Efficiently Enforcing Input Validity in Secure Two-party Computation

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

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

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

Abstract

Secure two-party computation based on cut-and-choose has made great strides in recent years, with a significant reduction in the total number of garbled circuits required. Nevertheless, the overhead of cut-and-choose can still be significant for large circuits (i.e., a factor of ρ in both communication and computation for statistical security $2^{-\rho}$).

We show that for a particular class of computation it is possible to do better. Namely, consider the case where a function on the parties' inputs is computed only if each party's input satisfies some publicly checkable predicate (e.g., is signed by a third party, or lies in some desired domain). Using existing cut-and-choose-based protocols, both the predicate checks and the function would need to be garbled ρ times. Here we show a protocol in which only the underlying function is garbled ρ times, and the predicate checks are each garbled only *once*. For certain natural examples (e.g., signature verification followed by evaluation of a million-gate circuit), this can lead to huge savings in communication (up to 80×) and computation (up to 56×). We provide detailed estimates using realistic examples to validate our claims.

1 Introduction

Secure two-party computation (2PC) allows two parties with private input to compute some common function such that both parties learn the output while keeping their inputs private. One way to build such protocols is to use *garbled circuits* [Yao82]. Here, one party, the *generator*, constructs a "garbled" version of a boolean circuit representing the function to compute, where each wire is represented by a *wire label* that hides the value on that underlying wire and each gate is garbled in such a way that the party evaluating the garbled circuit cannot learn the underlying bit of any given wire label. The generator sends this garbled circuit to the other party, the *evaluator*. The evaluator receives the wire labels associated with its input using *oblivious transfer* (OT), and evaluates the circuit to learn the output of the computation.

The basic garbled-circuit protocol described above is only secure against *semi-honest* adversaries, that is, adversaries that are assumed to follow the protocol but may try to deduce the other party's input from the protocol transcript. Lindell and Pinkas [LP07] showed how to secure garbled-circuit protocols against *malicious* adversaries (that is, adversaries who can deviate arbitrarily from the protocol) using the *cut-and-choose* technique. The basic idea with cut-and-choose is that the circuit generator constructs *multiple* garbled circuits, a certain fraction of which are opened by the evaluator to check that they are constructed correctly. If this check passes, the evaluator processes the remaining circuits and derives the appropriate output. Lindell and Pinkas [LP07] required 680 garbled circuits for statistical security 2^{-40} (i.e., such that a malicious generator can succeed in cheating only with probability $\leq 2^{-40}$). This was improved in a sequence of works [sS11, LP12, HKE13, Lin13, AMPR14], with the best current protocols requiring exactly ρ circuits to achieve statistical security $2^{-\rho}$. Although the number of circuits can be reduced in an *amortized* sense [HKK+14, LR14], it seems that the limit has been reached in the single-execution case.

Even with this progress over the last several years, most practical 2PC research still focuses on the semi-honest setting. We argue that this is due to several reasons. For one, a slowdown of $40\times$ to achieve security 2^{-40} is still significant. Moreover, even a protocol that is secure in the malicious model offers no assurance on its own that the adversarial party uses a "valid" input (for some definition of valid). Finally, in the semi-honest setting parties can rely on (some) *local computation* which can greatly reduce the size of the circuit that needs to be garbled. In contrast, in the malicious setting such local computation cannot (in general) be relied upon because there is no guarantee that an adversary correctly computes said computation. Below, we describe these latter two issues in more detail and describe how they can be addressed (inefficiently) using existing protocols.

Input validity. One inherent limitation of the malicious security model is that a malicious party can choose an arbitrary value as its input. This potentially allows a malicious party to learn a significant amount of information, or violate correctness (at least in an intuitive sense). As an example of the former, consider a shortest-path computation where one party holds a weighted graph, the other holds a source-destination pair, and both parties learn the length of the shortest path. By manipulating edge weights, the first party can ensure that it learns the source-destination pair of the other party. As an example of the latter, consider computing the average of several temperature readings, where one party uses a temperature of 1000°C.

One possible solution to this input-validity problem is to let the two parties verify that the other party's input is signed by some trusted party, or satisfies some other predicate. However, verifying a signature can require more than *one hundred billion* non-free gates [KMsB13]. Recalling that malicious security requires an additional $O(\rho)$ multiplicative overhead due to cut-and-choose, this approach appears impractical, especially if the underlying function to be computed is small.

Local computation. One popular technique to improve efficiency in the semi-honest model is to utilize local computation. Namely, instead of each party submitting their input directly, each party first performs some local computation on their input and submits the result of that local computation as input to some secure computation. (An interactive approach, in which a secure computation is run to generate intermediate values which are further processed by the parties locally before further secure computation is done, can also be used.) Some works have shown that for specific examples this approach improves the running time of (semi-honest) secure computation by orders of magnitude, including private set intersection [HEK12] and edit-distance estimation [WHZ+15], etc. One common characteristic shared by these works is that most of the computation is done locally such that the part of the function requiring secure computation is significantly, and in many cases asymptotically, smaller. However, in the malicious setting, local computation is not beneficial at all, since there is no guarantee that the malicious party provides the correct result of a local computation starting from some input. Thus, all computation must be integrated into the secure-computation protocol itself.

Abstracting the problem. We observe that the two problems mentioned above relate to a common problem were the two parties, holding inputs x and y, respectively, wish to compute a function of the form

$$f(x,y) := \text{“if } f_1(x) \text{ and } f_2(y) \text{ then } g(x,y) \text{ else } \perp\text{”},$$

where $f_1(\cdot)$ and $f_2(\cdot)$ are (public) predicates on each party's input and $g(\cdot,\cdot)$ is the underlying function the parties would like to compute. Note that this directly captures the input-validity problem, in that the predicate functions could check validity however the parties choose to define it. Likewise, for the local-computation problem we can have the predicates verify that the local computation was done correctly—something which can often be more efficient than re-doing the computation.

As $f(\cdot,\cdot)$ is a two-party function, we can compute it securely using any existing malicious 2PC protocol. We refer to this as the “generic solution.” In this work we show how it is possible to do *much* better by using cut-and-choose only on $g(\cdot,\cdot)$. For the predicate checks, we use the zero-knowledge-based-on-garbled-circuits approach of Jawurek et al. [JKO13] to evaluate $f_1(\cdot)$ and $f_2(\cdot)$. This allows us to garble $f_1(\cdot)$ and $f_2(\cdot)$ only *once*, while only garbling $g(\cdot,\cdot)$ a total of ρ times. Combining these protocols in a naive way, however, does not guarantee that a malicious party uses consistent inputs between the predicate circuits (namely $f_1(\cdot)$ and $f_2(\cdot)$) and the computation circuit (namely $g(\cdot,\cdot)$). In order to solve this consistency problem efficiently, we extend a building block in the protocol of Afshar et al. [AMPR14] and utilize a novel functionality we call “half-committed OT” which can be efficiently instantiated by adapting existing OT protocols. See details below.

To understand the performance gains of our protocol versus the generic solution, we present a detailed cost analysis, comparing the computation and communication costs of our protocol with that of Afshar et al. We obtain savings of up to $\approx 80\times$ in communication and $\approx 56\times$ in computation for many realistic examples. We refer to Section 5 for more details.

Concurrent work. A recent and concurrent work by Baum [Bau16] also provides a solution for a similar problem we considered here. Their technique uses universal hash functions to enforce the consistency, which enlarges the size of circuit to be used by the maliciously secure two party protocol.

1.1 Relevant Prior Work

Because our protocol relies heavily on the existing works of Jawurek et al. [JKO13] and Afshar et al. [AMPR14], we briefly recap how those constructions work.

Efficient zero-knowledge using garbled circuits [JKO13]. In a zero-knowledge proof-of-knowledge (ZKPoK), two parties, a *prover* and a *verifier*, have some common predicate $f(\cdot)$, and the prover would like to demonstrate to the verifier that it knows some *witness* w such that $f(w) = 1$, without revealing w to the verifier. Such a protocol is a particular case of 2PC, so any generic secure-computation protocol, with malicious security, could be used. Jawurek et al. [JKO13] showed, however, that one can do much better, and devised a ZKPoK protocol with essentially the same cost as a semi-honest garbled-circuit protocol for the predicate f.

The basic idea is as follows. The verifier sends a garbling of $f(\cdot)$ to the prover, who evaluates it using the input-wire labels it receives through OT, learning an output-wire label Z. The prover commits to this value, and then asks the verifier to open the garbled circuit so the prover can verify that the garbled circuit sent by the verifier indeed corresponds to the correct predicate $f(\cdot)$. If this is the case, the prover decommits to reveal Z to the verifier; if Z is the output-wire label corresponding to '1' then the verifier learns that the prover supplied a valid witness. Security of the OT implies that the prover's input w is hidden from the verifier; security of the garbled circuit implies that the prover cannot learn the correct output-wire label Z if its witness does not satisfy the predicate.

Efficient malicious two-party computation [AMPR14]. Afshar et al. [AMPR14] propose an optimized variant of Lindell's "fast cut-and-choose with cheating punishment" protocol [Lin13], which garbles ρ circuits for $2^{-\rho}$ statistical security.[1] The basic idea with Lindell's protocol is that if any of the evaluation circuits lead to inconsistent outputs, these inconsistencies can be used to recover the circuit generator's input x, allowing the evaluator to locally compute $f(x, y)$. Lindell's protocol requires running an additional secure computation protocol for the "cheating punishment" phase; Afshar et al. show how to remove this (computationally expensive) step. Their idea is as follows. The circuit generator P_1 begins by committing to its input bits using a specific ElGamal commitment scheme. Namely, for all $i \in [n_1]$, where n_1 is the input length, P_1 computes $\mathsf{EGCommit}_h(x_i; r) = (g^r, h^r g^{x_i})$, where $h = g^w$ for some secret value w known to P_1, and sends these commitments to P_2. Note that if the evaluator P_2 learns w it can break the commitments and thus learn x. Party P_1 then constructs garbled circuits such that if P_2 learns both output-wire labels in an evaluation circuit, then it learns w. Thus, if P_1 tries to cheat, P_2 can recover w and thus learn P_1's input, allowing P_2 to compute $f(x, y)$ locally. Party P_1's input consistency is enforced by having P_1 prove that the input-wire labels it provides for the evaluation circuits are commitments to the bits P_1 initially committed to.

1.2 Our Solution

In this work, we combine the works of Jawurek et al. [JKO13] and Afshar et al. [AMPR14] to handle functions with predicate checks on each party's input. The parties first prove (in zero-knowledge) that their inputs satisfy the requisite predicate, and if so, the parties compute the underlying function. The main technical difficulty is thus devising a mechanism for tying together the inputs of the predicate checks with the inputs to the underlying computation function. Namely, we need to enforce that, for example, the input P_1 supplies to $f_1(\cdot)$ is the *same* input used when computing $g(\cdot, \cdot)$. We describe how we do this for each party in turn.

Enforcing consistency on P_1's input. Recall that in the protocol of Afshar et al., P_1 commits (using a specific ElGamal commitment scheme) to each individual input bit of x at the beginning of the protocol, and then proves in zero-knowledge that the input-wire labels it provides to the evaluation circuits are commitments to those same input bits. Thus, in order to support input consistency across $f_1(\cdot)$ and $g(\cdot, \cdot)$ we need to somehow enforce that P_1's inputs to $f_1(\cdot)$ are the same as those it committed to initially. We do so by using a specific ElGamal-based OT protocol which works with the ElGamal commitment scheme used by P_1. Namely, the ElGamal commitments to

[1] While Afshar et al. also show how their protocol can be used to provide *non-interactive secure computation*, we do not utilize this property in our setting.

x_i sent by P_1 are used to construct P_2's OT messages encoding the input-wire labels to the garbling of $f_1(\cdot)$; P_1 can only recover those wire labels associated with the bit values it committed to.

In more detail, recall that P_1 commits to its input bits using the commitment scheme $\mathsf{EGCommit}_h(b;r) = (g^r, h^r g^b) = (A, B)$. Note that if $b = 0$ then the pair $(g, g^r, g^s h^t, A^s B^t)$ is a Diffie-Hellman tuple. Likewise, if $b = 1$ then the pair $(g, g^r, g^s h^t, A^s (B/g)^t)$ is a Diffie-Hellman tuple. Thus, letting (A_i, B_i) be the ElGamal commitment of input bit x_i, P_2 can encode the input-wire labels to the garbling of $f_1(\cdot)$ as

$$\begin{aligned}(\widehat{A}_{i,0}, \widehat{B}_{i,0}) &\leftarrow (g^{s_{i,0}} h^{t_{i,0}}, (A_i)^{s_{i,0}} (B_i)^{t_{i,0}} \cdot X_{i,0})\\ (\widehat{A}_{i,1}, \widehat{B}_{i,1}) &\leftarrow (g^{s_{i,1}} h^{t_{i,1}}, (A_i)^{s_{i,1}} (B_i/g)^{t_{i,1}} \cdot X_{i,1}),\end{aligned}$$

for random $s_{i,0}, t_{i,0}, s_{i,1}, t_{i,1}$, and send $\widehat{A}_{i,0}, \widehat{B}_{i,0}, \widehat{A}_{i,1}, \widehat{B}_{i,1}$ to P_1, who can only recover one of the two wire labels based on which value x_i it committed to.

Note that this OT protocol is not maliciously secure in the sense that a simulator cannot extract P_2's inputs. This is okay in our setting, as the garbling of $f_1(\cdot)$ is fully opened later in the protocol, and thus we can recover the wire labels in that step.

Another issue is that when simulating a malicious P_1, we need to be able to extract its input x. In the protocol of Afshar et al., this extraction happens when P_1 sends the garbled circuits to P_2; here, the simulator can learn w and thus break the commitments sent by P_1. However, in our protocol we need to extract x *earlier*, in particular in the phase where we check whether $f_1(x) = 1$. We do this by having P_1 prove in zero-knowledge that it knows the exponent of h used in the commitments. When simulating, we can thus extract this exponent and break the commitments, learning P_1's input.

Enforcing consistency on P_2's input. In this step we need to enforce that P_2's input y is consistent between $f_2(\cdot)$ and $g(\cdot,\cdot)$. Note that P_1 garbles both these functions: $f_2(\cdot)$ is garbled once and $g(\cdot,\cdot)$ is garbled ρ times, with around half being used as evaluation circuits. Thus, given the wire labels for y needed to compute $f_2(y)$, we devise a scheme that allows P_2 to derive the appropriate wire labels for $g(\cdot,\cdot)$. Thus, P_2 can derive only those wire labels related to its input y, whereas P_1 can derive both wire labels. Since OT hides which wire labels P_2 selects in the predicate function computation, P_1 never learns which wire labels P_2 has acquired and thus cannot learn which wire labels P_2 is able to derive for the underlying function computation. Likewise, because P_2 only retrieves one of the two wire labels for each input, it cannot derive the wire labels for the underlying function computation for those bits that are not part of its input y. We describe this in more detail below.

Clearly, the input-wire labels for $g(\cdot,\cdot)$ cannot be derived directly from the input-wire labels for $f_2(\cdot)$, as these wire labels are opened when P_2 verifies that P_1 indeed correctly garbled $f_2(\cdot)$. Instead, we introduce a specific OT protocol called *half-committed OT*. This is the same as sender-committed OT (where the sender is committed to its inputs such that it can later decommit these values to the receiver); however, in half-committed OT only *half* of the sender's inputs are committed. That is, the sender inputs $(m_{0,0}, m_{0,1})$ and $(m_{1,0}, m_{1,1})$, with the receiver receiving $(m_{b,0}, m_{b,1})$ for choice bit b. The sender can then later decommit to $m_{0,0}$ and $m_{1,0}$. Such a primitive can be easily realized using existing OT protocols, such as the efficient maliciously-secure OT protocol of Peikert et al. [PVW08]. We use half-committed OT when transferring the wire labels for the predicate circuit $f_2(\cdot)$. Let $Y_{i,0}, Y_{i,1}$ denote these wire labels. As input to the half-committed OT, party P_1

submits $(Y_{i,0}, r_{i,0})$ and $(Y_{i,1}, r_{i,1})$, for some random values $r_{i,0}$ and $r_{i,1}$. Note that when P_1 opens the committed values to enable P_2 to check that the garbling of $f_2(\cdot)$ was done correctly, it *only* reveals $Y_{i,0}$ and $Y_{i,1}$, and not $r_{i,0}$ and $r_{i,1}$. The parties use these latter values to derive the input-wire labels for the underlying circuit $g(\cdot,\cdot)$. Namely, P_1 constructs garblings of $g(\cdot,\cdot)$ with input-wire labels for P_2 equal to $\mathrm{PRF}_{r_{i,0}}(j)$ and $\mathrm{PRF}_{r_{i,1}}(j)$, where j denotes the jth garbling of $g(\cdot,\cdot)$, and PRF is a pseudorandom function. Thus, if P_2 chooses $b = 0$ in $\mathcal{F}_{\mathsf{hcOT}}$ it can *only* derive the zero-bit input-wire label for the ith input to $g(\cdot,\cdot)$ using $r_{i,0}$, and likewise, if $b = 1$ then P_2 can *only* derive the one-bit input-wire label.

The approach as described above however has a selective-failure attack in that P_1 can use, for example, $r'_{i,0} \neq r_{i,0}$ as input into the half-committed OT. If P_2's ith input is zero it aborts (since the input-wire labels it derives using $r'_{i,0}$ are invalid) and otherwise it succeeds. This allows P_1 to learn the ith bit of P_2's input. We can fix this by using the XOR-tree approach of Lindell and Pinkas [LP07]. Instead of P_2 having n_2 bits of input, the parties modify *both* the $f_2(\cdot)$ and $g(\cdot,\cdot)$ circuits such that P_2 now has ρn_2 bits of input, where ρ is the statistical security parameter, and the new inputs are XORed together to equal P_2's original input. Namely, let y be P_2's original input and let $y^1, \ldots, y^\rho$ be the new inputs. Then P_2 chooses the y^i values such that $\bigoplus y^i = y$. Thus, a selective-failure attack on a single input bit leaks a bit which reveals nothing about P_2's original input y. While P_1 can launch a selective failure attack on *multiple* input bits, it only learns a bit of y if it succeeds in guessing all ρ shares, and thus succeeds with probability $\leq 2^{-40}$.

Although the naive XOR-tree works, it leads to a blow-up of ρ in P_2's input size. However, Lindell and Pinkas [LP07] proposed a scheme that requires only $\max\{4n_2, 8\rho\}$ input wires for P_2, and thus for reasonably large input sizes the overhead is only $4\times$ (instead of $\rho\times$ using the naive XOR-tree approach).

1.3 Other Related Work

We have already touched upon the myriad of garbled-circuit-based protocols for malicious 2PC, and thus in this section we focus on malicious 2PC protocols based on other building blocks. One such approach is the "LEGO" technique, where instead of applying cut-and-choose at the circuit level one applies it at the gate level. A series of works [NO09, FJN+13, FJNT15] has investigated this approach, with the most recent TinyLEGO approach [FJNT15] giving competitive performance results in terms of communication with the garbled circuit approach. As an example, for circuits with one billion gates, the number of bits communicated when using TinyLEGO is around *half* that of the garbled circuit approach. However, we note that it is not clear whether TinyLEGO can be adapted to take advantage of privacy-free computation, as can be done in our protocol. Thus, while our communication gains are halved when compared with TinyLEGO, this still implies a roughly $40\times$ improvement in communication using our approach. In addition, it is not clear whether TinyLEGO is competitive with the garbled circuit approach from a *computation* standpoint.

Another line of malicious secure computation work has been based on using the GMW protocol [GMW87] with maliciously-secure MAC checks [NNOB12, DPSZ12, DKL+12]. These protocols work in the *preprocessing model*, and while they have very efficient (information theoretic) online running times, the required offline computation and communication is very heavy.

Functionality $\mathcal{F}_{\mathsf{2pc}}$

Private inputs: P_1 has input $x \in \{0,1\}^{n_1}$ and P_2 has input $y \in \{0,1\}^{n_2}$.
Common input: Circuit $C_0 : \{0,1\}^{n_1} \times \{0,1\}^{n_2} \to \{0,1\}^{n_3}$, where

$$C_0(x, y) := \text{if } f_1(x) \text{ and } f_2(y) \text{ then } g(x, y) \text{ else } \perp.$$

1. Upon receiving either (input, x) or $(\mathsf{input}, \perp)$ from P_1, proceed as follows:
   - If x was received and $f_1(x) = 1$, then send $(\mathsf{received}, \mathsf{ok})$ to P_2 and continue.
   - If either $\perp$ was received or $f_1(x) = 0$, send $(\mathsf{received}, \perp)$ to P_2 and halt.
2. Upon receiving either (input, y) or $(\mathsf{input}, \perp)$ from P_2, proceed as follows:
   - If y was received and $f_2(y) = 1$, then send $(\mathsf{received}, \mathsf{ok})$ to P_1 and continue.
   - If $\perp$ was received or $f_2(y) = 0$, send $(\mathsf{received}, \perp)$ to P_1 and halt.
3. Upon receiving either (abort) or $(\mathsf{continue})$ from P_1, proceed as follows:
   - If abort was received, send $(\mathsf{output}, \perp)$ to P_2 and halt.
   - If $\mathsf{continue}$ was received, send $(\mathsf{output}, g(x, y))$ to P_2 and halt.

Figure 2.1: Functionality $\mathcal{F}_{\mathsf{2pc}}$ for two-party secure computation with predicate checks.

2 Preliminaries

We use κ to denote the computational security parameter and ρ to denote the statistical security parameter. We assume the reader is familiar with secure computation and the cut-and-choose paradigm for constructing malicious protocols based on garbled circuits.

Two-party functionality for enforcing predicate checks. We consider a *reactive* two-party functionality $\mathcal{F}_{\mathsf{2pc}}$ of a certain form, where each party's input must satisfy some predicate function before some underlying function (computed on both parties' inputs) is run. In case a party's input does not satisfy the necessary predicate, the functionality outputs $\perp$ to the other party.

The functionality begins by taking either an input x or $\perp$ from P_1; if the functionality receives x such that $f_1(x) = 1$ then it sends an ok message to P_2 and waits for either an input y or $\perp$ from P_2, and otherwise it halts. Likewise, if the functionality receives y such that $f_2(y) = 1$ from P_2 then it sends an ok message to P_1 and otherwise it halts. If both parties send valid inputs to the functionality, then it waits for a $\mathsf{continue}$ message from P_1, at which point it outputs $g(x, y)$ to P_2 and halts. See Figure 2.1 for the formal description.

$\mathcal{F}_{\mathsf{2pc}}$ is slightly weaker than the non-reactive functionality $\mathcal{F}'_{\mathsf{2pc}}$ that accepts inputs x and y from the two parties, and then returns $\perp$ to both parties if either $f_1(x) = 0$ or $f_2(y) = 0$, and $g(x, y)$ otherwise. In particular, $\mathcal{F}_{\mathsf{2pc}}$ allows P_2 to learn whether $f_1(x) = 1$ even if $f_2(y) = 0$—something that is not possible when interacting with the non-reactive functionality $\mathcal{F}'_{\mathsf{2pc}}$ just described. In most practical scenarios, however, we expect that an honest P_1 would only ever use an input for which $f_1(x) = 1$, and so "leaking" that information to an attacker is insignificant.

Half-committed oblivious transfer. Our protocol requires a form of *committed* oblivious transfer (OT) in which only the first half of the inputs is sender-committed, whereas the second half is not. Namely, the sender's inputs are of the form (X_0, Y_0) and (X_1, Y_1), where on input b the

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

- On receiver input (choose, i, b), if no message of the form $(\mathsf{choose}, i, \cdot)$ exists then store (choose, i, b) and send (chosen, i) to the sender.
- On sender input $(\mathsf{transfer}, i, X_0, Y_0, X_1, Y_1)$, if no message of the form $(\mathsf{transfer}, i, \cdot, \cdot, \cdot, \cdot)$ exists and a message of the form (chosen, i, b) does, then send $(\mathsf{transferred}, i, X_b, Y_b)$ to the receiver.
- On sender input $(\mathsf{open\text{-}all})$, send $(\mathsf{transferred}, i, X_0, X_1)$, for all i, to the receiver, and halt.

Figure 2.2: Half-committed oblivious transfer ideal functionality $\mathcal{F}_{\mathsf{hcOT}}$.

Protocol Π_{hcOT}

Inputs: The sender has input (X_0, Y_0, X_1, Y_1); the receiver has input $b \in \{0, 1\}$.
Auxiliary input: Tuple $(\mathbb{G}, q, g_0)$, where $\mathbb{G}$ is a group of order q with generator g_0.

- The receiver chooses $y, \alpha_0 \leftarrow\$ \mathbb{Z}_q$, sets $\alpha_1 := \alpha_0 + 1$, computes $g_1 := (g_0)^y$, $h_0 := (g_0)^{\alpha_0}$, and $h_1 := (g_1)^{\alpha_1}$, and sends (g_1, h_0, h_1) to the sender.
- The receiver proves in zero-knowledge that $(g_0, g_1, h_0, h_1/g_1)$ is a Diffie-Hellman tuple.
- The receiver chooses r at random, computes $g := (g_b)^r$ and $h := (h_b)^r$, and sends (g, h) to the sender.
- Define the function $RAND(w, x, y, z) = (w^s y^t, x^s z^t, s, t)$ for $s, t \leftarrow\$ \mathbb{Z}_q$.
- The sender computes

$$\begin{aligned}(u_{0,0}, v_{0,0}, s_{0,0}, t_{0,0}) &\leftarrow\$ RAND(g_0, g, h_0, h),\\ (u_{0,1}, v_{0,1}, s_{0,1}, t_{0,1}) &\leftarrow\$ RAND(g_0, g, h_0, h),\\ (u_{1,0}, v_{1,0}, s_{1,0}, t_{1,0}) &\leftarrow\$ RAND(g_1, g, h_1, h),\\ (u_{1,1}, v_{1,1}, s_{1,1}, t_{1,1}) &\leftarrow\$ RAND(g_1, g, h_1, h),\end{aligned}$$

  and sends $(u_{0,0}, v_{0,0} \cdot X_0, u_{0,1}, v_{0,1} \cdot Y_0, u_{1,0}, v_{1,0} \cdot X_1, u_{1,1}, v_{1,1} \cdot Y_1)$ to the receiver.
- The receiver computes $X_b := v_{0,b} \cdot X_b / (u_{0,b})^r$ and $Y_b := v_{1,b} \cdot X_b / (u_{1,b})^r$.
- To open the commitments to X_0 and X_1, the sender sends $s_{0,0}, t_{0,0}$ and $s_{1,0}, t_{1,0}$, and the receiver recomputes $RAND(g_b, g, h_b, h)$ using randomness $s_{1,b}, t_{1,b}$, for $b \in \{0, 1\}$.

Figure 2.3: Half-committed oblivious transfer implementation, based on the plain-model variant of the Peikert et al. oblivious transfer [LP11, PVW08].

receiver gets (X_b, Y_b). What makes the OT *half-committed* is that the sender can later decommit to *only* the values X_0 and X_1, and not Y_0 and Y_1.

Our functionality is modeled after the sender-committed OT of Jawurek et al. [JKO13, Fig. 3]; see Figure 2.2. The main difference is that in our functionality the sender inputs two pairs of messages, and when opening the values, only the first entry in each pair is revealed to the receiver.

Note that the maliciously-secure OT protocol of Peikert et al. [PVW08] can be used to construct half-committed OT in a straightforward manner. Namely, to decommit to a given input, the sender reveals the randomness used to mask *only* that input; see Figure 2.3.

3 Our Protocol

Our construction carefully combines Jawurek et al.'s ZKPoK protocol [JKO13] with the maliciously secure 2PC protocol of Afshar et al [AMPR14], where the functions we are interested in are of the form

$$f(x,y) = \text{"if } f_1(x) = 1 \text{ and } f_2(y) = 1 \text{ then } g(x,y) \text{ else } \perp\text{".}$$

As we presented the protocol intuition in the Introduction, we jump straight to the full protocol description, and we assume familiarity with both of the works we build off of.

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: Circuit $C_0 : \{0,1\}^{n_1} \times \{0,1\}^{n_2} \to \{0,1\}^{n_3}$, where

$$C_0(x,y) := \text{if } f_1(x) \text{ and } f_2(y) \text{ then } g(x,y) \text{ else } \perp;$$

computational security parameter κ; statistical security parameter ρ; hash function $H : \{0,1\}^* \to \{0,1\}^\kappa$; commitment scheme $(\mathsf{Com}, \mathsf{Open})$; ideal functionalities $\mathcal{F}_{\mathsf{hcOT}}$ and $\mathcal{F}_{\mathsf{OT}}$.

Protocol:

Check that $f_1(x) = 1$:

1. If $f_1(x) = 0$ then P_1 sends $\perp$ to P_2.
2. P_1 chooses $w \leftarrow\!\$\, \mathbb{Z}_p$, computes $h \leftarrow g^w$, and sends h to P_2. P_1 gives a zero-knowledge proof of knowledge that it knows w such that $g^w = h$.
3. P_2 constructs garbled circuit GC_{f_1} of function f_1. Let $\{X_{i,b}\}_{i\in[n_1],b\in\{0,1\}}$ denote the input-wire labels.
4. For $i \in [n_1]$, P_1 computes $(A_i, B_i) \leftarrow \mathsf{EGCommit}_h(x_i; r_i)$, for random r_i, and sends (A_i, B_i) to P_2. Denote these as P_1's *input commitments.*
5. For $i \in [n_1]$, P_2 computes

$$(\widehat{A}_{i,0}, \widehat{B}_{i,0}) \leftarrow (g^{s_{i,0}} h^{t_{i,0}}, A_i^{s_{i,0}} B_i^{t_{i,0}} \cdot X_{i,0})$$
$$(\widehat{A}_{i,1}, \widehat{B}_{i,1}) \leftarrow (g^{s_{i,1}} h^{t_{i,1}}, A_i^{s_{i,1}} (B_i/g)^{t_{i,1}} \cdot X_{i,1}),$$

   for random $s_{i,0}, t_{i,0}, s_{i,1}, t_{i,1}$, and sends $\widehat{A}_{i,0}, \widehat{B}_{i,0}, \widehat{A}_{i,1}, \widehat{B}_{i,1}$ to P_1.
6. For $i \in [n_1]$, P_1 recovers X_{i,x_i} by computing $\widehat{B}_{i,x_i}/(\widehat{A}_{i,x_i})^{r_i}$.
7. P_2 sends GC_{f_1} to P_1, who evaluates it, learning output-wire label Z_{f_1}. P_1 computes $(\mathsf{com}_{f_1}, \mathsf{decom}_{f_1}) \leftarrow\!\$\, \mathsf{Com}(Z_{f_1})$, where Com is an equivocal and extractable commitment scheme, and sends com_{f_1} to P_2.
8. P_2 sends $\{s_{i,0}, t_{i,0}, s_{i,1}, t_{i,1}\}_{i\in[n_1]}$ to P_1, who recovers all the input-wire labels and aborts if GC_{f_1} was not constructed correctly. Otherwise, P_1 sends decom_{f_1} to P_2, who computes $Z_{f_1} \leftarrow \mathsf{Open}(\mathsf{com}_{f_1}, \mathsf{decom}_{f_1})$. If Z_{f_1} is the 1-bit output-wire label of GC_{f_1} then P_2 continues. Otherwise, P_2 outputs $\perp$.

Check that $f_2(y) = 1$:

9. If $f_2(y) = 0$ then P_2 sends $\perp$ to P_1.
10. P_1 constructs garbled circuit GC_{f_2} of function $f_2'(y^1, \ldots, y^\rho) = f_2(\bigoplus_i y^i)$, where each y^i is an n_2-bit bitstring. Let $\{Y_{i,b}\}_{i\in[\rho n_2],b\in\{0,1\}}$ denote the input wires.
11. For $i \in [\rho n_2]$, P_1 chooses $r_{i,0} \leftarrow\!\$\, \{0,1\}^\kappa$ and $r_{i,1} \leftarrow\!\$\, \{0,1\}^\kappa$.
12. P_1 and P_2 run $\mathcal{F}_{\mathsf{hcOT}}$ ρn_2 times, where in the ith run P_1 inputs $(\mathsf{transfer}, i, Y_{i,0}, r_{i,0}, Y_{i,1}, r_{i,1})$ acting as the sender and P_2 inputs $(\mathsf{choose}, i, y_i)$ acting as the receiver, receiving $(\mathsf{transferred}, i, Y_{i,y_i}, r_{i,y_i})$ as output.
13. P_1 sends GC_{f_2} to P_2, who evaluates it, learning output wire label Z_{f_2}. P_2 computes $(\mathsf{com}_{f_2}, \mathsf{decom}_{f_2}) \leftarrow\!\$\, \mathsf{Com}(Z_{f_2})$, where Com is an extractable commitment, and sends com_{f_2} to P_1.
14. P_1 sends $(\mathsf{open\text{-}all})$ to $\mathcal{F}_{\mathsf{hcOT}}$ with P_2 receiving $(\mathsf{transferred}, i, Y_{i,0}, Y_{i,1})$ for all i. P_2 uses these wire labels to check that GC_{f_2} was constructed correctly, and if not P_2 aborts. Otherwise, P_2 sends decom_{f_2} to P_1, who computes $Z_{f_2} \leftarrow \mathsf{Open}(\mathsf{com}_{f_2}, \mathsf{decom}_{f_2})$. If Z_{f_2} is the 1-bit output-wire label of GC_{f_2} then P_1 continues. Otherwise, P_1 outputs $\perp$.

Evaluate $g(x, y)$:

15. For $i \in [n_1]$, P_1 chooses $w_{i,0} \leftarrow\$ \mathbb{Z}_p$ and sets $w_{i,1} := w - w_{i,0}$, computes *output commitments* $h_{i,0} := g^{w_{i,0}}$ and $h_{i,1} := g^{w_{i,1}}$, and sends $\{h_{i,0}\}$ and $\{h_{i,1}\}$ to P_2.
16. For $j \in [\rho]$, P_1 chooses seed $\mathsf{seed}_j \leftarrow\$ \{0,1\}^\kappa$ and key $k_j \leftarrow\$ \{0,1\}^\kappa$.
17. P_1 and P_2 run $\mathcal{F}_{\mathsf{OT}}$ ρ times, where in the jth run P_1 inputs (k_j, seed_j) acting as the sender, and P_2 inputs $b \leftarrow\$ \{0,1\}$ acting as the receiver.
18. For $j \in [\rho]$, proceed as follows:
    (a) For $i \in [n_1]$ and $b \in \{0,1\}$, P_1 computes $u_{j,i,b} \leftarrow \mathsf{EGCommit}_h(b; r_{j,i,b})$, where $r_{j,i,b}$ is derived from seed_j.
    (b) P_1 constructs garbling GC_j of function $g'(x, y_1, \ldots, y_\rho) = g(x, \bigoplus_i y_i)$, where P_1's ith input-wire labels are defined as $\{H(u_{j,i,0}), H(u_{j,i,1})\}$, P_2's ith input-wire labels are defined as $\{\mathrm{PRF}_{r_{i,0}}(j), \mathrm{PRF}_{r_{i,1}}(j)\}$, where $r_{i,0}$ and $r_{i,1}$ are the values input to $\mathcal{F}_{\mathsf{hcOT}}$ in Step 12, and the randomness used to construct GC_j is derived from seed_j. P_1 sends GC_j to P_2.
    (c) For $i \in [n_1]$, P_1 computes $(\mathsf{com}_{j,i,0}, \mathsf{decom}_{j,i,0}) \leftarrow \mathsf{Com}(u_{j,i,0})$, $(\mathsf{com}_{j,i,1}, \mathsf{decom}_{j,i,1}) \leftarrow \mathsf{Com}(u_{j,i,1})$, and sends $\{\mathsf{com}_{j,i,\pi}, \mathsf{com}_{j,i,1-\pi} : \pi \leftarrow\$ \{0,1\}\}$ to P_2.
    (d) For $i \in [n_3]$, P_1 chooses $K_{j,i,0}, K_{j,i,1} \leftarrow\$ \mathbb{Z}_p$ and sends *output recovery commitments* $h_{i,0} \cdot g^{K_{j,i,0}}$ and $h_{i,1} \cdot g^{K_{j,i,1}}$ to P_2. Likewise, P_1 sends $\mathsf{Enc}_{Z_{i,0}}(K_{j,i,0})$ and $\mathsf{Enc}_{Z_{i,1}}(K_{j,i,1})$ to P_2.
    (e) Let

$$\begin{aligned} \mathsf{Inputs}_j &\leftarrow \{\mathsf{com}_{j,i,x_i}, \mathsf{decom}_{j,i,x_i}\}_{i \in [n_1]} \\ \mathsf{InputEquality}_j &\leftarrow \{r_i - r_{j,i,x_i}\}_{i \in [n_1]} \\ \mathsf{OutputDecom}_j &\leftarrow \{(w_{i,0} + K_{j,i,0}, w_{i,1} + K_{j,i,1})\}_{i \in [n_3]} \end{aligned}$$

    P_1 sends $\mathsf{Enc}_{k_j}(\mathsf{Inputs}_j, \mathsf{InputEquality}_j, \mathsf{OutputDecom}_j)$ to P_2.
19. For all check circuits j (i.e., where P_2 received seed_j in Step 17), proceed as follows:
    (a) P_2 checks that seed_j generates GC_j and the other values constructed using randomness derived from seed_j, and aborts if not.
20. Set $\mathsf{cheat} := 0$. For all evaluation circuits j (i.e., where P_2 received key k_j in Step 17), proceed as follows:
    (a) P_2 decrypts $\mathsf{Enc}_{k_j}(\mathsf{Inputs}_j, \mathsf{InputEquality}_j, \mathsf{OutputDecom}_j)$.
    (b) For $i \in [n_1]$, P_2 computes $\widetilde{u}_{j,i,x_i} \leftarrow \mathsf{Open}(\mathsf{com}_{j,i,x_i}, \mathsf{decom}_{j,i,x_i})$ and checks that $\widehat{u}_{j,i,x_i} \cdot (g^{r_i - r_{j,i,x_i}}, h^{r_i - r_{j,i,x_i}}) = \mathsf{EGCommit}_h(x_i; r_i)$ for all $i \in [n_1]$, otherwise set $\mathsf{cheat} := 1$.
    (c) For $i \in [n_3]$ and $b \in \{0,1\}$, P_1 checks that $g^{w_{i,b} + K_{j,i,b}}$ equals the output recovery commitments sent by P_1, otherwise set $\mathsf{cheat} := 1$.
    (d) P_2 evaluates GC_j, using $\mathrm{PRF}_{r_{i,y_i}}(j)$ as its input-wire labels, learning output-wire labels $\{Z_i\}$. P_2 then uses these labels to learn the appropriate $K_{j,i,b}$ values, and uses these to check that $h_{j,b} \cdot g^{K_{j,i,b}}$ equals the appropriate output recovery commitment sent by P_1; otherwise set $\mathsf{cheat} := 1$. If this succeeds, P_2 marks the circuit as "semi-trusted."
21. If $\mathsf{cheat} = 1$ then abort. Otherwise, if all the semi-trusted circuits have the same output wire labels, P_2 outputs that value. Otherwise, let $Z_{j,i}$ and $Z_{j',i}$ be two differing output wire labels for garbled circuits j and j' and output wire i. P_2 can extract w_i^0 and w_i^1 by using the sets $\mathsf{OutputDecom}_j$ and $\mathsf{OutputDecom}_{j'}$, and thus learn w, allowing P_2 to decrypt P_1's initial commitments to learn x. P_2 then outputs $g(x, y)$.

Proof of security. We now prove that the above protocol realizes $\mathcal{F}_{\mathsf{2pc}}$ in the $(\mathcal{F}_{\mathsf{hcOT}}, \mathcal{F}_{\mathsf{OT}})$-hybrid model by constructing simulators for the case that either P_1 or P_2 is corrupted.

Malicious P_1. Suppose adversary $\mathcal{A}$ corrupts P_1. We construct a simulator $\mathcal{S}$ as follows.

1. $\mathcal{S}$ invokes $\mathcal{A}$ on its input.

2. If $\mathcal{A}$ sends $\perp$ in Step 1, $\mathcal{S}$ sends $(\mathsf{input}, \perp)$ to $\mathcal{F}_{\mathsf{2pc}}$ and outputs whatever $\mathcal{A}$ outputs.

3. In Step 2, $\mathcal{S}$ acts as an honest P_2. If the ZKPoK fails then $\mathcal{S}$ sends $(\mathsf{input}, \perp)$ to $\mathcal{F}_{\mathsf{2pc}}$ and outputs whatever $\mathcal{A}$ outputs. Otherwise, $\mathcal{S}$ extracts w from the ZKPoK.

4. In Step 4, $\mathcal{S}$ uses w extracted above to extract $x \in \{0,1\}^{n_1} \cup \{\perp\}$ from the commitments sent by $\mathcal{A}$, where $x = \perp$ if any of the commitments are invalid.

5. $\mathcal{S}$ continues to act as an honest P_2 would. In Step 8, $\mathcal{S}$ checks if either $x = \perp$ or $f_1(x) = 0$; if so, $\mathcal{S}$ sends $(\mathsf{input}, \perp)$ to $\mathcal{F}_{\mathsf{2pc}}$ and outputs whatever $\mathcal{A}$ outputs. Otherwise, $\mathcal{S}$ sends (input, x) to $\mathcal{F}_{\mathsf{2pc}}$, receiving either $(\mathsf{received}, \mathsf{ok})$ or $(\mathsf{received}, \perp)$ from $\mathcal{F}_{\mathsf{2pc}}$. If $\perp$ was received, $\mathcal{S}$ sends $\perp$ to $\mathcal{A}$ in Step 9 and outputs whatever $\mathcal{A}$ outputs.

6. $\mathcal{S}$ extracts $\mathcal{A}$'s input to $\mathcal{F}_{\mathsf{hcOT}}$, and uses these values to open the garbled circuit sent by $\mathcal{A}$, thus learning the one-bit output-wire label Z^1. $\mathcal{S}$ sends $\mathsf{Com}(Z^1)$ to $\mathcal{A}$.

7. $\mathcal{S}$ receives the opening to $\mathcal{F}_{\mathsf{hcOT}}$ and checks consistency with the values received above. If anything fails, $\mathcal{S}$ sends (abort) to $\mathcal{F}_{\mathsf{2pc}}$ and outputs whatever $\mathcal{A}$ outputs.

8. $\mathcal{S}$ continues to act as an honest P_2 would. If $\mathsf{cheat} = 0$ in Step 21 then $\mathcal{S}$ sends $(\mathsf{continue})$ to $\mathcal{F}_{\mathsf{2pc}}$ and outputs whatever $\mathcal{A}$ outputs. Otherwise, (i.e., $\mathsf{cheat} = 1$), $\mathcal{S}$ sends (abort) to $\mathcal{F}_{\mathsf{2pc}}$ and outputs whatever $\mathcal{A}$ outputs.

We now prove that the view of $\mathcal{A}$ is computationally indistinguishable in the real and ideal worlds. We do so by a series of hybrid experiments.

- $\mathbf{Hybrid_1}$. Same as the real execution.

- $\mathbf{Hybrid_2}$. Same as $\mathbf{Hybrid_1}$, except that P_2 extracts w from the ZKPoK and aborts if it fails to extract.

  These two hybrids are computationally indistinguishable, as by the security of the ZKPoK the probability that P_2 fails to extract w is negligible.

- $\mathbf{Hybrid_3}$. Same as $\mathbf{Hybrid_2}$, except that P_2 aborts in Step 8 if $f_1(x) = 0$.

  These two hybrids are computationally indistinguishable by the hiding property of the ElGamal-based oblivious transfer and the security of the garbling scheme. Namely, in $\mathbf{Hybrid_2}$, $\mathcal{A}$ cannot recover the appropriate input-wire label in Step 5 for those input bits which are incorrectly committed and likewise can only recover one of the two input-wire labels for those input bits which are correctly committed. Thus, by the authenticity property of the garbling scheme, $\mathcal{A}$ is unable to recover the one-bit output-wire label Z^1 with high probability. Thus, if $\mathcal{A}$ can distinguish between $\mathbf{Hybrid_2}$, where P_2 aborts due to $\mathcal{A}$ committing to an invalid output-wire label, and $\mathbf{Hybrid_3}$, where P_2 aborts regardless of what $\mathcal{A}$ commits to, then this leads to an attack on the authenticity property of the garbling scheme.

- **Hybrid$_4$**. Same as **Hybrid$_3$**, except that P_2 aborts if all the evaluated circuits are not good.

  These two hybrids are perfectly indistinguishable except that P_2 may abort in **Hybrid$_4$** and not **Hybrid$_3$**. However, this only happens if $\mathcal{A}$ correctly guesses which circuits will end up as check versus evaluation circuits, which happens with probability $2^{-\rho}$.

- **Hybrid$_5$**. Same as **Hybrid$_4$**, except that P_2 uses P_1's extracted input x to compute and output $g(x, y)$ instead of evaluating the garbled circuits.

  These two hybrids are perfectly indistinguishable because if $\mathcal{A}$ tries to cheat in **Hybrid$_5$** then P_2 can extract $\mathcal{A}$'s input and just compute $g(x, y)$ locally and otherwise P_2 retrieves $g(x, y)$ by evaluating the garbled circuits.

As **Hybrid$_5$** is the same as the ideal world protocol, this completes the proof for a malicious P_1. ■

Malicious P_2. Suppose adversary $\mathcal{A}$ corrupts P_2. We construct a simulator $\mathcal{S}$ as follows.

1. $\mathcal{S}$ invokes $\mathcal{A}$ on its input.

2. If $\mathcal{S}$ receives $(\mathsf{input}, \bot)$ from $\mathcal{F}_{\mathsf{2pc}}$, then $\mathcal{S}$ sends $\bot$ to $\mathcal{A}$ and outputs whatever $\mathcal{A}$ outputs.

3. $\mathcal{S}$ acts as an honest P_1 would, using 0^{n_1} as P_1's input, until Step 7, at which point $\mathcal{S}$ commits to a random value.

4. $\mathcal{S}$ continues to act as an honest P_2 would, where in Step 8 it opens the garbled circuit sent by $\mathcal{A}$ and learns the one-bit output-wire label Z^1. If $\mathcal{S}$ fails to open the garbled circuit, it sends $\bot$ to $\mathcal{F}_{\mathsf{2pc}}$ and outputs whatever $\mathcal{A}$ outputs. Otherwise, it equivocates on its previously sent commitment to make the committed value equal to Z^1.

5. In Step 9, if $\mathcal{A}$ sends $\bot$ then $\mathcal{S}$ sends $\bot$ to $\mathcal{F}_{\mathsf{2pc}}$ and outputs whatever $\mathcal{A}$ outputs.

6. $\mathcal{S}$ extracts y from $\mathcal{F}_{\mathsf{hcOT}}$ and proceeds to act as an honest P_1 would until Step 13. Here, if $f_2(y) = 0$ then $\mathcal{S}$ sends $(\mathsf{input}, \bot)$ to $\mathcal{F}_{\mathsf{2pc}}$, outputting whatever $\mathcal{A}$ outputs.

7. $\mathcal{S}$ continues to act as an honest P_1 would until Step 17. Here, $\mathcal{S}$ extracts $\mathcal{A}$'s choices as to which circuits are check circuits and which are evaluation circuits. For check circuit j, $\mathcal{S}$ replaces the key k_j input to $\mathcal{F}_{\mathsf{OT}}$ with a random string.

8. In Step 18, $\mathcal{S}$ sends $(\mathsf{input}, \mathsf{ok})$ to $\mathcal{F}_{\mathsf{2pc}}$, receiving (output, z), and proceeds as follows:

   - For the check circuits, $\mathcal{S}$ constructs them as an honest P_1 would.
   - For the evaluation circuits, $\mathcal{S}$ uses fresh randomness to generate everything related to the garbling and garbles a circuit with fixed output z. It also replaces the input wire label $\mathrm{PRF}_{r_{i,1-y_i}}(j)$, where y_i denotes the ith input of P_2, with a random wire label, the commitment $\mathsf{Com}(u_{j,i,1-y_i})$ values with commitments to zeros, and the encryption $\mathsf{Enc}_{Z_{i,1-z_i}}(K_{j,i,1-z_i})$, where z_i denotes the ith output, with an encryption to zeros.

9. $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

We now prove that the view of $\mathcal{A}$ is computationally indistinguishable in the real and ideal worlds. We do so by a series of hybrid experiments.

- $\mathbf{Hybrid_1}$. Same as the real execution.
- $\mathbf{Hybrid_2}$. Same as $\mathbf{Hybrid_1}$, except P_1 equivocates on the commitment it sends to P_2 in Step 7 to be the output of GC_{f_1}.

  These two hybrids are computationally indistinguishable based on the security of the equivocal commitment scheme.
- $\mathbf{Hybrid_3}$. Same as $\mathbf{Hybrid_2}$, except that in Step 14 P_1 aborts if $f_2(y) = 0$.

  These two hybrids are computationally indistinguishable based on the authenticity property of the garbled circuit.
- $\mathbf{Hybrid_4}$. Same as $\mathbf{Hybrid_3}$, except that P_1 replaces the k_j values for the check circuits with random values and generates the evaluation circuits using fresh randomness.

  These two hybrids are perfectly indistinguishable in the $\mathcal{F}_{\mathsf{OT}}$-hybrid model.
- $\mathbf{Hybrid_5}$. Same as $\mathbf{Hybrid_4}$, except that P_1 uses 0^{n_1} as its input to the check circuits.

  These two hybrids are computationally indistinguishable by the security of the encryption scheme.
- $\mathbf{Hybrid_6}$. Same as $\mathbf{Hybrid_5}$, except that P_1 replaces the commitments of $u_{j,i,1-y_i}$ with commitments to zeros in the evaluation circuits.

  These two hybrids are computationally indistinguishable by the security of the commitment scheme.
- $\mathbf{Hybrid_7}$. Same as $\mathbf{Hybrid_6}$, except that P_1 uses the output z of $\mathcal{F}_{\mathsf{2pc}}$ to construct fake garbled circuits with fixed output z for all evaluation circuits.

  These two hybrids are computationally indistinguishable by the security of the garbling scheme.
- $\mathbf{Hybrid_8}$. Same as $\mathbf{Hybrid_7}$, except that P_1 replaces the output encryptions for all output bits that do not correspond to z with encryptions of zero.

  These two hybrids are computationally indistinguishable by the security of the encryption scheme.
- $\mathbf{Hybrid_9}$. Same as $\mathbf{Hybrid_8}$, except that P_1 replaces its input with 0^{n_1} in the evaluation circuits and input commitments.

  These two hybrids are computationally indistinguishable by the security of the ElGamal commitment scheme.

- **Hybrid$_{10}$**. Same as **Hybrid$_9$**, except that P_1 replaces the input-wire labels for P_2's input that do not correspond to y with random strings.

  These two hybrids are computationally indistinguishable by the security of the PRF. Namely, if $\mathcal{A}$ can distinguish between the random strings and the correctly computed wire labels it can break the security of the PRF.

As **Hybrid$_{10}$** is the same as the ideal world protocol, this completes the proof for a malicious P_2. ■

4 Protocol Optimizations

We begin by noting a couple of immediate optimizations to our protocol. First off, assuming the random oracle model, we can instantiate all the commitment operations with a hash function. We also note that we can use *privacy-free* garbled circuits [FNO15] with the "half gate" optimization [ZRE15] for the garbling of f_1 and f_2, taking only one ciphertext per non-free gate. Finally, the ZKPoK that P_1 knows some w such that $h = g^w$ can be efficiently implemented using a Schnorr protocol [Sch90].

As our protocol requires public key operations for both P_1's and P_2's inputs, we consider optimizations to reduce the number of exponentiations required. First off, when P_1 computes values of the form $g^s h^t$ in $\mathsf{EGCommit}$ and the protocol for $\mathcal{F}_{\mathsf{hcOT}}$, only one exponentiation is needed since P_1 knows w such that $h = g^w$ and thus can directly compute g^{s+wt} $(= g^s h^t)$. For P_2, $g^s h^t$ can be computed more efficiently using the "Euclidean method" described by de Rooij [de 95]. The high level idea is to apply the following observation recursively:

$$g^s h^t = (gh^q)^s h^p, q = \lfloor \frac{t}{s} \rfloor, p = t \mod s.$$

We also note that for both P_1 and P_2, most of the exponentiations are *fixed-base* exponentiations, which can be computed much more efficiently using pre-computed tables [BGMW93].

We also note that our protocol as written only addresses the situation where all the input bits are used both in the predicate check stage (i.e., the proofs that $f_1(x) = 1$ and $f_2(y) = 1$) and the computation stage (i.e., the computation of $g(x, y)$), which may not always be the case. When only parts of the input are used in the predicate check or computation stage, we do not need the heavy machinery we use to ensure input consistency between each party's input in the two stages.

To be more specific, we consider the input of each party as three parts:

1. Input used only in the predicate check stage (denote these inputs as x_1, y_1);
2. Inputs used in both the predicate check and computation stages (denote these inputs as x_2, y_2);
3. Inputs used only in the computation stage (denote these inputs as x_3, y_3).

For the first case (i.e., inputs x_1 and y_1) we can use (standard) committed OT which allows us to utilize OT extension. For the third case (i.e., inputs x_3 and y_3), we can handle these as in the work of Afshar et al. [AMPR14]; see below for details.

Denote P_1's input by $x = (x_1 \| x_2 \| x_3)$, P_2's input by $y = (y_1 \| y_2 \| y_3)$, and the function to be computed by:

$$f(x, y) := \text{ if } f_1(x_1, x_2) \text{ and } f_2(y_1, y_2) \text{ then } g(x_2, x_3, y_2, y_3) \text{ else } \perp.$$

We can construct a protocol for dealing with this extended case as follows. It is the same as the protocol described in Section 3 except with the following changes:

1. For input x_1, we can skip the input commitment steps (Steps 4–6) and checking step (Step 8). This allows us to use a committed OT which works with OT extension.

2. For input y_1, we can skip the XOR-tree (Step 10) and half-committed OT (Step 12). Instead, we can use committed OT as above.

3. For inputs x_2 and y_2, these are handled as in our original protocol.

4. When computing $g(\cdot, \cdot)$, we use $\mathsf{EGCommit}$ to ensure the consistency of x_3 among computation circuits.

5. For input y_3 we do not need the XOR-tree, and can instead use committed OT during the computation stage.

For several real world examples, these extensions lead to important practical improvements; see Section 5.

5 Evaluation

In this section, we compare our protocol with generic malicious two-party computation protocols for several example functions to showcase the gains in communication and computation that our approach gives. In particular, we compare our protocol with the protocol of Afshar et al. [AMPR14], the most efficient and practical malicious 2PC construction that we are aware of. We refer to this protocol as the "generic solution" in contrast to our solution which is specifically designed for the type of functions we consider. We evaluate the improvement based on the speedup of both computation and communication. We do so by calculating the number of symmetric key operations, public key operations, and bytes sent by both our protocol and the generic solution. While obviously a rough approximation of the actual running time of an implementation, we believe this gives a good benchmark independent of implementation details, computer/network configuration, etc.

While we are aware of more efficient *customized* protocols for some of the examples discussed below, these protocols are not as flexible as our approach. For example, it is usually very difficult, and sometimes even impossible, to change or even just extend a customized protocol to support secure pre- or post-computation, which in many real-world settings seems necessary. As an example, consider the following use-case for private set intersection: a dating application would like to securely compute the intersection of two peoples' interests, and then give weights to the matched items in order to compute some expected match percentage. This requires some post-processing on the matched items, which existing customized protocols are unable to do as they reveal the items upon completion of the private set intersection protocol.

We assume a computational security parameter of $\kappa = 128$ and a statistical security parameter of $\rho = 40$. We utilize all known garbled circuit optimizations, including privacy-free garbled circuits [FNO15] for computing the predicate checks, the "half-gates" optimization [ZRE15] for reducing the size of the garbled circuit, elliptic curve cryptography for smaller public key sizes, etc. If not specified otherwise, we use $\gamma = 1250$ as the ratio between the cost of a public key operation and a symmetric key operation. (As our protocol makes heavy use of public key operations, a smaller ratio leads directly to better results for our protocol.) This number is derived from estimates using the Crypto++ benchmark [Cry] and OpenSSL, and while this is of course a rough estimate, we believe it is reasonably accurate for current systems. Note that we do not separate the cost of, e.g., fixed-base exponentiations and the exponentiate-and-multiply optimizations as discussed in Section 4, which in a real implementation would further reduce this ratio.

In what follows we show different examples where input checking improves the performance of realistic functions. To briefly summarize our findings, we find that in many applications our improvement is asymptotic, and yields up to about $56\times$ improvement in terms of computation and $80\times$ improvement in terms of communication. (The exact improvement in concrete running time will of course be a combination of these two improvements depending on the computational power of the parties and the network throughput.) Although we discuss signature checks and local computation separately, they can be used together, which makes the predicate circuit larger and our result better.

5.1 Signature Checks on Inputs

One of the main applications of our improved protocol is to efficiently check that the input of each party is correctly signed. As mentioned in the Introduction, the motivation here is that the malicious security model allows an attacker to carefully choose some fake but consistent input that helps it learn extra information from the other party, such as by supplying the full universe in a private set intersection computation to learn the other party's input. A solution to this problem using existing protocols is to maliciously compute a functionality that first checks a pre-signed signature on the input and then computes the original function if and only if the signature is valid. However, checking a signature within a garbled circuit is extremely expensive, and often more expensive than the underlying computation itself. Our protocol is particularly beneficial here, as it reduces the cost of the signature checks by $O(\rho)$ times with only a slight increase in public key operations required.

In the following, we evaluate our protocol using both "small" and "large" inputs. For computing the signature verification, we follow the hash-and-sign paradigm and first hash the input to a 512-bit digest which we verify, and use SHA-256 as the underlying hash function.

Signature checks for "small" inputs. Suppose both parties have 5000 bits of input and P_1 also has a signature on its input. The parties would like to compute a circuit with ten million (non-free) gates if P_1's input is correctly signed.[2]

In Figure 5.1, we show the improvement of this setting for various sizes of the predicate circuit, from 10^6 to 10^{12}. Particularly, we highlight three special cases, where the size of the predicate circuit corresponds to signature verification using either RSA 512, RSA 1024, or RSA 2048.[3] We obtained the sizes for these circuits using an existing circuit compiler work [KMsB13]. As we can see in Figure 5.1, for RSA 512 we are able to achieve an improvement of about $40\times$ for computation

[2]We use a computation circuit with ten million gates to be able to cover many practical circuits. Using a computation circuit with smaller size only benefits our comparison.

[3]We use an RSA-based signature scheme because this is the only signature scheme with known circuit sizes.

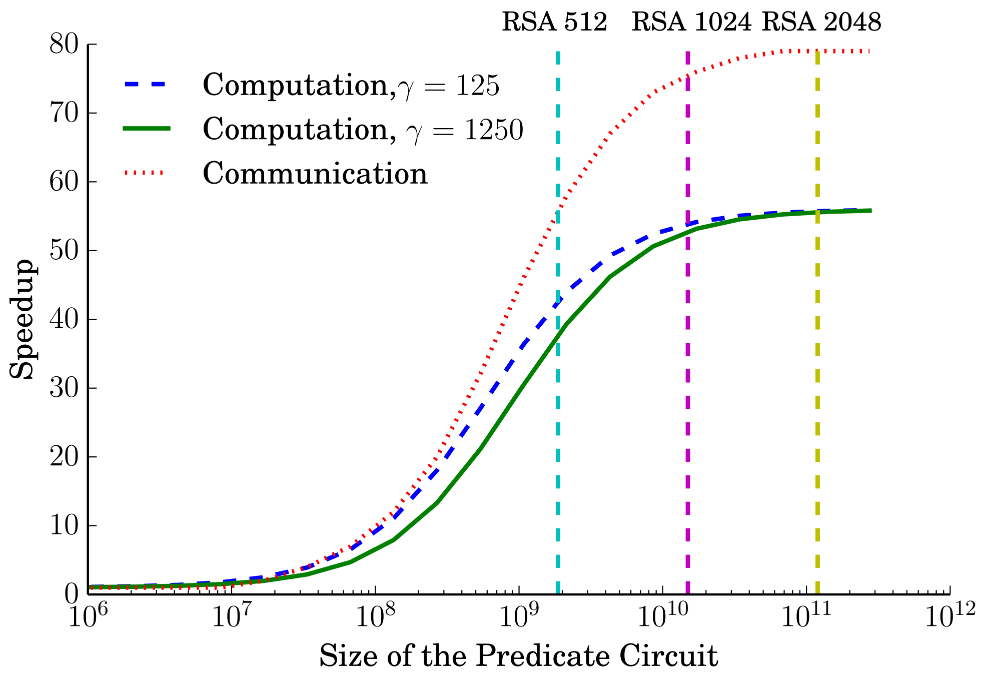


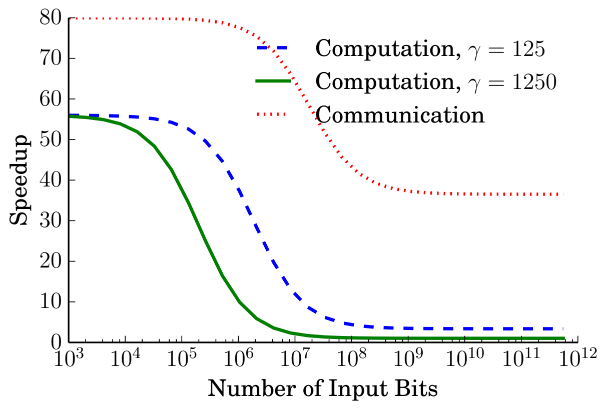
Figure 5.1: **Varying the predicate circuit size.** We fix the input size of each party to 5000 bits and the size of computation circuit $g(\cdot,\cdot)$ to ten million gates, and vary the size of the predicate circuit for party P_1. We use two ratios, $\gamma = 125$ and $\gamma = 1250$, for the public-key to symmetric-key cost. The curves represent the communication and computation improvement of our protocol compared to the generic protocol by Afshar et al., with the vertical lines denoting the sizes of the circuits for RSA 512, RSA 1024 and RSA 2048.

and 50× for communication. For a large enough predicate circuit, such as when using RSA 2048, we are able to achieve up to about 56× speedup in computation and up to about 80× speedup in communication.

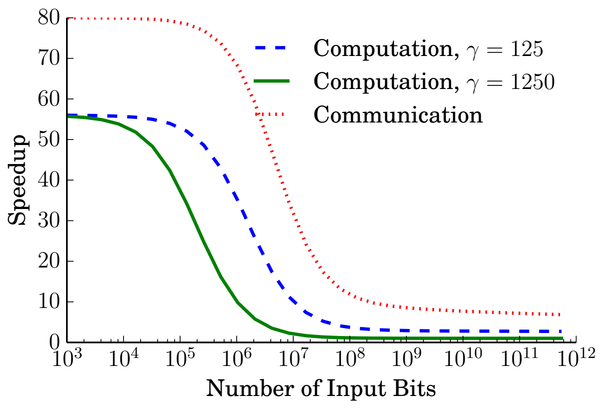
Note that these numbers agree with what we would expect asymptotically. Let $|C|$ be the size of the predicate circuit. The protocol by Afshar et al. [AMPR14] needs to perform $40 \cdot 4 \cdot |C| + 20 \cdot 4 \cdot |C| + 20 \cdot 2 \cdot |C| = 280|C|$ symmetric key operations (to garble and evaluate the circuits), and send $40 \cdot 2 \cdot |C| = 80|C|\kappa$ bits. On the other hand, our protocol only need to perform $2|C|+2|C|+|C| = 5|C|$ symmetric key operations and send $|C|\kappa$ bits when using privacy-free garbled circuits and the "half-gates" optimization. Thus, the asymptotic improvement is $280/5 = 56$ for computation and $80/1 = 80$ for communication when calculating the predicate circuit on its own. Thus, when the predicate circuit is much larger than the computation circuit, these cost dominate the overall cost and the asymptotic bound is reached.

Signature checks for "large" inputs. In Figure 5.2, we consider a similar situation as above, but here we vary the *input size* of P_1's input, using RSA 2048 as the signature scheme. In Figure 5.2a the computation circuit is of size N for N bit input, while in Figure 5.2b the computation circuit size is $N \log N$.

We can see that the improvement is about 80 for communication and about 56 for computation



(a) N sized computation circuit



(b) $N \log N$ sized computation circuit

Figure 5.2: **Varying the input size.** We fix the predicate circuit to be RSA 2048 and vary P_1's input length N from 10^3–10^{12} bits, with the size of the computation circuit based on the input size. The left graph presents the speedup versus the generic approach for a computation circuit of size N, and the right graph presents the speedup versus the generic approach for a computation circuit of size $N \log N$. We present results for both $\gamma = 125$ and $\gamma = 1250$ for the ratio of public-key to symmetric-key costs.

up to around 10^5 input bits. When the input size becomes more than 10^7 bits, the improvement for computation is less than $10\times$, and the improvement for communication reduces to about $40\times$ for the linear computation circuit and about $10\times$ for the $N \log N$ computation circuit. Note that the main reason for such a reduction is that as the number of input bits increase the cost of checking the signature becomes amortized away, in which case our improvement becomes less significant.

Note however, that (1) in both cases, our protocol never performs worse than that of Afshar et al. [AMPR14] in terms of computation and improves 10–40$\times$ in terms of communication, and (2) the reduction in the improvement only happens when the number of input bits is huge (about ten million).

5.2 Enforcing Correct Local Computation

Using local computation to reduce the cost of 2PC in the semi-honest model has been used in several existing works [HEK12, WHZ$^+$15, etc.]. Our protocol is able to provide some of these same benefits in the malicious model. Suppose two parties want to compute $f(x, y)$, which can be represented as $h_3(h_1(x), h_2(y))$, for some functions $h_1(\cdot)$, $h_2(\cdot)$, and $h_3(\cdot,\cdot)$. In the semi-honest setting, we let the parties compute $h_1(x)$ and $h_2(y)$ locally and then jointly perform a semi-honest secure computation on $h_3(\cdot,\cdot)$. Here, the bottleneck is now computing $h_3(\cdot,\cdot)$, as the other computations are all local. However, in the malicious setting, the advantage of local computation is completely lost: the result of the local computation cannot be trusted in the malicious setting. Therefore, a generic malicious protocol needs to compute a circuit that contains both local computation ($h_1(\cdot)$ and $h_2(\cdot)$) and joint computation ($h_3(\cdot,\cdot)$).

However, using our protocol, we can view predicate checking as a way to ensure that local computation is done honestly. That is, the two parties first locally compute $H_1 = h_1(x)$ and

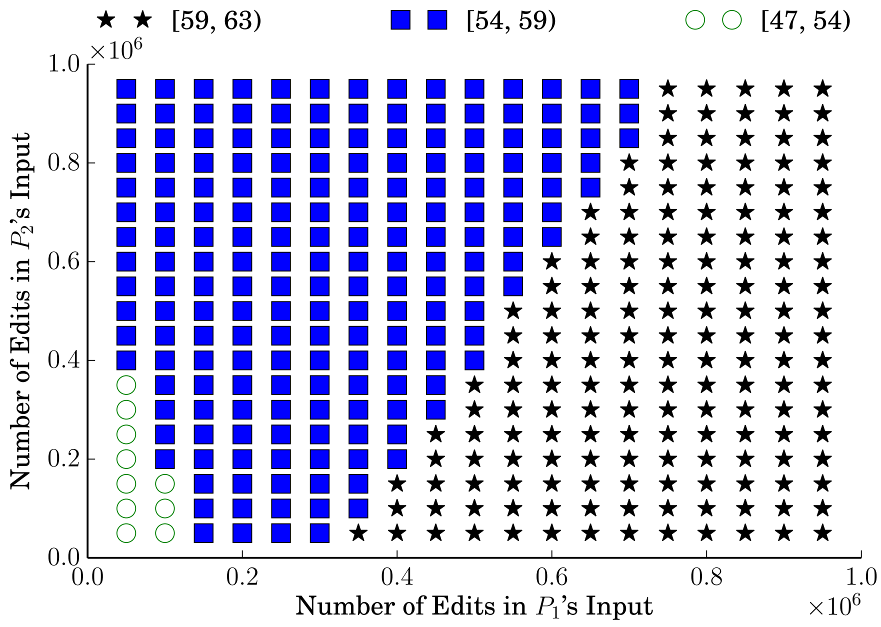


Figure 5.3: **Computation improvement for private edit distance approximation.** We vary the input size of each party and fix the ratio of public-key to symmetric-key costs to $\gamma = 1250$. ★ represents a speedup in the range $[59, 63)$, ■ represents a speedup in the range $[54, 59)$, and ○ represents a speedup in the range $[47, 54)$.

$H_2 = h_2(y)$. Then they use $H_1 \| x$ and $H_2 \| y$ as their input to the protocol, using predicate functions $f_1(H_1 \| x) := (H_1 \overset{?}{=} h_1(x))$ and $f_2(H_2 \| y) := (H_2 \overset{?}{=} h_2(y))$ and computation function $g(x, y) = h_3(H_1, H_2)$. This is particularly beneficial when there are more efficient ways of checking that, say, $H_1 \overset{?}{=} h_1(x)$, than redoing the local computation itself. For example, checking that a list of N elements is sorted takes $O(N)$ time whereas sorting a list of N elements takes $O(N \log N)$ time.

Thus, using our protocol improves over generic malicious 2PC for the following two reasons:

1. We save a factor of $O(\rho)$ on the predicate circuits used to check the local computation.

2. Since x and y are not used in the underlying computation directly, they do do not require the machinery needed to enforce input consistency. That is, we only need to ensure the consistency of $h_1(x)$ and $h_2(y)$, which can be much smaller than the original input (see the examples below for more details).

We look at three examples of protocols that can be improved using local computation: (1) private edit distance approximation, (2) solving a linear system, and (3) private set intersection.

Private edit distance approximation. Wang et al. [WHZ+15] designed an algorithm to approximate the edit distance of two genome sequences in the semi-honest setting. They proposed several optimizations that minimize the circuit for joint computation. Let N be the number of edits in the genome compared to the reference genome, and let ϵ be the relative error we want to achieve with $2^{-\delta}$ failure probability. During the local computation, each party hashes each edit to either 1 or -1 and sums them up, while the joint computation computes the square of the difference between

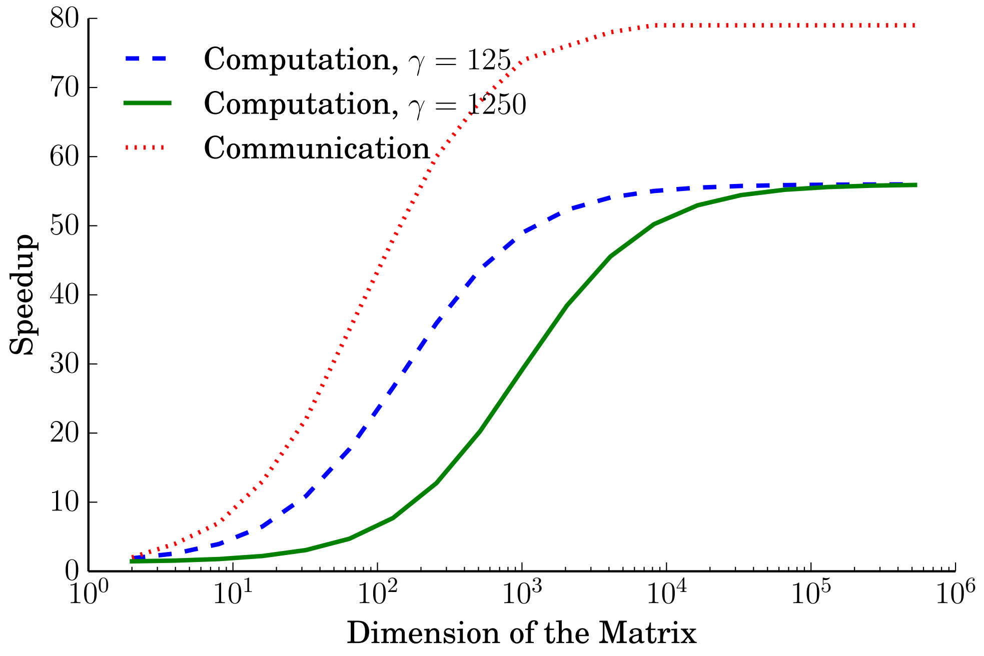


Figure 5.4: **Improvement when solving linear systems.** This graph shows the speedup in terms of computation and communication versus the naive approach when solving linear systems, where we vary P_1's input size and use $\gamma = 125$ and $\gamma = 1250$ as the ratios of public-key to symmetric-key costs.

the two sums. In order to achieve the error mentioned above, we need to compute this $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$ times, each time using a new random hash function. Therefore, local computation is on the order of $O(N/\epsilon^2 \log \frac{1}{\delta})$, while the joint computation has a circuit of size $O(\frac{\log N}{\epsilon^2} \log \frac{1}{\delta})$. Thus, whereas the generic solution in the malicious setting has a complexity of $O\left(\rho\kappa\left(\frac{N}{\epsilon^2} \log \frac{1}{\delta} + \frac{\log N}{\epsilon^2} \log \frac{1}{\delta}\right)\right)$, our protocol has only $O\left(\kappa\frac{N}{\epsilon^2} + \rho\kappa\frac{\log N}{\epsilon^2} \log \frac{1}{\delta}\right)$ complexity.

We compare the two protocols for a varying number of genome edits, based on an error rate of 1% with 95% confidence; see Figure 5.3. Our protocol achieves about 79× communication improvements for all combinations we tested, therefore we only show the computation improvement. We achieve a computation improvement up to about 63×, with the exact improvement increasing as we increase the input size of P_1 or P_2. Note that the improvement here is greater than the asymptotic bound of 56× described in Section 5.1 because here both parties do an input check while in the previous setting only P_1 did an input check. Having P_2 also do an input check leads to additional improvements.

Note that our protocol also works for other algorithms with a similar pattern as private edit distance approximation, such as heavy hitters, quantiles, etc.

Solving a linear system. Suppose P_1 holds an invertible matrix A and P_2 holds a vector b. The two parties want to securely solve the linear system $Ax = b$. A naive solution is to perform Gaussian elimination obliviously within the secure computation, which requires a circuit with $O(N^3)$ multiplications. A better solution in the semi-honest setting is to let P_1 compute A^{-1} locally so that the parties only need to perform $O(N^2)$ multiplications in the secure computation portion of the protocol.

When it comes to the malicious setting, we can check that P_1 inputs a correct inverse by checking that $A^{-1}A = I$. Applying the generic solution gives us a protocol with complexity $O(\rho\kappa N^3)$ whereas our protocol achieves a complexity of $O(\kappa N^3 + \rho\kappa N^2)$.

As shown in Figure 5.4, we achieve an improvement of $10\times$ in terms of communication when the dimension of the matrix is as small as 10. The improvement reaches the theoretical improvement calculated in Section 5.1 when the dimension increases to about one thousand. The computation improvement also behaves similarly to the previous example of checking signatures.

Private set intersection. We evaluated private set intersection following the approach of Huang et al. [HEK12]. Private set intersection has a predicate circuit of size N and a computation circuit of size $O(N \log N)$. We evaluated our protocol on this with input size up to one million and found a $1.3\times$ improvement in computation and communication. While these gains are not as great as the order-of-magnitude gains for other functions, we note that a 30% improvement in running time is still significant.

The main reason for a smaller improvement than the order-of-magnitude improvements we see in the previous examples is because the predicate circuit is of size N for N input bits while the computation circuit size is $O(N \log N)$. This means that the cost is dominated by the computation circuit and hence we get smaller gains.

Acknowledgments

Work of Alex J. Malozemoff conducted 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 supported in part by NSF award #1111599.

References

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

[Bau16] Carsten Baum. On garbling schemes with and without privacy. Cryptology ePrint Archive, Report 2016/150, 2016. http://eprint.iacr.org/.

[BGMW93] Ernest F. Brickell, Daniel M. Gordon, Kevin S. McCurley, and David Bruce Wilson. Fast exponentiation with precomputation (extended abstract). In Rainer A. Rueppel, editor, *EUROCRYPT '92*, volume 658 of *LNCS*, pages 200–207. Springer, Heidelberg, May 1993.

[Cry] Crypto++ 5.6.0 benchmarks. http://www.cryptopp.com/benchmarks.html. Accessed 2015-05-08.

[de 95] Peter de Rooij. Efficient exponentiation using procomputation and vector addition chains. In Alfredo De Santis, editor, *EUROCRYPT '94*, volume 950 of *LNCS*, pages 389–399. Springer, Heidelberg, May 1995.

[DKL+12] Ivan Damgård, Marcel Keller, Enrique Larraia, Christian Miles, and Nigel P. Smart. Implementing AES via an actively/covertly secure dishonest-majority MPC protocol. In Ivan Visconti and Roberto De Prisco, editors, *SCN 12*, volume 7485 of *LNCS*, pages 241–263. Springer, Heidelberg, September 2012.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.

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

[FJNT15] Tore Kasper Frederiksen, Thomas P. Jakobsen, Jesper Buus Nielsen, and Roberto Trifiletti. TinyLEGO: An interactive garbling scheme for maliciously secure two-party computation. Cryptology ePrint Archive, Report 2015/309, 2015. http://eprint.iacr.org/2015/309.

[FNO15] Tore Kasper Frederiksen, Jesper Buus Nielsen, and Claudio Orlandi. Privacy-free garbled circuits with applications to efficient zero-knowledge. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 191–219. Springer, Heidelberg, April 2015.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th STOC*, pages 218–229. ACM Press, May 1987.

[HEK12] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS 2012*. The Internet Society, February 2012.

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

[HKK+14] Yan Huang, Jonathan Katz, Vladimir Kolesnikov, Ranjit Kumaresan, and Alex J. Malozemoff. Amortizing garbled circuits. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 458–475. Springer, Heidelberg, August 2014.

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

[KMsB13] Ben Kreuter, Benjamin Mood, abhi shelat, and Kevin Butler. PCF: A portable circuit format for scalable two-party secure computation. In Sam King, editor, *22nd USENIX Security Symposium*, Washington, D.C., USA, August 14–16, 2013. USENIX Association.

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

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

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

[LP12] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. *Journal of Cryptology*, 25(4):680–722, October 2012.

[LR14] Yehuda Lindell and Ben Riva. Cut-and-choose Yao-based secure computation in the online/offline and batch settings. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 476–494. Springer, Heidelberg, August 2014.

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

[NO09] Jesper Buus Nielsen and Claudio Orlandi. LEGO for two-party secure computation. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 368–386. Springer, Heidelberg, March 2009.

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

[Sch90] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *CRYPTO '89*, volume 435 of *LNCS*, pages 239–252. Springer, Heidelberg, August 1990.

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

[WHZ$^+$15] Xiao Shaun Wang, Yan Huang, Yongan Zhao, Haixu Tang, Xiaofeng Wang, and Diyue Bu. Efficient genome-wide, privacy-preserving similar patient query based on private edit distance. In *ACM CCS 15*. ACM Press, 2015.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*, pages 160–164. IEEE Computer Society Press, November 1982.

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

Changelog

- Version 1.1 (February 28, 2016): Added acknowledgments.
- Version 1.0 (February 22, 2016): First release.