz[1]*

[1] University of Maryland, USA
{wangxiao,jkatz}@cs.umd.edu
[2] Galois, USA
amaloz@galois.com

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 setting (i.e., non-amortized and with no pre-processing). In particular, for computational security parameter κ and statistical security parameter ρ, our protocol uses only ρ garbled circuits and $O(\rho + \kappa)$ public-key operations, whereas previous work with the same number of garbled circuits required either $O(\rho \cdot n + \kappa)$ public-key operations (where n is the input/output length) or a second execution of a secure-computation sub-protocol. Our protocol can be based on the decisional Diffie-Hellman assumption in the standard model.

We implement our protocol to evaluate its performance. With $\rho = 40$, our implementation securely computes an AES evaluation in 65 ms over a local-area network using a single thread without any pre-computation, 22× faster than the best prior work in the non-amortized setting. The relative performance of our protocol is even better for functions with larger input/output lengths.

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 [38], MPC has become more practical in recent years, with several companies now using the technology. A particularly important case is secure *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

* Research supported by NSF awards #1111599 and #1563722.

** 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 done while at the University of Maryland.

security guarantees exist, 2PC protocols secure against malicious adversaries are relatively slow, especially when compared to protocols in the semi-honest setting. To address this, researchers have considered variants of the classical, "single-execution" setting for secure two-party computation, including both the *batch* setting [18,28,29,33] (in which the computational cost is amortized over multiple evaluations of the same function) and the *offline/online* setting [28,29,33] (in which parties perform pre-processing when the circuit—but not the parties' inputs—is known). The best prior result [33] (done concurrently and independently of our own work) relies on both amortization and pre-processing (as well as extensive parallelization) to achieve an overall amortized time of 6.4 ms for evaluating AES with 40-bit statistical security. Due to the pre-processing, however, it introduces a latency of 5222 ms until the first execution can be done.

In addition, existing 2PC schemes with security against malicious adversaries perform poorly on even moderate-size inputs or very large circuits. For example, the schemes of Lindell [24] and Afshar et al. [1] 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. The schemes tailored to the batch, offline/online setting [29,33] do not scale well for large circuits due to memory constraints: the garbled circuits created during the offline phase need either to 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[1] due to disk I/O (see §5).

Motivated by these issues, we design a new 2PC protocol with security against malicious adversaries that is tailored for the *single-execution* setting (i.e., no amortization) without any pre-processing. Our protocol uses the cut-and-choose paradigm [25] and the input-recovery approach introduced by Lindell [24], but the number of public-key operations required is independent of the input/output length. Overall, 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). 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.
– We identify and fix bottlenecks in various building blocks for secure computation; these fixes may prove useful in subsequent work. As an example, we use Streaming SIMD Extensions (SSE) to improve the performance of oblivious-transfer extension, and improve the efficiency of the XOR-tree technique to avoid high (non-cryptographic) complexity when applied to large inputs. Our optimizations reduce the cost of processing the circuit evaluator's input by $1000\times$ for 2^{16}-bit inputs, and even more for larger inputs.
– We release an open-source implementation, `EMP-toolkit` [36], with the aim of providing a benchmark for secure computation and allowing other researchers to experiment with, use, and extend our code.

[1] The performance numbers reported in [29,33] do not take this into account.

Protocol	ρ	Time	Notes
PSSW09 [32]	40	1,114 s	
SS11 [34]	40	192 s	
NNOB12 [31]	55	4,000 ms	
KSS12 [23]	80	1,400 ms	256 CPUs/party
FN13 [13]	39	1,082 ms	GPU
AMPR14 [1]	40	5,860 ms	
FJN14 [11]	40	455 ms	GPU
LR15 [29]	40	1,442 ms	
Here	40	65 ms	

Table 1. Times for 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 [29] are taken directly from the cited paper, and thus are based on different hardware/network configurations. The numbers for [29] are from our own experiments, using the same hardware/network configuration as for our own implementation. We do not include [33] here because it is not in the single-execution setting. See §5 for more details.

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. At a high level, party P_1, also called the *circuit garbler*, begins by generating s garbled circuits for f and sending those to P_2, the *circuit evaluator*. Some portion of those circuits (the *check circuits*) are randomly selected and checked for correctness by the evaluator, and the remaining circuits (the *evaluation circuits*) are evaluated. The outputs of the evaluation circuits are then processed in some way to determine the output.

The input-recovery technique. To achieve statistical security $2^{-\rho}$, early cut-and-choose protocols [34,26,35] required $s \approx 3\rho$. Lindell [24] introduced the *input-recovery technique* and demonstrated a protocol requiring only $s = \rho$ garbled circuits of f (plus additional, smaller garbled circuits computing another function). At a high level, the input-recovery technique allows P_2 to obtain P_1's input x if P_1 cheats; having done so, P_2 can then compute the function itself to learn the output. For example, in one way of instantiating this approach [24], every garbled circuit uses the same output-wire labels for a given output wire i, and moreover the labels on every output wire share the same XOR difference Δ. That is, for every wire i, the output-wire label $Z_{i,0}$ corresponding to '0' is random whereas the output-wire label $Z_{i,1}$ corresponding to '1' is set to $Z_{i,1} := Z_{i,0} \oplus \Delta$. (The protocol is set up so that Δ is not revealed by the check circuits.) If P_2 learns different outputs for some output wire i in two different gabled circuits—which means that P_1 cheated—then P_2 recovers Δ. 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.

Afshar et al. [1] designed an input-recovery mechanism that does not require a secondary 2PC protocol. In their scheme, P_1 first commits to its input bit-by-bit using ElGamal encryption; that is, for each bit $x[i]$ of its input, 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. As part of the protocol, P_1 sends $\{Z_{i,b} + \omega_b\}_{b\in\{0,1\}}$ to P_2 (where, as before, $Z_{i,b}$ is the label corresponding to bit b for output wire i), with $\omega = \omega_0 + \omega_1$. Now, if P_2 learns two different output-wire labels for some output wire, P_2 can recover ω and hence recover x. Afshar et al. use homomorphic properties of ElGamal encryption to enable P_2 to efficiently check that the $\{Z_{i,b} + \omega_b\}_{b\in\{0,1\}}$ are computed correctly, and for this bit-by-bit encryption of the input is required. Overall, $O(\rho \cdot n)$ public-key operations (where $|x| = n$) are needed.

Our construction relies on the same general idea introduced by Afshar et al., but our key innovation is that we are able to replace most of the public-key operations with symmetric-key operations, overall using only $O(\rho)$ public-key operations rather than $O(\rho \cdot n)$; see §3.1 for details.

Input consistency. One challenge in the cut-and-choose approach with the input-recovery technique is that P_2 needs to enforce that P_1 uses the same input x in all the evaluation circuits, as well as in the input-recovery phase. Afshar et al. address this using zero-knowledge proofs to demonstrate (in part) that the ElGamal ciphertexts sent by P_1 all commit to the same bit across all evaluation circuits. We observe that it is not actually necessary to ensure that P_1 uses the same input x across all evaluation circuits and in the input-recovery step; rather, it is sufficient to enforce that the input x used in the input-recovery step is used in *at least one* of the evaluation circuits. This results in a dramatic efficiency improvement; see §3.1 for details.

Preventing a selective-failure attack. 2PC protocols must also prevent a selective-failure attack whereby a malicious P_1 uses one valid input-wire label and one invalid input-wire label (for one of P_2's input wires) 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 or not P_2 aborts. Lindell and Pinkas [25] proposed to deal with this using the *XOR-tree approach* in which P_2 replaces each bit y_i of its input by ρ random bits that XOR to y_i. By doing so, it can be shown that the probability with which P_2 aborts is (almost) *independent* of its actual input. This approach increases the number of oblivious transfers by a factor of ρ, but this can be improved by using a ρ-probe matrix [25,35], which only increases the length of the effective input by a constant factor.

Nevertheless, this constant-factor blow-up in the number of (effective) input bits corresponds to a *quadratic* blow-up in the number of XOR operations required. Somewhat surprisingly (since these XORs are non-cryptographic operations), this blow-up can become quite prohibitive. For example, for inputs as small as 4096 bits, we find that the time to compute all the XORs required for a ρ-probe matrix is over *3 seconds*! We resolve this bottleneck by breaking P_2's input into small chunks and constructing smaller ρ-probe matrices for each chunk, thereby reducing the overall processing required. See §4 for details.

Results. Combining the above ideas, as well as other optimizations identified in §4, we obtain a new 2PC protocol with provable security against malicious adversaries; see §3.2 for a full description. Implementing this protocol, we find that it outperforms prior work by up to several orders of magnitude in the single-execution setting; see Table 1 and §5.

Subsequent work. In our extended version [37] we adopt ideas by David et al. [10] to further improve the efficiency of our protocol—especially when communication is the bottleneck—by reducing the communication required for the check circuits (as in [14]).

1.2 Related Work

Since the first implementation of a 2PC protocol with malicious security [27], many implementations with better performance (including those already discussed in the introduction) have been developed [32,31,23,35,11,13]. Although other approaches have been proposed for two-party computation with malicious security (e.g., [12,31,9]), here we focus on protocols using the *cut-and-choose* paradigm that is currently the most efficient approach in the single-execution setting when pre-processing is not used. Lindell and Pinkas [25] first showed how to use the cut-and-choose technique to achieve malicious security. Their construction required 680 garbled circuits for statistical security 2^{-40}, but this has been improved in a sequence of works [34,26,17,24,7,11,1] to the point where currently only 40 circuits are required.

2 Preliminaries

Let κ be the computational security parameter and let ρ be the statistical security parameter. For a bit-string 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.

In Figures 1 and 2, we show functionalities $\mathcal{F}_{\mathsf{OT}}$ and $\mathcal{F}_{\mathsf{cOT}}$ for parallel oblivious transfer (OT) and a weak flavor of committing OT used also by Jawurek et al. [19]. $\mathcal{F}_{\mathsf{cOT}}$ can be made compatible with OT extension as in [19].

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.

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 [25,34].

Building blocks. Our implementation of garbled circuits uses all recent optimizations [22,32,5,21,39]. Our implementation uses the base OT protocol of Chou and Orlandi [8], and the OT extension protocol of Asharov et al. [4].

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

Fig. 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. Upon receiving x from P_1 and $\{X_{i,b}\}_{i\in[n],b\in\{0,1\}}$ from P_2, send $\{X_{i,x[i]}\}_{i\in[n]}$ to P_1.
2. Upon receiving open from P_2, send $\{X_{i,b}\}_{i\in[n],b\in\{0,1\}}$ to P_1.

Fig. 2. Reactive functionality $\mathcal{F}_{\mathsf{cOT}}$ for committing oblivious transfer.

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 $y \in \{0,1\}^{n_2}$, then P_2 computes its effective input by sampling a random $y' \in \{0,1\}^m$ such that $y = My'$.

The original construction by Lindell and Pinkas [25] has $m = \max\{4n_2, 8\rho\}$. shelat and Shen [35] improved this to $m = n_2 + O(\rho + \log(n_2))$. Lindell and Riva [29] 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 Our Protocol

3.1 Protocol Overview

We describe in more detail the intuition behind the changes we introduce. This description is not complete, but only illustrates the main differences from prior work. Full details of our protocol are given in §3.2.

In our protocol, the two parties first run ρ instances of OT, where in the jth instance P_1 sends a random key key_j and a random seed seed_j, while P_2 chooses whether to learn key_j (thereby choosing to make the jth garbled circuit an evaluation circuit) or seed_j (thereby choosing to make the jth garbled circuit a check circuit). The protocol is designed such that key_j can be used to recover the input-wire labels associated with P_1's input in the jth garbled circuit, 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. [1]. However, we differ in how we recover P_1's input if P_1 is caught cheating, and in how we ensure input consistency for P_1's input.

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 circuits such that if P_2 learns *both* labels for some output wire (in different garbled circuits) then P_2 can recover the trapdoor and thus learn P_1's input. In slightly more detail, input recovery consists of the following high-level 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 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 locally computes (and outputs) $f(x, y)$, or (2) outputs the (unique) output of the evaluated garbled circuits, which is $f(x, y)$.

In Afshar et al. [1], the above is done using ElGamal encryption and efficient zero-knowledge checks to enforce input consistency. However, this approach requires $O(\rho \cdot (n_1 + n_3))$ 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 M_0, M_1, with P_1 receiving M_x. Then, for the jth garbled circuit, P_1 "commits" to x by computing $R_{j,x} := \mathrm{PRF}_{\mathsf{seed}_j}(\text{"}R\text{"}) \oplus M_x$ and sending to P_2 an encryption of $R_{j,x}$ under key_j. Note that P_1 cannot "commit" to $1 - x$ unless P_1 can guess M_{1-x}. Also, if P_1 is honest then x remains hidden from P_2 because P_2 knows either key_j or seed_j for each j, but not both. The value seed_j for any evaluation circuit j serves as a trapdoor since, in conjunction with the value key_j that P_2 already has, it allows P_2 to learn M_x (and hence determine 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. First, P_1 chooses random $\varDelta, \varDelta_0, \varDelta_1$ such that $\varDelta = \varDelta_0 \oplus \varDelta_1$. Then, for all j it encrypts $\varDelta_0$ using $Z_{j,0}$ and encrypts $\varDelta_1$ using $Z_{j,1}$, where $Z_{j,0}, Z_{j,1}$ are the two output-wire labels of the jth garbled circuit. It sends all these encryptions to P_2. Thus, if P_2 learns $Z_{j_1,0}$ for some j_1 it can recover $\varDelta_0$, and if it learns $Z_{j_2,1}$ for some j_2 it can recover $\varDelta_1$. If it learns *both* output-wire labels, it can then of course recover $\varDelta$.

P_1 and P_2 then run a protocol that guarantees that if P_2 knows $\varDelta$ it recovers seed_j, and otherwise it learns nothing. This is done as follows. P_2 sets $\varOmega := \varDelta$ if it learned $\varDelta$, and sets $\varOmega := 1$ otherwise. P_2 then computes $(h, g_1, h_1) := (g^\omega, g^r, h^r \varOmega)$, for random ω and r, and sends (h, g_1, h_1) to P_1. Then, for each index j, party P_1 computes $C_j := g^{s_j} h^{t_j}$ and $D_j := g_1^{s_j} (h_1/\varDelta)^{t_j}$ for random s_j, t_j, and sends C_j along with an encryption of seed_j under D_j. Note that if $\varOmega = \varDelta$, then $C_j^r = D_j$ and thus P_2 can recover seed_j, whereas if $\varOmega \neq \varDelta$ then P_2 learns nothing (in an information-theoretic sense).

Notation	Meaning
$\mathcal{E}$	evaluation set
E	ρ-probe matrix
GC_j	jth garbled circuit
$\{A_{j,i,b}\}_b$	ith input-wire labels for P_1 in GC_j
$\{B_{j,i,b}\}_b$	ith input-wire labels for P_2 in GC_j
$\{Z_{j,i,b}\}_b$	ith output-wire labels in GC_j
$\{T_{j,i,b}\}_b$	ith output-mapping table for GC_j
$\{R_{j,i,b}\}_{j,b}$	commitments for the ith bit of P_1's input
C_j, D_j	input-recovery elements

Table 2. Notation used in our protocol.

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 process, 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 with high probability without leaking any information.

Input consistency. As discussed in §1.1, prior schemes enforce that P_1 uses the same input x for all garbled circuits and also for the input-recovery sub-protocol. However, we observe that this is not necessary. Instead, it suffices to ensure that P_1 uses the same input in the input-recovery sub-protocol and *at least one* of the evaluated garbled circuit. Even if P_1 cheats by using different inputs in two different evaluated garbled circuits, P_2 still obtains 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 ensures that P_2 learns x and so can compute the correct output.

We ensure the above weaker notion of consistency by integrating the consistency check with the cut-and-choose process as follows. Recall that in our input-recovery scheme, P_1 sends to P_2 a "commitment" $R_{j,x} := \mathrm{PRF}_{\mathsf{seed}_j}(\text{“}R\text{”}) \oplus M_x$ for each index j. After these commitments are sent, we now have P_2 reveal $M_0 \oplus M_1$ to P_1 (we use committing OT for this purpose), so P_1 learns both M_0 and M_1. P_1 then computes and sends (in a randomly permuted order) $\mathsf{Com}(R_{j,0}, A_{j,0})$ and $\mathsf{Com}(R_{j,1}, A_{j,1})$, where $A_{j,0}, A_{j,1}$ are P_1's input-wire labels in the jth garbled circuit and the commitments are generated using randomness derived from seed_j. P_1 also sends $\mathsf{Enc}_{\mathsf{key}_j}(\mathsf{Decom}(\mathsf{Com}(R_{j,x}, A_{j,x})))$. Note that (1) if P_2 chose j as a check circuit then it can check correctness of the commitment pair, since everything is computed from seed_j, and (2) if P_2 chose j as an evaluation circuit then it can open the appropriate commitment to recover $R_{j,x}$, and check that this matches the value sent before.

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 $f : \{0,1\}^{n_1} \times \{0,1\}^{n_2} \to \{0,1\}^{n_3}$;
Circuit $f' : \{0,1\}^{n_1} \times \{0,1\}^{m} \to \{0,1\}^{n_3}$ such that $f'(x, y') = f(x, Ey')$;
Prime q with $|q| = \mathsf{poly}(\kappa)$.

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 $\mathcal{E} \in_R \{0,1\}^\rho$, 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 $\{M_{i,b}\}_{i\in[n_1], b\in\{0,1\}}$ to $\mathcal{F}_{\mathsf{cOT}}$. P_1 sends x to $\mathcal{F}_{\mathsf{cOT}}$ and receives $\{M_{i,x[i]}\}_{i\in[n_1]}$. For $j \in [\rho], i \in [n_1]$, P_1 computes $R_{j,i,x[i]} := \mathrm{PRF}_{\mathsf{seed}_j}(i, \text{“}R\text{”}) \oplus M_{i,x[i]}$, 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 $\{M_{i,0}, M_{i,1}\}_{i\in[n_1]}$ 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 $R_{j,i,1-x[i]} := R_{j,i,x[i]} \oplus M_{i,0} \oplus M_{i,1}$, $\{A_{j,i,b} := \mathrm{PRF}_{\mathsf{seed}_j}(i, b, \text{“}A\text{”})\}_{b\in\{0,1\}}$, and $\{(\mathsf{c}^R_{j,i,b}, \mathsf{d}^R_{j,i,b}) \leftarrow \mathsf{Com}(R_{j,i,b}, A_{j,i,b})\}_{b\in\{0,1\}}$ 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 labels $\{Z_{j,i}\}_i$. P_2 checks 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 $\Omega := 1$.

Fig. 3. The full description of our malicious 2PC protocol, part 1.

Protocol Π_{2pc} continued

Protocol:

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_j^r 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 $R_{j,i,x[i]} = \mathrm{PRF}_{\mathsf{seed}'_j}(i, \text{“}R\text{”}) \oplus M_{i,0}$, P_2 sets $x_j[i] := 0$; if $R_{j,i,x[i]} = \mathrm{PRF}_{\mathsf{seed}'_j}(i, \text{“}R\text{”}) \oplus M_{i,1}$, 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 all the following checks hold for all $j \notin \mathcal{E}$, then P_2 outputs z; otherwise, P_2 aborts.
   (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_j^T 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 $i \in [n_1], b \in \{0,1\}$, $\mathsf{c}_{j,i,b}^R$ is correctly computed using seed_j, $A_{j,i,b}$, and $R_{j,i,b}$ (which are themselves computed from seed_j).

Fig. 4. The full description of our malicious 2PC protocol, part 2.

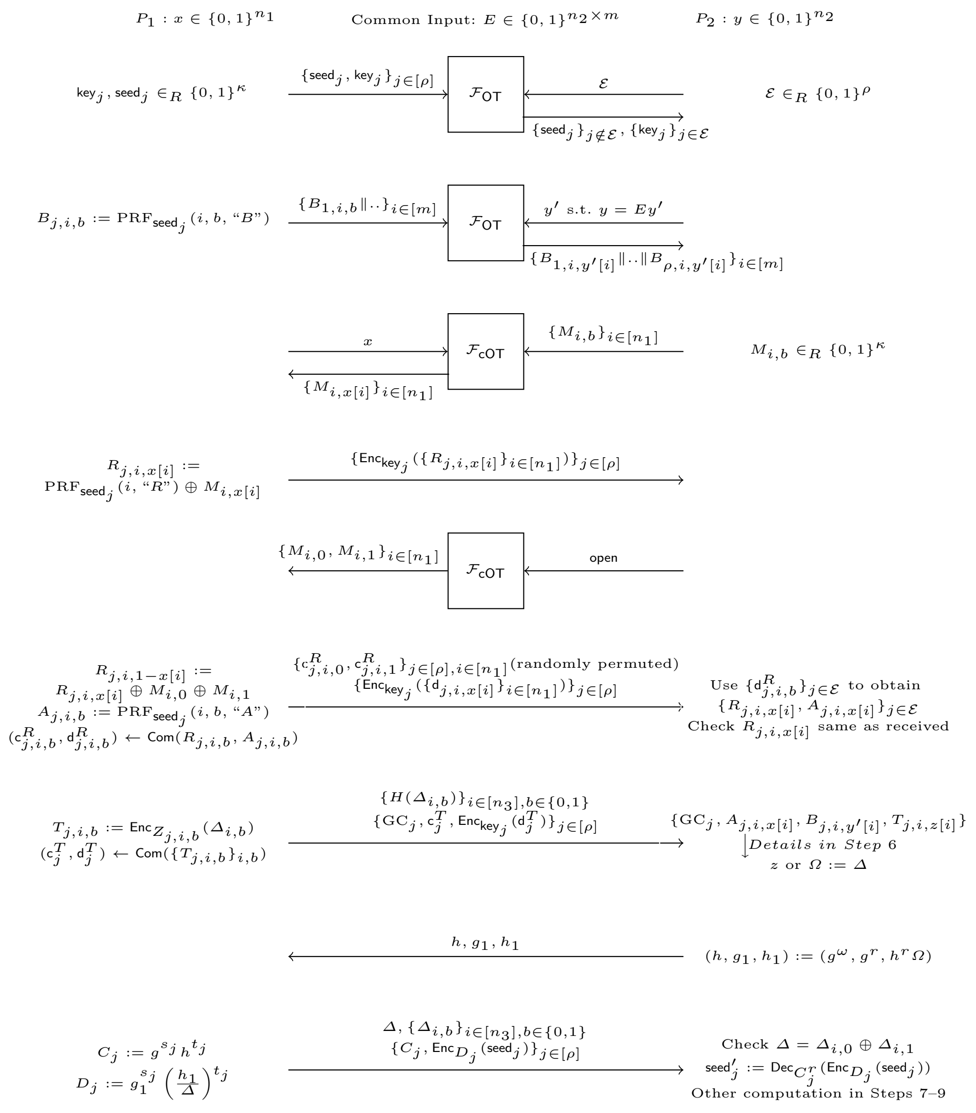
3.2 Protocol Details and Proof of Security

We present the full details of our protocol in Figures 3 and 4. To aid in understanding the protocol, we also present a graphical depiction in Figure 5. We summarize some important notations in Table 2 for reference.

Our protocol, including the optimizations detailed in §4, requires a total of $O(\rho\cdot(n_1+n_2+n_3+|C|))$ symmetric-key operations and $O(\rho+\kappa)$ group operations. Most of the symmetric-key operations, including circuit garbling and computing the PRFs, can be accelerated using hardware AES instructions.

Theorem 1. *Let* Com *be a computationally hiding/binding commitment scheme, let the garbling scheme satisfy authenticity, privacy, and obliviousness (cf. [6]), let* H *be collision-resistant, and assume the decisional Diffie-Hellman assumption holds. Then the protocol in Figures 3 and 4 securely computes* f *in the* $(\mathcal{F}_{\mathsf{OT}}, \mathcal{F}_{\mathsf{cOT}})$*-hybrid model with security* $2^{-\rho} + \mathsf{negl}(\kappa)$*.*

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



$P_1 : x \in \{0,1\}^{n_1}$
Common Input: $E \in \{0,1\}^{n_2 \times m}$
$P_2 : y \in \{0,1\}^{n_2}$
$\mathsf{key}_j, \mathsf{seed}_j \in_R \{0,1\}^\kappa$
$\{\mathsf{seed}_j, \mathsf{key}_j\}_{j\in[\rho]}$
$\mathcal{F}_{\mathsf{OT}}$
$\mathcal{E}$
$\{\mathsf{seed}_j\}_{j\notin\mathcal{E}}, \{\mathsf{key}_j\}_{j\in\mathcal{E}}$
$\mathcal{E} \in_R \{0,1\}^\rho$
$B_{j,i,b} := \mathrm{PRF}_{\mathsf{seed}_j}(i, b, \text{“}B\text{”})$
$\{B_{1,i,b}\|..\}_{i\in[m]}$
$\mathcal{F}_{\mathsf{OT}}$
y' s.t. $y = Ey'$
$\{B_{1,i,y'[i]}\|..\|B_{\rho,i,y'[i]}\}_{i\in[m]}$
x
$\mathcal{F}_{\mathsf{cOT}}$
$\{M_{i,b}\}_{i\in[n_1]}$
$M_{i,b} \in_R \{0,1\}^\kappa$
$\{M_{i,x[i]}\}_{i\in[n_1]}$
$R_{j,i,x[i]} :=$
$\mathrm{PRF}_{\mathsf{seed}_j}(i, \text{“}R\text{”}) \oplus M_{i,x[i]}$
$\{\mathsf{Enc}_{\mathsf{key}_j}(\{R_{j,i,x[i]}\}_{i\in[n_1]})\}_{j\in[\rho]}$
$\{M_{i,0}, M_{i,1}\}_{i\in[n_1]}$
$\mathcal{F}_{\mathsf{cOT}}$
open
$R_{j,i,1-x[i]} :=$
$R_{j,i,x[i]} \oplus M_{i,0} \oplus M_{i,1}$
$A_{j,i,b} := \mathrm{PRF}_{\mathsf{seed}_j}(i, b, \text{“}A\text{”})$
$(\mathsf{c}^R_{j,i,b}, \mathsf{d}^R_{j,i,b}) \leftarrow \mathsf{Com}(R_{j,i,b}, A_{j,i,b})$
$\{\mathsf{c}^R_{j,i,0}, \mathsf{c}^R_{j,i,1}\}_{j\in[\rho],i\in[n_1]}$(randomly permuted)
$\{\mathsf{Enc}_{\mathsf{key}_j}(\{\mathsf{d}_{j,i,x[i]}\}_{i\in[n_1]})\}_{j\in[\rho]}$
Use $\{\mathsf{d}^R_{j,i,b}\}_{j\in\mathcal{E}}$ to obtain
$\{R_{j,i,x[i]}, A_{j,i,x[i]}\}_{j\in\mathcal{E}}$
Check $R_{j,i,x[i]}$ same as received
$T_{j,i,b} := \mathsf{Enc}_{Z_{j,i,b}}(\Delta_{i,b})$
$(\mathsf{c}^T_j, \mathsf{d}^T_j) \leftarrow \mathsf{Com}(\{T_{j,i,b}\}_{i,b})$
$\{H(\Delta_{i,b})\}_{i\in[n_3],b\in\{0,1\}}$
$\{\mathrm{GC}_j, \mathsf{c}^T_j, \mathsf{Enc}_{\mathsf{key}_j}(\mathsf{d}^T_j)\}_{j\in[\rho]}$
$\{\mathrm{GC}_j, A_{j,i,x[i]}, B_{j,i,y'[i]}, T_{j,i,z[i]}\}$
Details in Step 6
z or $\Omega := \Delta$
h, g_1, h_1
$(h, g_1, h_1) := (g^\omega, g^r, h^r\Omega)$
$C_j := g^{s_j} h^{t_j}$
$D_j := g_1^{s_j}\left(\frac{h_1}{\Delta}\right)^{t_j}$
$\Delta, \{\Delta_{i,b}\}_{i\in[n_3],b\in\{0,1\}}$
$\{C_j, \mathsf{Enc}_{D_j}(\mathsf{seed}_j)\}_{j\in[\rho]}$
Check $\Delta = \Delta_{i,0} \oplus \Delta_{i,1}$
$\mathsf{seed}'_j := \mathsf{Dec}_{C^r_j}(\mathsf{Enc}_{D_j}(\mathsf{seed}_j))$
Other computation in Steps 7–9

Fig. 5. Graphical depiction of our protocol.

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. We rely on the following lemma, which we prove in §3.3.

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

Given this, the simulator essentially acts as an honest P_2 using input 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 $\mathcal{S}$ that runs $\mathcal{A}$ as a subroutine and plays the role of P_1 in the ideal world involving an ideal functionality $\mathcal{F}$ evaluating f. $\mathcal{S}$ is defined as follows.

1–2 $\mathcal{S}$ interacts with $\mathcal{A}$, acting as an honest P_2 using input 0.

3 $\mathcal{S}$ obtains the input x that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{cOT}}$. It forwards 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 joint distribution over the outputs of $\mathcal{A}$ and the honest P_2 in the real world is indistinguishable from their joint distribution in the ideal world.

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

$\mathbf{H_2}$. $\mathcal{S}$ now extracts the input x that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{cOT}}$ 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.

There are two ways $\mathcal{A}$ would cheat here, and we address each in turn. For simplicity, we let $I \subset [m]$ denote the set of indices i such that $B_{j,i,b}$ is not correctly computed.

1. $\mathcal{A}$ launches a selective-failure attack with $|I| < \rho$. Lemma 1 ensures (in $\mathbf{H_1}$) that P_2 either aborts or learns $f(x,y)$ with probability at least $1 - 2^{-\rho}$. In $\mathbf{H_2}$, note that P_2 either aborts or learns $f(x,y)$ with probability 1. Further, since fewer than ρ wires are corrupted, the probability of an abort due to the selective-failure attack is exactly the same in both hybrids. Therefore the statistical distance between $\mathbf{H_1}$ and $\mathbf{H_2}$ is at most $2^{-\rho}$.
2. $\mathcal{A}$ launches a selective-failure attack with $|I| \geq \rho$. By the security of the ρ-probe matrix [29], $\mathcal{S}$ aborts in $\mathbf{H_1}$ with probability at least $1 - 2^{-\rho}$. (If $\mathcal{A}$ cheats elsewhere, the probability of abort can only increase.) But in $\mathbf{H_2}$ in this case P_2 aborts with probability 1, and so there is at most a $2^{-\rho}$ difference between $\mathbf{H_1}$ and $\mathbf{H_2}$.

$\mathbf{H_3}$. Same as $\mathbf{H_2}$, except $\mathcal{S}$ uses $y := 0$ throughout of protocol and 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. Computationally indistinguishability of $\mathbf{H_2}$ and $\mathbf{H_3}$ follows.

As $\mathbf{H_3}$ is the ideal-world execution, the proof is complete. □

Malicious P_2. Here, we need to simulate the correct output $f(x, y)$ that P_2 learns. Rather than simulate the 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 input 0, which lets P_2 learn the output-wire labels for $f(0, y)$ when evaluating the garbled circuits. The simulator then "tweaks" the output mapping tables $\{T_{j,i,b}\}$ to ensure that P_2 reconstructs the "correct" output $f(x, y)$.

We now proceed to the formal details. Let $\mathcal{A}$ be an adversary 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 $\mathcal{S}$ acts as an honest P_1 with input $x := 0$. That is, $\mathcal{S}$ receives $\{M_{i,b}\}$ labels and sends $\{\mathsf{Enc}_{\mathsf{key}_j}(\{R_{j,i,0}\}_{i\in[n_1]})\}_{j\in[\rho]}$ to $\mathcal{A}$. If $\mathcal{A}$ send abort to $\mathcal{F}_{\mathsf{cOT}}$, $\mathcal{S}$ aborts, outputting whatever $\mathcal{A}$ outputs.
4 $\mathcal{S}$ acts as an honest P_1 with input $x := 0$. That is, $\mathcal{S}$ sends $\{(\mathsf{c}^{R}_{j,i,0}, \mathsf{c}^{R}_{j,i,1})\}$ (in random permuted order) and $\mathsf{Enc}_{\mathsf{key}_j}(\{\mathsf{d}^{R}_{j,i,0}\}_{i\in[n_1]})$ to $\mathcal{A}$.
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 $T_{j,i,b} := \mathsf{Enc}_{Z_{j,i,b}}(\Delta_{i,1-b})$ if $z[i] \neq z'[i]$.
6–7 $\mathcal{S}$ acts as an honest P_1.

We now show that the joint distribution over the outputs of the honest P_1 and $\mathcal{A}$ in the real world is indistinguishable from their joint distribution in the ideal world.

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

$\mathbf{H_2}$. $\mathcal{S}$ 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 "tweaks" $\{T_{j,i,b}\}$ as is done by the simulator using knowledge of z.

$\mathbf{H_1}$ and $\mathbf{H_2}$ are the same except:

1. In $\mathbf{H_1}$, P_2 learns $\{R_{j,i,x[i]}\}_{j\in\mathcal{E}}$, while P_2 learns $\{R_{j,i,0}\}_{j\in\mathcal{E}}$ in $\mathbf{H_2}$. Note that these values are computed such that $R_{j,i,b} := \mathrm{PRF}_{\mathsf{seed}_j}(i, \text{“}R\text{”}) \oplus M_{i,b}$. Since P_2 does not know any $\{\mathsf{seed}_j\}_{j\in\mathcal{E}}$, $\mathrm{PRF}_{\mathsf{seed}_j}(i, \text{“}R\text{”})$ looks random to P_2. Because only one of $\{R_{j,i,b}\}_{b\in\{0,1\}}$ is given in both $\mathbf{H}$s, $R_{j,i,x[i]}$ in $\mathbf{H_1}$ and $R_{j,i,0}$ in $\mathbf{H_2}$ are uniformly random to P_2.

2. In $\mathbf{H_1}$, party 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 $z[i] \neq z'[i]$ then 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 any information about the other output-wire label due to the authenticity property 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 execution, the proof is complete.

3.3 Proving Lemma 1

We now prove a series of lemmas toward proving Lemma 1. We begin with a definition of what it means for an index $j \in [\rho]$ to be "good."

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

We first show that P_2 is able to recover the correct output-wire labels for a good index.

Lemma 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 good and P_2 does not abort, then with all but negligible probability P_2 learns output labels $Z_{j,i,z[i]}$ with $z = f(x, y)$.*

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

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

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

We next show that P_2 can recover x if P_1 tries to cheat on a good index.

Lemma 3. *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 good 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 good, 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 $R_{j,i,x[i]} = \mathrm{PRF}_{\mathsf{seed}_j}(i, \text{“}R\text{”}) \oplus M_{i,x[i]}$ and the seed_j used in the computation. Further P_2 has $M_{i,0}, M_{i,1}$. 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. *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 negligible probability.*

Proof. Our proof is by contradiction. Assume that P_2 does not abort and recovers some $x' \neq x$ for some $j \in \mathcal{E}$. Let i be an index at which $x'[i] \neq x[i]$; we will show in the following that $\mathcal{A}$ will have to complete some task that is information-theoretically infeasible, and thus a contradiction.

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

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

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

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

Although $\mathcal{A}$ receives $M_{i,x[i]}$ in Step 3, $M_{i,1-x[i]}$ remains completely random before $\mathcal{A}$ sends $R_{j,i,x[i]}$. Further, $\mathcal{A}$ receives $M_{i,b}$ only after sending $R_{j,i,x[i]}$. Therefore, the value of $R_{j,i,x[i]} \oplus M_{i,1-x[i]}$ 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, we are ready to prove Lemma 1, namely, that P_2 either aborts or learns $f(x, y)$, regardless of P_1's behavior.

Lemma 1. *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 $1 - (2^{-\rho} + \mathsf{negl}(\kappa))$, P_2 either aborts or learns $f(x, y)$.*

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 2). However, it is still possible that P_2 learns more than one valid z. If this is the case, P_2 learns Δ. Lemma 3 ensures that P_2 obtains x; Lemma 4 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.

3.4 Universal Composability

Note that in our proof of security, the simulators do not rewind in any of the steps. Similarly, none of the simulators in the hybrid arguments need any rewinding. Therefore, if we instantiate all the functionalities using UC-secure variants then the resulting 2PC protocol is UC-secure.

4 Optimizations

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

4.1 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 [22], takes a significant amount of time. The naive XOR-tree [25] 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 [29] across all circuits. Further, it also introduces a large memory overhead: 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 concretely reduce the hidden constant factor in the ρ-probe matrix.

A general transformation to a sparse matrix. We first asymptotically 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 $51\times$ 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 [25,29] 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 is 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 [29]. 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 in prior work [25].

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 $|S|$.

	k						
Scheme	40	65	80	103	143	229	520
LP07 [25]	6.66	4.1	4	4	4	4	4
sS13 [35]	7.95	5.2	4.5	4.1	3.2	2.4	1.6
This work	5.68	4	3.5	3	2.5	2	1.5

Table 3. Values of c as a function of chunk size k for an ρ-probe matrix with $\rho = 40$.

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

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

where $\mathit{cdf}()$ is the cumulative distribution function for $\mathrm{Bin}(ck - k, \frac{1}{2})$. Now, for each k we can find the smallest c such that $\Pr[A \text{ is bad}] \leq 2^{-\rho}$; we include some results in Table 3. We see that our new ρ-probe matrix achieves smaller c than prior work [25,35]. Note that the number of XORs is $c\rho kn$ and the number of OTs is cn. In our implementation we use $k = 232$ and $c = 2$ to achieve the maximum overall efficiency.

Performance results. See Figure 6 for a comparison between our approach and the best previous scheme [35]. 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 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 for the prior work takes at least 8.59 GB of memory.

4.2 Other Optimizations

Oblivious transfer with hardware acceleration. As observed by Asharov et al. [3], 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 [30]. The use of SSE-based matrix transposition in the OT extension

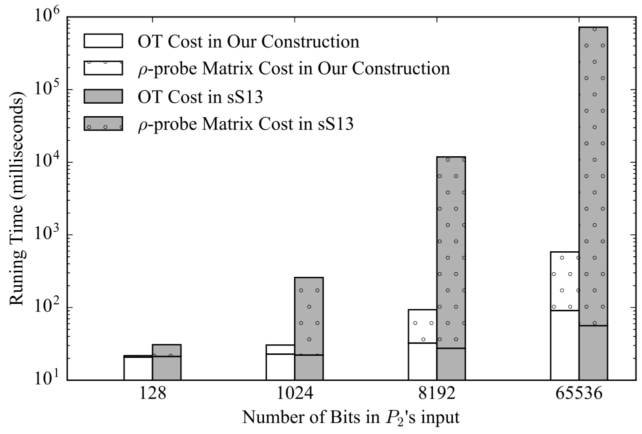


Fig. 6. Comparing the cost of our ρ-probe matrix design with the prior best scheme [35]. When used in a malicious 2PC protocol, computing the ρ-probe matrix needs to be done ρ times, and OT extension needs to process a cn-bit input because of the blowup of the input caused by the ρ-probe matrix.

protocol is also independently studied in a concurrent work by Keller et al. [20] in a 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 §5.1 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 by Asharov et al. [3] 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. [16] 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 §5.4, we are able to evaluate a 4.3 billion-gate circuit without any memory issue, something that offline/online protocols [29] cannot do without using lots of memory or disk I/O.

					localhost			LAN			WAN		
	n_1	n_2	n_3	$\lvert C\rvert$	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.

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
8×10^6 malicious OT extension	4.99 s	5.64 s	25.6 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 for both garbling and sending. The remaining rows give the performance of OT and malicious OT extension.

Pushing computation offline. Although the focus of our work is better efficiency in the absence of pre-processing, it is still worth noting that several steps of our protocol can be pushed offline (i.e., before the parties' inputs are known) when that is an option. Specifically:

1. In addition to the base OTs, 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 the D_j values. During the online phase, P_1 and P_2 only need to perform ρ exponentiations.
2. Garbled circuits can be computed, sent, and checked offline. P_2 can also decommit c_j^T to learn the output translation tables for the evaluation circuits.

5 Implementation and Evaluation

We implemented our protocol in C++ using `RELIC` [2] for group operations, OpenSSL `libssl` for instantiating the hash function, and `libgarble` for garbling [15]. We adopted most of the recent advances in the field [5,39,4,8,29] as well as the optimizations introduced in §4. We instantiate the commitment scheme as $(\text{SHA-1}(x, r), r) \leftarrow \mathsf{Com}(x)$, though when x has sufficient entropy we use SHA-1(x) alone as the commitment.

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 in three different network settings:

	Our Protocol	[29]	[1]
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 [29] were obtained by adapting their implementation to the single-execution setting, using the same hardware as our results. Numbers for [1] are taken from their paper and are for a single execution, not including any I/O time.

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

All numbers are average results of 10 runs. We observed very small variance between multiple executions.

5.1 Subprotocol Performance

Because of the various optimizations mentioned in §4, as well as a carefully engineered implementation, many parts of our system perform better than previously reported implementations. We summarize these results in Table 5.

The garbling speed is 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, our malicious OT extension reports 5.64 seconds for 8 million OTs. Our implementation takes 0.133 seconds for 1024 base OTs. We observe that when two machines are involved, bandwidth is the main bottleneck.

5.2 Overall 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 [29], 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

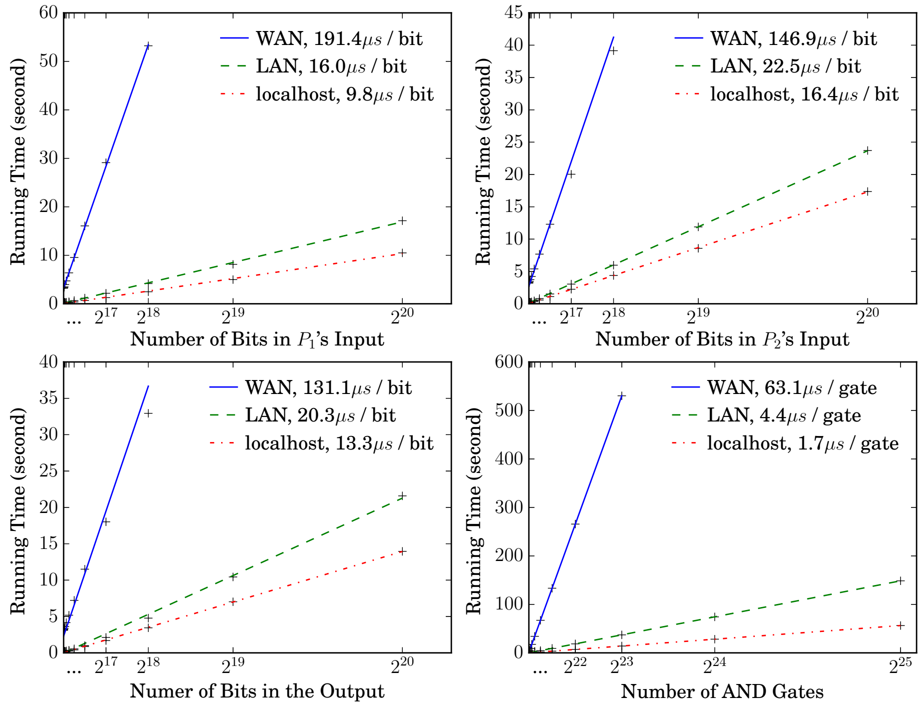


Fig. 7. The performance of our protocol while modifying the input lengths, output length and the circuit size. Input and output lengths are set to 128 bits initially and circuit size is set as 16,384 AND gates. Numbers in the figure show the slope of the lines, namely the cost to process an additional bit or gate.

in parentheses. We tried to compare with the implementation by Rindal and Rosulek [33]; however, their implementation is inherently parallelized, making comparisons difficult. Estimations suggest that their implementation is faster than Lindell and Riva but still less efficient than ours in the single-execution setting.

In Table 6, we compare the performance of our protocol 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. [1]. 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 [29] 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

	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 7. Scalability of our protocol. All numbers are in microseconds per bit or microseconds per 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 8. 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 §5.3; *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.

implementation is about 3× to 4× better than Afshar et al., but still 6× to 26× slower than our LAN results.

5.3 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 7, 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 7, with the same constant cost of the base OTs.

5.4 Additional Examples

Finally, in Table 8 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 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

The authors thank Samuel Ranellucci and Yan Huang for helpful discussions and comments.

References

1. Afshar, A., Mohassel, P., Pinkas, B., Riva, B.: Non-interactive secure computation based on cut-and-choose. In: Advances in Cryptology—Eurocrypt 2014. LNCS, vol. 8441, pp. 387–404. Springer (2014)
2. Aranha, D.F., Gouvêa, C.P.L.: RELIC is an Efficient LIbrary for Cryptography. `https://github.com/relic-toolkit/relic`
3. Asharov, G., Lindell, Y., Schneider, T., Zohner, M.: More efficient oblivious transfer and extensions for faster secure computation. In: 20th ACM Conf. on Computer and Communications Security (CCS). pp. 535–548. ACM Press (2013)
4. Asharov, G., Lindell, Y., Schneider, T., Zohner, M.: More efficient oblivious transfer extensions with security for malicious adversaries. In: Advances in Cryptology—Eurocrypt 2015, Part I. LNCS, vol. 9056, pp. 673–701. Springer (2015)
5. Bellare, M., Hoang, V.T., Keelveedhi, S., Rogaway, P.: Efficient garbling from a fixed-key blockcipher. In: 2013 IEEE Symposium on Security & Privacy. pp. 478–492. IEEE (2013)
6. Bellare, M., Hoang, V.T., Rogaway, P.: Foundations of garbled circuits. In: 19th ACM Conf. on Computer and Communications Security (CCS). pp. 784–796. ACM Press (2012)
7. Brandão, L.T.A.N.: Secure two-party computation with reusable bit-commitments, via a cut-and-choose with forge-and-lose technique. In: Advances in Cryptology—Asiacrypt 2013, Part II. LNCS, vol. 8270, pp. 441–463. Springer (2013)
8. Chou, T., Orlandi, C.: The simplest protocol for oblivious transfer. In: Progress in Cryptology—Latincrypt 2015. LNCS, vol. 9230, pp. 40–58. Springer (2015)
9. Damgård, I., Lauritsen, R., Toft, T.: An empirical study and some improvements of the MiniMac protocol for secure computation. In: 9th Intl. Conf. on Security and Cryptography for Networks (SCN). LNCS, vol. 8642, pp. 398–415. Springer (2014)
10. David, B.M., Nishimaki, R., Ranellucci, S., Tapp, A.: Generalizing efficient multiparty computation. In: 8th Intl. Conference on Information Theoretic Security (ICITS). LNCS, vol. 9063, pp. 15–32. Springer (2015)

11. Frederiksen, T.K., Jakobsen, T.P., Nielsen, J.B.: Faster maliciously secure two-party computation using the GPU. In: 9th Intl. Conf. on Security and Cryptography for Networks (SCN). LNCS, vol. 8642, pp. 358–379. Springer (2014)
12. Frederiksen, T.K., Jakobsen, T.P., Nielsen, J.B., Nordholt, P.S., Orlandi, C.: MiniLEGO: Efficient secure two-party computation from general assumptions. In: Advances in Cryptology—Eurocrypt 2013. LNCS, vol. 7881, pp. 537–556. Springer (2013)
13. Frederiksen, T.K., Nielsen, J.B.: Fast and maliciously secure two-party computation using the GPU. In: 11th Intl. Conference on Applied Cryptography and Network Security (ACNS). LNCS, vol. 7954, pp. 339–356. Springer (2013)
14. Goyal, V., Mohassel, P., Smith, A.: Efficient two party and multi party computation against covert adversaries. In: Advances in Cryptology—Eurocrypt 2008. LNCS, vol. 4965, pp. 289–306. Springer (2008)
15. Groce, A., Ledger, A., Malozemoff, A.J., Yerukhimovich, A.: CompGC: Efficient offline/online semi-honest two-party computation. Cryptology ePrint Archive, Report 2016/458 (2016), `http://eprint.iacr.org/2016/458`
16. Huang, Y., Evans, D., Katz, J., Malka, L.: Faster secure two-party computation using garbled circuits. In: 20th USENIX Security Symposium. USENIX Association (2011)
17. Huang, Y., Katz, J., Evans, D.: Efficient secure two-party computation using symmetric cut-and-choose. In: Advances in Cryptology—Crypto 2013, Part II. LNCS, vol. 8043, pp. 18–35. Springer (2013)
18. Huang, Y., Katz, J., Kolesnikov, V., Kumaresan, R., Malozemoff, A.J.: Amortizing garbled circuits. In: Advances in Cryptology—Crypto 2014, Part II. LNCS, vol. 8617, pp. 458–475. Springer (2014)
19. Jawurek, M., Kerschbaum, F., Orlandi, C.: Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In: 20th ACM Conf. on Computer and Communications Security (CCS). pp. 955–966. ACM Press (2013)
20. Keller, M., Orsini, E., Scholl, P.: MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. pp. 830–842. ACM Press (2016)
21. Kolesnikov, V., Mohassel, P., Rosulek, M.: FleXOR: Flexible garbling for XOR gates that beats free-XOR. In: Advances in Cryptology—Crypto 2014, Part II. LNCS, vol. 8617, pp. 440–457. Springer (2014)
22. Kolesnikov, V., Schneider, T.: Improved garbled circuit: Free XOR gates and applications. In: 35th Intl. Colloquium on Automata, Languages, and Programming (ICALP), Part II. LNCS, vol. 5126, pp. 486–498. Springer (2008)
23. Kreuter, B., Shelat, A., Shen, C.H.: Billion-gate secure computation with malicious adversaries. In: USENIX Security Symposium. pp. 285–300. USENIX Association (2012)
24. Lindell, Y.: Fast cut-and-choose based protocols for malicious and covert adversaries. In: Advances in Cryptology—Crypto 2013, Part II. LNCS, vol. 8043, pp. 1–17. Springer (2013)
25. Lindell, Y., Pinkas, B.: An efficient protocol for secure two-party computation in the presence of malicious adversaries. In: Naor, M. (ed.) Advances in Cryptology—Eurocrypt 2007. LNCS, vol. 4515, pp. 52–78. Springer (2007)
26. Lindell, Y., Pinkas, B.: Secure two-party computation via cut-and-choose oblivious transfer. In: 8th Theory of Cryptography Conference (TCC) 2011. LNCS, vol. 6597, pp. 329–346. Springer (2011)
27. Lindell, Y., Pinkas, B., Smart, N.P.: Implementing two-party computation efficiently with security against malicious adversaries. In: 6th Intl. Conf. on Security and Cryptography for Networks (SCN). LNCS, vol. 5229, pp. 2–20. Springer (2008)

28. Lindell, Y., Riva, B.: Cut-and-choose Yao-based secure computation in the online/offline and batch settings. In: Advances in Cryptology—Crypto 2014, Part II. LNCS, vol. 8617, pp. 476–494. Springer (2014)
29. Lindell, Y., Riva, B.: Blazing fast 2PC in the offline/online setting with security for malicious adversaries. In: 22nd ACM Conf. on Computer and Communications Security (CCS). pp. 579–590. ACM Press (2015)
30. 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
31. Nielsen, J.B., Nordholt, P.S., Orlandi, C., Burra, S.S.: A new approach to practical active-secure two-party computation. In: Advances in Cryptology—Crypto 2012. LNCS, vol. 7417, pp. 681–700. Springer (2012)
32. Pinkas, B., Schneider, T., Smart, N.P., Williams, S.C.: Secure two-party computation is practical. In: Advances in Cryptology—Asiacrypt 2009. LNCS, vol. 5912, pp. 250–267. Springer (Dec 2009)
33. Rindal, P., Rosulek, M.: Faster malicious 2-party secure computation with online/offline dual execution. In: USENIX Security Symposium. pp. 297–314. USENIX Association (2016)
34. Shelat, A., Shen, C.H.: Two-output secure computation with malicious adversaries. In: Advances in Cryptology—Eurocrypt 2011. LNCS, vol. 6632, pp. 386–405. Springer (2011)
35. Shelat, A., Shen, C.H.: Fast two-party secure computation with minimal assumptions. In: 20th ACM Conf. on Computer and Communications Security (CCS). pp. 523–534. ACM Press (2013)
36. Wang, X., Malozemoff, A.J., Katz, J.: EMP-toolkit: Efficient multiparty computation toolkit. `https://github.com/emp-toolkit`
37. Wang, X., Ranellucci, S., Malozemoff, A.J., Katz, J.: Faster secure two-party computation in the single-execution setting. Cryptology ePrint Archive, Report 2016/762 (2016), `http://eprint.iacr.org/2016/762`
38. Yao, A.C.C.: Protocols for secure computations. In: 23rd Annual Symposium on Foundations of Computer Science (FOCS). pp. 160–164. IEEE (1982)
39. Zahur, S., Rosulek, M., Evans, D.: Two halves make a whole—reducing data transfer in garbled circuits using half gates. In: Advances in Cryptology—Eurocrypt 2015, Part II. LNCS, vol. 9057, pp. 220–250. Springer (2015)