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*  
Galois  
amaloz@galois.com

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 $\rho$ garbled circuits, where $\rho$ 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 $3\times$ slower than a semi-honest execution of the same functionality, and $22\times$ faster than the best prior work in the single-execution setting. On a local-area network, our protocol requires around $20 \mu s$ to process each input/output bit and around $4 \mu s$ 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

*Work done while at University of Maryland.
Table 1: Two-party computation of AES, with security against malicious adversaries, in the single-execution setting. The statistical security parameter is $\rho$. 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.

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 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 $1000\times$ for a 65,536-bit input.
- Our implementation is open sourced as a part of the EMP-toolkit\footnote{https://github.com/emp-toolkit}, 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 one way of instantiating this approach \[Lin13\] all the garbled circuits use output-wire labels with the same XOR difference \(\Delta\). 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 \(\Delta\). (The protocol is set up so that \(\Delta\) is not revealed by the check circuits.) The parties then run a second 2PC protocol in which \(P_2\) learns \(x\) if it knows \(\Delta\); 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. 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 \(\omega\) known only to \(P_1\). Note that if \(P_2\) learns \(\omega\) then it can decrypt everything and thus learn \(x\). Now, for each output wire \(P_1\) secret shares \(\omega\) as \(\omega_0\) and \(\omega_1\) 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 \(\omega\). 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 \(\rho\) 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 \(\rho\) 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 \(\rho\). However, this can be improved by using a \(\rho\)-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 \(\rho\)-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.

---

2While 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 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 $\kappa$ be the security parameter, and let $\rho$ be the statistical security parameter. For bitstring $x$, let $x[i]$ denote the $i$th 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 $a \in S$ to denote choosing a uniform value from set $S$. Let $[n] = \{1, \ldots, n\}$. We use the notation $(c, d) \leftarrow \text{Com}(x)$ 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 $\mathcal{F}_{\text{OT}}$ and $\mathcal{F}_{\text{cOT}}$. $\mathcal{F}_{\text{cOT}}$ is the weaker flavor of committing OT that 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.

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].

$\rho$-probe matrix. A $\rho$-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 $i$th row of $M$) is at least $\rho$. 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 $\rho$-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 $y := My'$.

---

3This does not include base OTs and GPU initialization.
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 $\rho$ instances of oblivious transfer (OT), where in instance $j$ $P_1$ sends a random key $\text{key}_j$ and random seed $\text{seed}_j$, while $P_2$ chooses whether to learn $\text{key}_j$ (thereby putting $j$ in the evaluation set) or $\text{seed}_j$ (thereby putting $j$ in the check set). The protocol is designed such that $\text{key}_j$ can be used to recover the input-wire labels associated with $P_1$’s input, whereas $\text{seed}_j$ can be used to recover all the randomness used to generate the $j$th 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 $R_{j, x} := \text{PRF}_{\text{seed}}(\text{"R"}) \oplus X_x$ and sending an encryption of $R_{j, x}$ under $\text{key}_j$ 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 $\text{key}_j$.
or seed\textsubscript{j} for each j, but not both. The trapdoor to recover x is seed\textsubscript{j} for any evaluation circuit \textit{j}, which (in conjunction with key\textsubscript{j} that P\textsubscript{2} already has) allows P\textsubscript{2} to recompute X\textsubscript{x} (and hence x).

The next step is to devise a way for P\textsubscript{1} to recover seed\textsubscript{j}, if it learns \textit{inconsistent} output-wire labels in two different evaluation circuits. We do this as follows. Let Z\textsubscript{j,0}, Z\textsubscript{j,1} be the two output-wire labels of the jth circuit. P\textsubscript{1} chooses some random value Δ and secret shares this value as Δ\textsubscript{0}, Δ\textsubscript{1} such that Δ = Δ\textsubscript{0} ⊕ Δ\textsubscript{1}. Next, it encrypts each share under the appropriate output-wire label and sends these encryptions to P\textsubscript{2}. Thus, if P\textsubscript{2} learns Z\textsubscript{j,0} it can recover Δ\textsubscript{0} and if it learns Z\textsubscript{j,1} it can recover Δ\textsubscript{1}. If it learns \textit{both} output-wire labels, it can then of course recover Δ.

P\textsubscript{1} and P\textsubscript{2} then run a protocol which guarantees that if P\textsubscript{2} knows Δ then it recovers seed\textsubscript{j}, and otherwise it learns nothing. This is done as follows. P\textsubscript{2} sets Ω = Δ if it learned Δ, and sets Ω := 1 otherwise. P\textsubscript{2} then computes (h, g\textsubscript{1}, g\textsubscript{2}) := (g\textsuperscript{w}, g\textsuperscript{r}, h\textsuperscript{Ω}), for random w and r, and sends (h, g\textsubscript{1}, g\textsubscript{2}) to P\textsubscript{1}. Then, for each index j, party P\textsubscript{1} then computes C\textsubscript{j} := g\textsuperscript{s\textsubscript{j}}h\textsuperscript{t\textsubscript{j}} and D\textsubscript{j} := g\textsuperscript{x\textsubscript{j}}(h/Δ\textsuperscript{t\textsubscript{j}}), for random s\textsubscript{j} and t\textsubscript{j}, and sends C\textsubscript{j} along with an encryption of seed\textsubscript{j} under D\textsubscript{j}. Note that if Ω = Δ, then C\textsubscript{j} = D\textsubscript{j} and thus P\textsubscript{2} can recover seed\textsubscript{j}, whereas if Ω ≠ Δ then P\textsubscript{2} learns nothing.

Of course, the protocol as described does not account for the fact that P\textsubscript{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\textsubscript{1} tries to cheat then P\textsubscript{2} either aborts (due to detected cheating) or learns P\textsubscript{1}’s input.

**Input consistency.** As discussed in §1.1, prior schemes enforce that P\textsubscript{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\textsubscript{1} uses the same input in the input-recovery mechanism and at least one evaluated garbled circuit. Even if P\textsubscript{1} cheats by using different inputs in two different evaluated garbled circuits, P\textsubscript{2} can always obtain the correct output: if P\textsubscript{2} learns only one output then this is the correct output; if P\textsubscript{2} learns multiple outputs, then the input-recovery procedure helps P\textsubscript{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\textsubscript{1} sends to P\textsubscript{2} “commitments” R\textsubscript{j,0} := PRF\textsubscript{seed\textsubscript{j}}("R") ⊕ X\textsubscript{x} for each index j. After these values are sent, P\textsubscript{2} sends X\textsubscript{0} ⊕ X\textsubscript{1} to P\textsubscript{1}, allowing P\textsubscript{1} to learn both X\textsubscript{0} and X\textsubscript{1}. P\textsubscript{1} then sends a random permutation of Com(A\textsubscript{j,0}, PRF\textsubscript{seed\textsubscript{j}}("R") ⊕ X\textsubscript{0}) and Com(A\textsubscript{j,1}, PRF\textsubscript{seed\textsubscript{j}}("R") ⊕ X\textsubscript{1}), where A\textsubscript{j,0}, A\textsubscript{j,1} are P\textsubscript{1}’s input-wire labels to the jth garbled circuit. P\textsubscript{1} also sends Enc\textsubscript{key\textsubscript{j}}(Decom(Com(A\textsubscript{j,x}, R\textsubscript{j,x}))). Note that (1) if P\textsubscript{2} chooses j as a check circuit then it can check the correctness of the commitment pair, since everything is computed from seed\textsubscript{j}, and (2) if P\textsubscript{2} choose j as an evaluation circuit then it can recover P\textsubscript{1}’s input-wire label A\textsubscript{j,x} and check if the X\textsubscript{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.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>cut-and-choose set</td>
</tr>
<tr>
<td>E</td>
<td>p-probe matrix</td>
</tr>
<tr>
<td>GC\textsubscript{j}</td>
<td>jth garbled circuit</td>
</tr>
<tr>
<td>A\textsubscript{j,i,b}</td>
<td>ith input-wire labels for P\textsubscript{1} in GC\textsubscript{j}</td>
</tr>
<tr>
<td>B\textsubscript{j,i,b}</td>
<td>ith input-wire labels for P\textsubscript{2} in GC\textsubscript{j}</td>
</tr>
<tr>
<td>Z\textsubscript{j,i,b}</td>
<td>ith output-wire labels in GC\textsubscript{j}</td>
</tr>
<tr>
<td>T\textsubscript{j,i,b}</td>
<td>ith output mapping table for GC\textsubscript{j}</td>
</tr>
<tr>
<td>R\textsubscript{j,i,b}</td>
<td>P\textsubscript{1}’s input commitment</td>
</tr>
<tr>
<td>C\textsubscript{j}, D\textsubscript{j}</td>
<td>Input recovery elements</td>
</tr>
</tbody>
</table>

Table 2: Notation used in our protocol.
Protocol II\textsubscript{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:** $\rho$-probe matrix $E \in \{0, 1\}^{n_2 \times m}$, where $m = O(n_2)$; circuit $f : \{0, 1\}^{n_1} \times \{0, 1\}^{n_2} \rightarrow \{0, 1\}^{n_3}$; and circuit $f' : \{0, 1\}^{n_1} \times \{0, 1\}^{n_2} \rightarrow \{0, 1\}^{n_3}$ such that $f'(x, y') = f(x, E' y')$.

**Protocol:**

1. $P_1$ picks random $k$-bit strings $\{\text{key}, j, \text{seed}\}_j \in \mathcal{E}$, and sends them to $\mathcal{F}_\text{OT}$. $P_2$ picks $\mathcal{E} \in \{0, 1\}^{k}$ and sends $\mathcal{E}$ to $\mathcal{F}_\text{OT}$ and receives $\{\text{seed}, j\}_j \in \mathcal{E}$ and $\{\text{key}, j\}_j \in \mathcal{E}$.

2. (a) $P_1$ computes $\{R_{j,i,x} := \text{PRF}_\text{seed}(i, b, \ldots) \}_{i \in \{0, 1\}}$ and sends $\{\{R_{j,i,b} \mid i \in \{0, 1\}\} \in \{0, 1\}^\mathcal{E}\}$ to $\mathcal{F}_\text{OT}$. $P_2$ picks random $y' \in \{0, 1\}^k$ such that $y = E' y'$, sends $y'$ to $\mathcal{F}_\text{OT}$, and receives $\{\{B_{j,i,b} \mid i \in \{0, 1\}\} \in \{0, 1\}^\mathcal{E}\}$.

3. $P_2$ sends random labels $\{X_{j,i,b} \mid i \in \{0, 1\}\} \in \mathcal{E}$ to $\mathcal{F}_\text{OT}$. $P_1$ sends $x$ to $\mathcal{F}_\text{OT}$ and receives $\{X_{j,i,b} \mid i \in \{0, 1\}\}$. For $j \in [\rho], i \in [n_1]$, $P_1$ computes $R_{j,i,x} := \text{PRF}_\text{seed}(i, \ldots) \oplus X_{j,i,x}$; and sends $\text{Enc}_{\text{key}, \{X_{j,i,b} \mid i \in \{0, 1\}\}}(P_{j,i,x})$ to $P_2$. $P_2$ sends open to $\mathcal{F}_\text{OT}$, which sends $(X_{j,0,0,1})_i \in \mathcal{E}$ to $P_1$, and for $j \in \mathcal{E}$ uses $\text{key}_j$ to decrypt and learn $R_{j,i,x}$.

4. For $j \in [\rho], i \in [n_1]$, $P_1$ computes $R_{j,i,1-x} := R_{j,i,x} \oplus X_{j,0} \oplus X_{j,1}$, $\{\text{seed}, j \mid i \in \{0, 1\}\}$ using randomness derived from seed, and sends $\{\{R_{j,i,b} \mid i \in \{0, 1\}\} \in \mathcal{E}\}$ in random permuted order and $\text{Enc}_{\text{key}, \{X_{j,i,b} \mid i \in \{0, 1\}\}}(P_{j,i,x})$ to $P_2$. For $j \in \mathcal{E}, i \in [n_1]$, $P_2$ opens $R_{j,i,1-x}^R$ to obtain $R_{j,i,x}$ and $A_{j,i,x}$, and checks that $R_{j,i,x}$ equals the value from Step 3. If any decommitment is invalid or any check fails, $P_2$ aborts.

5. $P_1$ picks random $k$-bit labels $\Delta, \{\text{seed}, j \mid i \in \{0, 1\}\}$, sets $\Delta_{i,b} := \text{seed}_i \oplus \Delta_i \in \mathcal{E}$, and sends $\{H(\Delta_{i,b}) \mid i \in \{0, 1\}\} \in \mathcal{E}$ to $P_2$. For $j \in [\rho], P_1$ computes garbled circuit $G_j$ for function $f$ using $A_{j,i,b}, B_{j,i,b}$ as the input-wire labels and randomness derived from seed, and sends $G_j$ for $\mathcal{E}$, and sends $G_j$ for $\mathcal{E}$, recipients to $\mathcal{F}_\text{OT}$. Let $Z_{j,i,b}$ denote the output-wire labels. $P_1$ computes $\{T_{j,i,b} := \text{Dec}_{\text{seed}, j}(G_{j}(\Delta_{i,b})) \mid i \in \{0, 1\}\}$ and $\{c_{j,b}^{R}, d_{j,b}^{R} \mid i \in \{0, 1\}\}$ using randomness derived from seed, and sends $G_j, c_{j,b}^{R}, d_{j,b}^{R}$, and $\text{Enc}_{\text{key}, \{Z_{j,i,b} \mid i \in \{0, 1\}\}}(P_{j,i,b})$ to $P_2$.

6. For $j \in \mathcal{E}, P_2$ decrypts to learn $d_{j,b}^{R}$ and opens $c_{j,b}^{R}$ to learn $\{T_{j,i,b} \mid b \in \{0, 1\}\}$; if any decommitment is invalid, $P_2$ aborts. $P_2$ evaluates $G_j$ using $\{A_{j,i,b} \mid i \in \{0, 1\}\}$ and $\{B_{j,i,b} \mid i \in \{0, 1\}\}$, and obtains output-wire label $Z_{j,i,b}$. $P_2$ checks the validity of these labels by checking if $H(\text{Dec}_{\text{seed}, j}(T_{j,i,b}))$ matches $H(\Delta_{i,b})$ for some $b \in \{0, 1\}$, and if so sets $z_{j}^R[i] = 1$; else it sets $z_{j}^R[i] = 0$.

7. $P_2$ picks $x, y \in \mathcal{E}$, and sends $(h, g_{1, h_1} := (g^y, g^1, h_1 \cdot \Omega))$ to $P_1$. $P_1$ sends $\Delta$ and $\{\text{seed}, j \mid i \in \{0, 1\}\}$ to $P_2$, who checks that $\Delta = \text{seed}_i \oplus \Delta_i$ 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 $x_j, y_j \in \mathcal{E}$ using randomness derived from seed, computes $C_{j} := g^{x_j} h^{y_j}, D_{j} := g^{x_j} \left( h^{1} \right)^{y_j}$, and sends $C_j$ and $\text{Enc}_{\text{key}, \{Z_{j,i,b} \mid i \in \{0, 1\}\}}(P_{j,i,b})$ to $P_2$. For $j \in \mathcal{E}, P_2$ uses $C_j$ 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 $R_{j,i,x} := \text{PRF}_\text{seed}(i, b, \ldots) \oplus X_{j,0,0}, P_2$ sets $x_j[i] := 0$; if $R_{j,i,x} := \text{PRF}_\text{seed}(i, b, \ldots) \oplus X_{j,0,1}, P_2$ sets $x_j[i] := 1$; and otherwise, $P_2$ sets $x_j[i] := 0$. If no valid $x_j$ is obtained, or more than two different $x_j$, $P_2$ sets $z := \perp$; otherwise $P_2$ sets $z := f(x_j, y)$. If any of the following checks fail for any $j \in \mathcal{E}, P_2$ aborts; otherwise $P_2$ outputs $z$.

(a) For $i \in \{0, 1\}, \text{seed}_{j, i, b} := \text{PRF}_\text{seed}(i, b, \ldots)$.

(b) $G_j$ is computed correctly using $A_{j,i,b} := \text{PRF}_\text{seed}(i, b, \ldots)$ and $B_{j,i,b} := \text{PRF}_\text{seed}(i, b, \ldots)$ as input-wire labels and randomness derived from seed.

(c) Compute $T_{j,i,b} := \text{Dec}_{\text{seed}, j}(G_j \oplus \Delta_{i,b} \cdot \text{seed})$, and check that $c_{j,b}^{R}$ is computed correctly with randomness derived from seed.

(d) The $C_j, \text{Enc}_{\text{key}, \{Z_{j,i,b} \mid i \in \{0, 1\}\}}(P_{j,i,b})$ values in Step 7 are correctly computed, using $\Delta$ and seed.

(e) For $i \in [n_1], b \in \{0, 1\}, c_{j,b}^{R}$ is correctly computed using $\text{seed}_j, A_{j,i,b},$ and $R_{j,i,b}$, which are also computed from seed.

Figure 3: The full description of our malicious 2PC protocol.
Let \( (\mathcal{F}_{\text{OT}}, \mathcal{E}_{\text{OT}}) \)-hybrid model with security \( 2^{-\rho} + \text{neg}(\kappa) \).

**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 \( \mathcal{F}_{\text{OT}} \) 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 \( \mathcal{E}_{\text{OT}} \), and outputs \( f(x, y) \) if no party aborts.

Our protocol, including the optimizations detailed in \( \S 5 \), 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 \((\text{Com, Open})\) 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 \((\mathcal{F}_{\text{OT}}, \mathcal{E}_{\text{OT}})\)-hybrid model with security \( 2^{-\rho} + \text{neg}(\kappa) \).
We now proceed to the formal details. Let \( \mathcal{A} \) be an adversary corrupting \( P_1 \); we construct a simulator \( S \) interacting with an ideal functionality \( F \) evaluating \( f \), defined as follows.

1–2 \( S \) acts as an honest \( P_2 \), using input \( y := 0 \).

3 \( S \) acts as an honest \( P_2 \), and obtains the input \( x \) that \( \mathcal{A} \) sends to \( F_{\text{cOT}} \). It sends \( x \) to \( F \).

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

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

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

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

**H₁.** Same as the hybrid-world protocol, where \( S \) plays the role of an honest \( P_2 \) using the actual input \( y \).

**H₂.** Same as \( H₁ \), except that \( S \) extracts the input \( x \) that \( \mathcal{A} \) sends to \( F_{\text{cOT}} \), uses \( y := 0 \) throughout the protocol, except as explained below, and sends \( x \) to \( F \) if no party aborts. \( S \) also performs the additional checks as described above in Step 9 of the simulator. Moreover, \( S \) sets the value of \( \Omega \) 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 \( I \subseteq [m] \) as the set of indices used in the selective failure attack, that is \( I \) is the set of indices \( i \) 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 \( H₁ \), \( \mathcal{S} \) either aborts or learns \( f(x, y) \) with probability at least \( 1 - 2^{-\rho} \). In \( H₂ \), \( \mathcal{S} \) either aborts or learns \( f(x, y) \) with probability 1. Further, since less than \( \rho \) wires are corrupted, the probability of abort due to the selective-failure attack is exactly the same in both hybrids. Therefore the distribution between \( H₁ \) and \( H₂ \) is different by at most \( 2^{-\rho} \).

2. \( \mathcal{A} \) launches a selective-failure attack with \( |I| \geq \rho \). By the security of the \( \rho \)-probe matrix [LR15], \( \mathcal{S} \) aborts in \( H₁ \) 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 \( H₂ \), \( \mathcal{S} \) aborts with probability 1, therefore there is at most \( 2^{-\rho} \) difference between \( H₁ \) and \( H₂ \).

Thus, in any case the \( H₁ \) and \( H₂ \) can be distinguished by at most \( 2^{-\rho} \) probability.

**H₃.** Same as \( H₂ \), except \( S \) always sets \( \Omega := 1 \) in Step 7.

In \( H₃ \), \( P₂ \) sends \((h, g_1, h_1) := (g^{\omega'}, g^r, g^{\omega \Omega})\), which is indistinguishable from \((g^{\omega'}, g^r, g^{\omega \Omega \Omega})\) by the decisional Diffie-Hellman problem. Thus, the views in \( H₂ \) and \( H₃ \) are computationally indistinguishable.

As \( H₃ \) is the same as the ideal world protocol, the proof is complete.

**Malicious \( P₂ \).** Here, we need to simulate the correct output \( f(x, y) \) towards \( P₂ \). 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 \( P₁ \) with \( x = 0 \), which lets \( P₂ \) 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 \( i \)th bit of \( f(0, y) \) and \( f(x, y) \) are different then \( P₂ \) learns the opposite value.

We now proceed to the formal details. Suppose there exists an adversary \( \mathcal{A} \) corrupting \( P₂ \); we construct a simulator \( S \) as follows.
Consider an adversary \( P \), and denote \( x \) as the input \( P \) sent to \( \mathcal{F}_{OT} \). We now show that the views in the hybrid and ideal worlds are indistinguishable.

**Fact 4.1.** If an index \( j \in [\rho] \) is a good index and \( P \) acts as an honest \( P \) with all but negligible probability.

We now build a series of lemmas towards proving Lemma 4.4, which we use above to prove security for a malicious \( P \).

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

1. The \( B_{j,i,y'[i]} \) values \( A \) sent to \( \mathcal{F}_{OT} \) in Step 2 are computed honestly using \( \text{seed}_j \).
2. The commitments \( \{c_{j,i,b}^R\}_{i \in [n_1], b \in \{0,1\}} \) that \( A \) sent to \( P_2 \) in Step 4 are computed honestly using \( \text{seed}_j \).
3. \( \text{GC}_j \) is computed honestly using \( \{A_{j,i,b}\} \) and \( \{B_{j,i,b}\} \) as the input-wire labels and \( \text{seed}_j \).
4. The values \( C_j \) and \( \text{Enc}_{D_j}(\text{seed}_j) \) are computed honestly using \( \text{seed}_j \) and the \( \Delta \) value sent by \( A \) in Step 7.
5. The commitment \( c_j^T \) is computed honestly using \( \Delta_{i,b} \) and \( \text{seed}_j \).

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 \( A \) corrupting \( P \), and denote \( x \) as the input \( A \) sent to \( \mathcal{F}_{OT} \). 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 \( A \) and honest \( B_{j,i,x[i]} \) from \( F_{\text{OT}} \). 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 \( A \) sent to \( F_{\text{OT}} \). We will show that this can only happen with negligible probability.

Note that if \( j \) is good, then the commitments \( \{c_{j,i,b}^R\} \) are computed correctly. Since \( P_2 \) obtains the 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 \( A \) sent to \( F_{\text{OT}} \).

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 \( c_{j,i,b} \) 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

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

Therefore \( A \) 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 \( A \) corrupting \( P_1 \), and denote \( x \) as the input \( A \) sent to \( F_{\text{OT}} \). 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 \( \text{Enc}_{D_j}(\text{seed}_j) \) are constructed correctly, where \( \text{seed}_j \) is the one \( P_1 \) sent to \( F_{\text{OT}} \) in Step 1. Therefore, \( P_2 \) can recompute \( \text{seed}_j \) from them. We just need to show that \( P_2 \) is able to recover \( x \) from a good index using \( \text{seed}_j \).

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 \( F_{\text{OT}} \) in Step 3: Since \( j \) is good, the \( c_{j,i,b}^R \) 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 \( R_{j,i,x[i]} = \text{PRF}_{\text{seed}_j}(i, "R") \oplus X_{i,x[i]} \) and the \( \text{seed}_j \) 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 \( F_{\text{OT}} \) if \( P_2 \) has \( \Omega = \Delta \). \( \square \)

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 \( A \) corrupting \( P_1 \) and denote \( x \) as the input \( P_1 \) sends to \( F_{\text{OT}} \) 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

\[
R_{j,i,x[i]} = \text{PRF}_{\text{seed}_j}(i, "R") \oplus X_{i,x'[i]},
\]

where \( \text{seed}_j \) is the seed \( P_2 \) recovers in Step 7. Therefore we conclude that

\[
\text{PRF}_{\text{seed}_j}(i, "R") = R_{j,i,x[i]} \oplus X_{i,x'[i]} = R_{j,i,x[i]} \oplus X_{i,1-x[i]}. 
\]

Although \( A \) receives \( X_{i,x[i]} \) in Step 3, \( X_{i,1-x[i]} \) remains completely random before \( A \) sends \( R_{j,i,x[i]} \). Further, \( A \) 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 \( A \). If \( A \) wants to “flip” a bit in \( x \), \( A \) needs to find some \( \text{seed}_j \) such that \( \{\text{PRF}_{\text{seed}_j}(i, "R")\}_{i \in \{n_1\}} \) equals a randomly chosen string, which is information theoretically infeasible if \( n_1 > 1 \). \( \square \)
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 \( A \) corrupting \( P_1 \) and denote \( x \) as the input \( P_1 \) sends to \( \mathcal{F}_{\text{cOT}} \) in Step 3. With probability at least \( 1 - (2^{-\rho} + \epsilon(k)) \), \( P_2 \) 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:

- \( \mathcal{E} \cap \mathcal{E}' \neq \emptyset \). 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 \( Z_{j,k,i,z[i]} \) from the \( j \)th 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 \( \Delta \). 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 \( A \) guesses \( \mathcal{E} \) correctly, which happens with at most \( 2^{-\rho} \) probability.

This completes the proof. \( \square \)

## 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 \( \rho \)-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 \( \rho n \) OTs and \( \rho n \) XOR gates; on the other hand, using a \( \rho \)-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 \( \rho \)-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 \( \rho \)-probe matrix.

**A general transformation to a sparse matrix.** We first reduce the number of XORs needed. Assuming a \( \rho \)-probe matrix with dimensions \( n \times cn \), we need \( c\rho n^2 \) XOR gates to process the \( \rho \)-probe matrices across all \( \rho \) 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 \( \rho \cdot \frac{c}{k} \epsilon(k)^2 = c\rho n^2 \). 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 \( \rho \)-probe matrix.** After applying the above technique, our problem is reduced to finding an efficient \( \rho \)-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].
Table 3: Choices for $c$ for $\rho$-probe matrix for $\rho = 40$, where $k$ is the chunk size after applying the sparse matrix transformation.

<table>
<thead>
<tr>
<th></th>
<th>40</th>
<th>65</th>
<th>80</th>
<th>103</th>
<th>143</th>
<th>229</th>
<th>520</th>
</tr>
</thead>
<tbody>
<tr>
<td>LP07</td>
<td>6.66</td>
<td>4.1</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>sS13</td>
<td>7.95</td>
<td>5.2</td>
<td>4.5</td>
<td>4.1</td>
<td>3.2</td>
<td>2.4</td>
<td>1.6</td>
</tr>
<tr>
<td>This work</td>
<td>5.67</td>
<td>4</td>
<td>3.5</td>
<td>3</td>
<td>2.5</td>
<td>2</td>
<td>1.5</td>
</tr>
</tbody>
</table>

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 $\rho$-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 $\text{Bin}(ck - k, \frac{1}{2})$. Therefore, we can compute the probability that $A$ is not a $\rho$-probe matrix as follows:

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

where $\text{cdf}(\cdot)$ is the cumulative distribution function for $\text{Bin}(ck - k, \frac{1}{2})$. 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 $\rho$-probe matrix over all circuits dominates the overall cost. As we can see, our design is about $10 \times$ better for just 1,024-bit inputs and can be $1000 \times$ better for 65,536-bit inputs. We are not able to compare beyond this point, because just storing the $\rho$-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 \texttt{mm_round} returns the concatenation of the highest bits from all $a[i]$s. This makes it possible to transpose a matrix of dimension $8 \times 16$ 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 $\rho$-probe matrix design with the prior best scheme [sS13]. When used in a malicious 2PC protocol, computing the $\rho$-probe matrix needs to be done $\rho$ times, and OT extension needs to process a $cn$-bit input because of the blowup of the input caused by the $\rho$-probe matrix.

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 $(h, g_1) := (g^r, g^r)$ 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 $\rho$ 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.
Table 5: Performance of our building blocks. The first row gives the running time of $P_2$ recovering its input when using a $\rho$-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.

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

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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>39</td>
<td>1034</td>
</tr>
<tr>
<td>AES</td>
<td>65</td>
<td>1442</td>
</tr>
<tr>
<td>SHA1</td>
<td>200</td>
<td>2007</td>
</tr>
<tr>
<td>SHA256</td>
<td>438</td>
<td>2621</td>
</tr>
</tbody>
</table>

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-toolkit4.

We instantiate the commitment scheme as $(SHA-1(x,r),r) \leftarrow Com(x)$. 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

4https://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  | 16,383  | 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       | 1,024   | 1,024   | 1,024   | 1,024 | 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   | 32,768  | 32,768  | 32,768  | 1,802,240| 9.43 s      | 9.8 s         | 2.6 GB      | 11.5%        |
| Sort 4096 32-bit   | 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. \textit{Running Time} reports the performance of our single execution over LAN; \textit{Projected Time} is calculated using the formula in §6.3; \textit{Total Comm.} is the total communication as measured by our implementation; and \textit{Non-GC Comm.} is the percentage of communication not used for garbled circuits.

The \texttt{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

### Table 8: Offline/online performance

<table>
<thead>
<tr>
<th>Function</th>
<th>Offline</th>
<th>Online</th>
<th>Amortized</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>27</td>
<td>12</td>
<td>(0.2) 89,492</td>
</tr>
<tr>
<td>AES</td>
<td>62</td>
<td>21</td>
<td>(3) 132,276</td>
</tr>
<tr>
<td>SHA1</td>
<td>206</td>
<td>52</td>
<td>(27) 226,570</td>
</tr>
<tr>
<td>SHA256</td>
<td>497</td>
<td>92</td>
<td>(128) 338,927</td>
</tr>
</tbody>
</table>

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. **Offline** 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.

### Table 9: Scalability of our protocol

<table>
<thead>
<tr>
<th></th>
<th>localhost</th>
<th>LAN</th>
<th>WAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time per $P_1$’s input bit</td>
<td>9.8</td>
<td>16</td>
<td>191.4</td>
</tr>
<tr>
<td>Time per $P_2$’s input bit</td>
<td>16.4</td>
<td>22.5</td>
<td>146.9</td>
</tr>
<tr>
<td>Time per output bit</td>
<td>13.3</td>
<td>20.3</td>
<td>131.1</td>
</tr>
<tr>
<td>Time per AND gate</td>
<td>1.7</td>
<td>4.4</td>
<td>63.1</td>
</tr>
</tbody>
</table>

All numbers are in microseconds per bit or microseconds per gate.

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 $\mu$s) 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 same 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


