Faster Malicious 2-Party Secure Computation with Online/Offline Dual Execution

Peter Rindal and Mike Rosulek, Oregon State University

https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/rindal
Faster Malicious 2-party Secure Computation with Online/Offline Dual Execution

Peter Rindal∗
Oregon State University
Mike Rosulek∗
Oregon State University

Abstract
We describe a highly optimized protocol for general-purpose secure two-party computation (2PC) in the presence of malicious adversaries. Our starting point is a protocol of Kolesnikov et al. (TCC 2015). We adapt that protocol to the online/offline setting, where two parties repeatedly evaluate the same function (on possibly different inputs each time) and perform as much of the computation as possible in an offline preprocessing phase before their inputs are known. Along the way we develop several significant simplifications and optimizations to the protocol.

We have implemented a prototype of our protocol and report on its performance. When two parties on Amazon servers in the same region use our implementation to securely evaluate the AES circuit 1024 times, the amortized cost per evaluation is $5.1\text{ms offline} + 1.3\text{ms online}$. The total offline+online cost of our protocol is in fact less than the online cost of any reported protocol with malicious security. For comparison, our protocol’s closest competitor (Lindell & Riva, CCS 2015) uses $74\text{ms offline} + 7\text{ms online}$ in an identical setup.

Our protocol can be further tuned to trade performance for leakage. As an example, the performance in the above scenario improves to $2.4\text{ms offline} + 1.0\text{ms online}$ if we allow an adversary to learn a single bit about the honest party’s input with probability $2^{-20}$ (but not violate any other security property, e.g. correctness).

1 Introduction
Secure two-party computation (2PC) allows mutually distrusting parties to perform a computation on their combined inputs, while revealing only the result. 2PC was conceived in a seminal paper by Yao [34] and shown to be feasible in principle using a construction now known as garbled circuits. Later, the Fairplay project [24] was the first implementation of Yao’s protocol, which inspired interest in the practical performance of 2PC.

1.1 Cut & Choose, Online/Offline Setting
The leading technique to secure Yao’s protocol against malicious adversaries is known as cut-and-choose. The idea is to have the sender generate many garbled circuits. The receiver will choose a random subset of these to be checked for correctness. If all checked circuits are found to be correct, then the receiver has some confidence about the unopened circuits, which can be evaluated.

The cost of the cut-and-choose technique is therefore tied to the number of garbled circuits that are generated. To restrict a malicious adversary to a $2^{-s}$ chance of violating security, initial cut-and-choose mechanisms required approximately $17s$ circuits [20]. This overhead was later reduced to $3s$ circuits [21, 31, 32] and then $s$ circuits [19].

Suppose two parties wish to perform $N$ secure computations of the same function $f$ (on possibly different inputs each time), and are willing to do offline preprocessing (which does not depend on the inputs). In this online/offline setting, far fewer garbled circuits are needed per execution. The idea, due to [14, 22], is to generate many garbled circuits (enough for all $N$ executions) and perform a single cut-and-choose on them all. Then each execution of $f$ will evaluate a random subset (typically called a bucket) of the unopened circuits. Because the unopened circuits are randomly assigned to executions, only $O(s/\log N)$ circuits are needed per bucket to achieve security $2^{-s}$. Concretely, 4 circuits per bucket suffice for security $2^{-40}$ and $N = 1024$.

1.2 Dual-execution Paradigm
An alternative to cut-and-choose for malicious-secure 2PC is the dual-execution protocol of Mohassel & Franklin [25], which requires only two garbled circuits.
The idea is that two parties run two instances of Yao’s protocol, with each party acting as sender in one instance and receiver in the other. They then perform a reconciliation step in which their garbled outputs are securely compared for equality. Intuitively, one of the garbled outputs is guaranteed to be correct, so the reconciliation step allows the honest party to check whether its garbled output agrees with the correct one held by the adversary.

Unfortunately, the dual execution protocol allows an adversary to learn an arbitrary bit about the honest party’s input. Consider an adversary who instead of garbling the function $f$, garbles a different function $f'$. Then the output of the reconciliation step (secure equality test) reveals whether $f(x_1, x_2) = f'(x_1, x_2)$. However, it can be shown that the adversary can learn only a single bit, and, importantly, cannot violate output correctness for the honest party.

1.3 Reducing Leakage in Dual-execution

Kolesnikov et al. [16] proposed a combination of dual-execution and cut-and-choose that reduces the probability of a leaked bit. The idea is for each party to garble and send $s$ circuits instead of 1, and perform a cut-and-choose to check each circuit with probability $1/2$. Each circuit should have the same garbled encoding for its outputs, so if both parties are honest, both should receive just one candidate output.

However, a malicious party could cause the honest party to obtain several candidate outputs. The approach taken in [16] is to have the parties use private set intersection (PSI) to find a common value among their sets of reconciliation values. This allows the honest party to identify which of its candidate outputs is the correct one.

In Section 4 we discuss in more detail the security offered by this protocol. Briefly, an adversary cannot violate output correctness for the honest party, and learns only a single bit about the honest party’s input with probability at most $1/2^s$ (which happens only when the honest part doesn’t evaluate any correct garbled circuit).

2 Overview of Our Results

We adapt the dual-execution protocol of [16] to the online/offline setting. The result is the fastest protocol to date for 2PC in the presence of malicious adversaries. At a very high level, both parties exchange many garbled circuits in the offline phase and perform a cut-and-choose. In the online phase, each party evaluates a random bucket of its counterpart’s circuits. The parties then use the PSI-based reconciliation to check the outputs.

2.1 Technical Contributions

While the high-level idea is straightforward, some non-trivial technical changes are necessary to adapt [16] to the online/offline setting while ensuring high performance in practice.

In particular, an important part of any malicious-secure protocol is to ensure that parties use the same inputs in all garbled circuits. The method suggested in [16] is incompatible with offline pre-processing, whereas the method from [23] does not ensure consistency between circuits generated by different parties, which is the case for dual-execution (both parties generate garbled circuits). We develop a new method for input consistency that is tailored specifically to the dual-execution paradigm and that incurs less overhead than any existing technique.

In [16], the parties evaluate garbled circuits and then use active-secure private set intersection (PSI) to reconcile their outputs. We improve the analysis of [16] and show that it suffices to use PSI that gives a somewhat weaker level of security. Taking advantage of this, we describe an extremely lightweight PSI protocol (a variant of one in [30]) that satisfies this weak level of security while being round-optimal.

2.2 Implementation, Performance

We implemented a C++ prototype of our protocol using state-of-the-art optimizations, including the garbled-circuit construction of [35]: the OT-extension protocol of [15] instantiated with the base OTs of [7]. The prototype is heavily parallelized within both phases. Work is divided amongst threads that concurrently generate & evaluate circuits, allowing network throughput to be the primary bottleneck. The result is an extremely fast 2PC system. When securely evaluating the AES circuit on colocated Amazon AWS instances, we achieve the lowest amortized cost to date of 5.1ms offline + 1.3ms online per execution.

2.3 Comparison to GC-based Protocols

There have been several implementations of garbled-circuit-based 2PC protocols that achieve malicious security [1, 12, 18, 23, 29, 31, 32]. Except for [23], none of these protocols are in the online/offline settings so their performance is naturally much lower (100-1000× slower than online/offline protocols). Among them, the fastest reported secure evaluation of AES is that of [12], which was 0.46s exploiting consumer GPUs. Other protocols have been described (but not implemented) that combine cut-and-choose with the dual-execution paradigm to achieve malicious security [13, 26]. The protocol of [13] leaks more than one bit when the adversary successfully cheats during cut-and-choose.

Our protocol is most closely related to that of [23], which also achieves fast, active-secure 2PC in the online/offline setting. [23] is an implementation of the protocol of [22], and we refer to the protocol and its implementation as “LR” in this section. Both the LR protocol and ours are based on garbled circuits but use fundamentally different approaches to achieving malicious secu-
Reconciliation

| LR [23] | $|x_i(B + B')\kappa_3|$ | $|x_i|B\kappa_2$ |
|---------|-----------------|-----------------|
| Us (Async PSI) | $|x_i|B\kappa_2$ | $B^{\kappa_3}\kappa_3$ |
| Us (Sync PSI) | $B\kappa_3 + B^{2}\kappa_3$ |

Figure 1: Asymptotic communication costs of the LR protocol vs. ours (comparing online phases). $B$ is the number of circuits in a bucket; $B' \approx 3B$ is the number of auxiliary cheating-recovery circuits in [23]; $|x|$ is length of sender’s inputs; $\kappa_3$ is the statistical security parameter; $\kappa_2$ is the computational security parameter.

For clarity, we now provide a list of major differences between the two protocols.

1. LR uses a more traditional cut-and-choose mechanism where one party acts as sender and the other as receiver & evaluator. Our protocol on the other hand uses a dual-execution paradigm in which both parties play symmetric roles, so their costs are identical.

Since parties act as both sender and receiver, each party performs more work than in the traditional cut-and-choose paradigm. However, the symmetry of dual-execution means that both parties are performing computational work simultaneously, rather than idle waiting. The increase in combined work does not significantly affect latency or throughput if the communication channel is full-duplex.

2. Our protocol can provide more flexible security guarantees; in particular, it may be used with smaller parameter choices. In more detail, let $\kappa_3$ denote a statistical security parameter, meaning that the protocol ensures the adversary to completely break security with probability $1/2^{\kappa_3}$. In the LR protocol, a failure of the cut-and-choose step can violate all security properties, so the number of garbled circuits is proportional to $\kappa_3$.

Our protocol has an additional parameter $\kappa_0$, where the protocol leaks (only) a single bit to the adversary with probability $1/2^{\kappa_0}$. In our protocol (as in [16]), the number of garbled circuits is proportional to $\kappa_0$. When instantiated with $\kappa_0 = \kappa_3 = 40$, our protocol gives an equivalent guarantee to the LR protocol with $\kappa_3 = 40$. From this baseline, our protocol allows either $\kappa_3$ to be increased (strictly improving the security guarantee without involving more garbled circuits) or $\kappa_0$ to be decreased (trading performance for a small chance of a single bit leaking).\footnote{For example, two parties might want to securely evaluate AES a million times on the same secret-shared key each time, where the key is not used for anything else. In that case, a $1/2^{30}$ or $1/2^{35}$ chance of leaking a single bit about this key might be permissible.}

3. Our online phase has superior asymptotic cost, stemming from the differences in protocol paradigm — see a summary in Figure 1. LR uses a cheating-recovery phase, introduced in [19]: after evaluating the main circuits, the parties evaluate auxiliary circuits that allow the receiver to learn the sender’s input if the receiver can “prove” that the sender was cheating. Our protocol uses the PSI-based dual-execution reconciliation phase.

The important difference is that in the LR protocol, the sender’s input is provided to both the main circuits and auxiliary circuits. If there are $B$ main garbled circuits in a bucket, then there are $B' \approx 3B$ auxiliary circuits, and garbled inputs must be sent for all of them in the online phase. Each individual garbled input is sent by decommitting to an offline commitment, so it contributes to communication as well as a call to a hash function. Furthermore, the cheating-recovery phase involves decommitments to garbled outputs for the auxiliary circuits, which are again proportional to the sender’s input length.

In contrast, our protocol uses no auxiliary circuits so has less garbled inputs to send (and less associated decommitments to check). Our reconciliation phase scales only with $B$ and is independent of the parties’ input size. The overall effect is that our online phase involves significantly less communication and computation, with the difference growing as the computations involve longer inputs. With typical parameters $B = 4$ and $\kappa_3 = 40$, our reconciliation phase is cheaper whenever $|x| \geq 54$ bits. Even for the relatively small AES circuit, our protocol sends roughly $10 \times$ less data in the online phase.

4. LR’s online phase uses 4 rounds of interaction\footnote{For our purposes, a round refers to both parties sending a message. In other words, messages in the same round are allowed to be sent simultaneously, and our implementation takes advantage of full-duplex communication to reduce latency. We emphasize that synchronicity is not required for our security analysis. The protocol is secure against an adversary who waits to obtain the honest party’s message in round $i$ before sending its own round $i$ message.} and delivers output only to one party. If both parties require output, their protocol must be modified to include an additional round. Our online phase also delivers outputs to both parties using either 5 or 6 rounds (depending on the choice of PSI subprotocols). We conjecture that our protocol can be modified to use only 4 rounds, but leave that question to follow-up work.

5. Our implementation is more efficient than LR. The offline phase more effectively exploits parallelism and LR is implemented using a mix of Java & C++. The architecture of LR has a serial control flow with computationally heavy tasks performed in parallel using low level C++ code. In contrast, our protocol implementation is in C++ and fully parallelized with low level synchronization primitives.

2.4 Comparison to Non-GC Protocols

Another paradigm for malicious security in the online/offline setting is based not on garbled circuits but arithmetic circuits and secret sharing. Notable protocols and implementations falling into this paradigm include [8, 9, 10, 11, 27]. These protocols indeed have
lightweight online phases, and many instances can be batched in parallel to achieve throughput comparable to our protocol. However, all of these protocols have an online phase whose number of rounds depends on the depth of the circuit being evaluated. As a result, they suffer from significantly higher latency than the constant-round protocols in the garbled circuit paradigm like ours. The latest implementations of \([9]\) can securely evaluate AES with online latency 20ms [33]. Of special note is the implementation of the [11] protocol reported in [10], which achieves latency of only 6ms to evaluate AES. However, the implementation is heavily optimized for the special case of computing AES, and it is not clear how applicable their techniques are for general-purpose MPC. In any case, no protocol has reported online latency for AES that is less than our protocol’s total offline+online cost.

The above protocols based on secret-sharing also have significantly more expensive offline phases. Not all implementations report the cost of their offline phases, but the latest implementations of the [9] protocol require 156 seconds of offline time for securely computing AES [33]; many orders of magnitude more than ours. We note that the protocols in the secret-sharing paradigm have an offline phase which does not depend on the function that will be evaluated, whereas ours does.

3 Preliminaries

Secure computation. We use the standard notion of universally composable (UC) security \([6]\) for 2-party computation. Briefly, the protocol is secure if for every adversary attacking the protocol, there is a straight-line simulator attacking the ideal functionality that achieves the same effect. We assume the reader is familiar with the details.

We define the ideal functionality \(\mathcal{F}_{\text{multi-sfe}}\) that we achieve in Figure 2. The functionality allows parties to evaluate the function \(f, N\) times. Adversaries have the power to delay (perhaps indefinitely) the honest party’s output, which is typical in the setting of malicious security. In other words, the functionality does not provide output fairness.

Furthermore, the functionality occasionally allows the adversary to learn an arbitrary additional bit about the inputs. This leakage happens according to the distribution \(\mathcal{D}\) chosen by the adversary at setup time. The probability of a leaked bit in any particular evaluation of \(f\) is guaranteed to be at most \(\varepsilon\). Further, the leakage is “risky” in the sense that the honest party detects cheating when the leaked bit is zero.

Building blocks. In Figures 10 & 11 we define oblivious transfer (OT) and commitment functionalities that are used in the protocol. In the random oracle model, where \(H\) is the random oracle, a party can commit to \(v\) by choosing random \(r \leftarrow \{0,1\}^κ\) and sending \(c = H(r|v)\).

We use and adapt the Garbled Circuit notation and terminology of \([5]\); for a formal treatment, consult that paper. In Appendix A we define the syntax and security requirements, highlighting the differences we adopt compared to \([5]\).

4 The Dual Execution Paradigm

We now give a high-level outline of the (non-online/offline) 2PC protocol paradigm of [16], which is the starting point for our protocol. The protocol makes use of a two-phase PSI subprotocol. In the first phase, both parties become committed to their PSI inputs; in the second phase, the PSI output is revealed. This component is modeled in terms of the \(\mathcal{F}_{\psi}^{\text{multi-sfe}}\) functionality in Figure 12.

Assume the parties agree on a function \(f\) to be evaluated on their inputs. The protocol is symmetric with respect to Alice and Bob, and for simplicity we describe only Alice’s behavior.

1. Alice generates \(κ\) garbled circuits computing \(f\), using a common garbled output encoding for all of them.
2. Alice announces a random subset of Bob’s circuits to open. However, the actual checking of the circuits is delayed until later in the protocol.
3. Alice uses OT to receive garbled inputs for the circuits generated by Bob, as in Yao’s protocol. Alice sends the garbled circuits she generated, along with her own garbled input for these circuits.
4. Alice evaluates the garbled circuits received from Bob. If Bob is honest, then all of his circuits use the same garbled output encoding and Alice will receive the same garbled output from each one. But in the general case, Alice might obtain several inconsistent garbled outputs.
5. Assume that Alice can decode the garbled outputs to obtain the logical circuit output. For each candidate circuit output \(y\) with garbled encoding \(Y^b_y\) (for a garbled output under Bob’s encoding), let \(Y^a_y\) denote the encoding of \(y\) under Alice’s garbled output encoding (which Alice can compute). Interpreting \(Y^a_y\) and \(Y^b_y\) as sets of individual wire labels, let \(R_y\) be the XOR of all items in \(Y^a_y \cup Y^b_y\), which we write as \(R_y = \bigoplus (Y^a_y \cup Y^b_y)\) and which we call the reconciliation value for \(y\). Alice sends the set of all \(\{R_y\}\) values as input to a PSI instance.
6. With the PSI inputs committed, the parties open and check the circuits chosen in the cut-and-choose step. They abort if any circuit is not correctly garbled, or the circuits do not have consistent garbled output encodings.
7. The parties release the PSI output. Alice aborts if the PSI output is not a singleton set. Otherwise, if
the output is \( \{R^*\} \) then Alice outputs the value \( y \) such that \( R^* = R_y \).

4.1 Security Analysis and Other Details

Suppose Alice is corrupt and Bob is honest. We will argue that Alice learns nothing beyond the function output, except that with probability \( 2^{-k_6} \) she learns a single bit about Bob’s input.

Suppose Alice uses input \( x_1 \) as input to the OTs, and Bob has input \( x_2 \). Since Bob’s circuits are honestly generated and use the same garbled output encoding, every circuit evaluated by Alice leads to the same garbled output \( Y_y^b \) that encodes logical value \( y^* = f(x_1, x_2) \). Note that by the authenticity property of the garbled circuits, this is the only valid garbled output that Alice can predict.

Since Alice may generate malicious garbled circuits, honest Bob may obtain several candidate outputs from these circuits. Bob’s input to the PSI computation will be a collection of reconciliation values, each of the form \( R_y^* = \bigoplus (Y_y^a \cup Y_y^b) \).

At the time of PSI input, none of Bob’s (honestly) garbled circuits have been opened, so they retain their authenticity property. Then Alice cannot predict any valid reconciliation value except for this \( R_y^* \). This implies that the PSI output will be either \( \{R_y^*\} \) or \( \emptyset \). In particular, Bob will either abort or output the correct output \( y^* \). Furthermore, the output of the PSI computation can be simulated knowing only whether honest Bob has included \( R_y^* \) in his PSI input.

The protocol includes a mechanism to ensure that Alice uses the same \( x_1 \) input for all of the garbled circuits. Hence, if Bob evaluates at least one correctly generated garbled circuit, it will give output \( y^* \) and Bob will surely include the \( R_y^* \) reconciliation value in his PSI input. In that case, the PSI output can be simulated as usual.

The probability the Alice manages to make Bob evaluate no correctly generated garbled circuits is \( 2^{-k_6} \) — she would have to completely predict Bob’s cut-and-choose challenge to make all opened circuits correct and all evaluated circuits incorrect. But even in this event, the simulator only needs to know whether \( f'(x_1, x_2) = y^* \) for any of the \( f' \) computed by Alice’s malicious garbled circuits. This is only one bit of information about \( x_2 \) which the simulator can request from the ideal functionality.

4.2 Outline for Online/Offline Dual-Execution

Our high-level approach is to adapt the [16] protocol to the online/offline setting. The idea is that the two parties plan to securely evaluate the same function \( f \), \( N \) times, on possibly different inputs each time. In preparation they perform an offline pre-processing phase that depends only on \( f \) and \( N \), but not on the inputs. They generate many garbled circuits and perform a cut-and-choose on all of them. Then the remaining circuits are assigned randomly to buckets. Later, once inputs are known in the online phase, one bucket’s worth of garbled circuits are consumed for each evaluation of \( f \).

Our protocol will leak a single bit about the honest party’s input only when a bucket contains no “good” circuit from the adversary (where “good” is the condition that is verified for opened circuits during cut-and-choose). Following the lead of [23], we focus on choosing the number of circuits so that the probability of such an event in any particular bucket is \( 2^{-k_6} \). We note that the analysis of parameters in [14, 22] considers an overall cheating condition, i.e., that there exists a bucket that has no “good” circuits, which leads to slightly different numbers.

**Lemma 1 ([23])**. If the parties plan to perform \( N \) exe-
cutions, using a bucket of $B$ circuits for each execution and a total of $N \geq NB$ garbled circuits generated for the overall cut-and-choose, then the probability that a specific bucket contains no good circuit is at most:

$$\max_{r \in \{B, \ldots, NB\}} \left\{ \binom{\tilde{N} - r}{(\tilde{N} - B)/B} \cdot \binom{r}{B} \right\}.$$ 

Suppose the parties will perform $N$ executions, using buckets of size $B$ in the online phase, and wish for $2^{-k_0}$ probability of leakage. We can use the formula to determine the smallest compatible $\tilde{N}$. In the full version we show all reasonable parameter settings for $k_0 \in \{20, 40, 80\}$ and $N \in \{8, 16, 32, \ldots, 32768\}$.

By adapting [16] to the online/offline setting, we obtain the generic protocol outlined in Figure 3. Even with pre-processing, an online OT requires two rounds, one of which can be combined with the direct sending of garbled inputs. The protocol therefore requires three rounds plus the number of rounds needed for the PSI subprotocol (at least two).

4.3 Technicalities

We highlight which parts of the [16] protocol break down in the online/offline setting and require technical modification:

Same garbled output encoding. In [16] each party is required to generate garbled circuits that have a common output encoding. Their protocol includes a mechanism to enforce this property. In our setting, we require each bucket of circuits to have the same garbled output encoding. But this is problematic because in our setting a garbled circuit is generated before the parties know which bucket it will be assigned to.

Our solution is to have the garbler provide for each bucket a translation of the following form. The garbler chooses a bucket-wide garbled output encoding, e.g., for the first output wire, he chooses wire labels $W_0^f, W_1^f$ encoding false and true, respectively. Then if $W_0^j, W_1^j$ are the output wire labels already chosen for the $j$th circuit in this bucket, the garbler is supposed to provide translation values $W_v^j \oplus W_v^f$ for $v \in \{0, 1\}$. After evaluating, the receiver will use these values to translate the garbled input to this bucket-wide encoding that is used for PSI reconciliation.

Of course, a cheating party can provide invalid translation values. So we use step 3 of the online phase (Figure 3) to check them. In more detail, a sender must commit in the offline phase to the output wire labels of every garbled circuit. These will be checked if the circuit is chosen in the cut-and-choose. In step 3 of the online phase, these commitments are opened so that the receiver can check the consistency of the translation values (i.e., whether they map to a hash of the common bucket-wide encoding provided during bucketing.). This step reveals all of the bucket-wide encoding values, making it now easy for an adversary to compute any reconciliation value. This is why we employ a 2-phase PSI protocol, so that PSI inputs are committed before these translation values are checked.

Adaptive garbling. Standard security definitions for garbled circuits require the evaluator to choose the input before the garbled circuit is given. However, the entire purpose of offline pre-processing is to generate & send the garbled circuits before the inputs are known. This requires the garbling scheme to satisfy an appropriate adaptive security property, which is common to all works in the online/offline setting [14, 22]. See Appendix A for details.

---

**Offline phase:**

1. Parties perform offline preprocessing for the OTs that will be needed, and for the PSI subprotocol, if appropriate.
2. Based on $N$ and $\kappa_0$, the parties determine appropriate parameters $\tilde{N}, B$ according to the discussion in Section 4.2. Each party generates and sends $\tilde{N}$ garbled circuits, and chooses a random subset of $\tilde{N} - NB$ of their counterpart’s circuits to be opened. The chosen circuits are opened and parties abort if circuits are found to be generated incorrectly.
3. Each party randomly assigns their counterpart’s circuits to buckets of size $B$. Each online execution will consume one bucket’s worth of circuits.

**Online phase:**

1. Parties exchange garbled inputs: For one’s own garbled circuits in the bucket, a party directly sends the appropriate garbled inputs; for the counterpart’s garbled circuits, a party uses OT as a receiver to obtain garbled inputs as in Yao’s protocol.
2. Parties evaluate the garbled circuits and compute the corresponding set of reconciliation values. They commit their sets of reconciliation values as inputs to a PSI computation.
3. With the PSI inputs committed, the parties open some checking information (see text in Section 4.3) and abort if it is found to be invalid.
4. The parties release the PSI output and abort if the output is \emptyset. Otherwise, they output the plaintext value whose reconciliation value is in the PSI output.

Figure 3: High-level outline of the online/offline, dual-execution protocol paradigm.
5 Input Consistency

In this section we describe a new, extremely lightweight input-consistency technique that is tailored for the dual-execution paradigm.

5.1 Consistency Between Alice’s & Bob’s Circuits

We start with the “classical” dual-execution scenario, where Alice and Bob each generate one garbled circuit. We describe how to force Alice to use the same input in both of these garbled circuits (of course, the symmetric steps are performed for Bob). The high-level idea is to bind her behavior as OT receiver (when obtaining garbled inputs for Bob’s circuits) to the commitments of her garbled inputs in her own circuits.

It is well-known [2] that oblivious transfers on random inputs can be performed offline, and later “derandomized” to OTs of chosen inputs. Suppose two parties perform a random string OT offline, where Alice receives \( c, m_c \) and Bob receives \( m_0, m_1 \), for random \( c \in \{0, 1\} \) and \( m_0, m_1 \in \{0, 1\}^k \). Later when the parties wish to perform an OT of chosen inputs \( c^* \) and \( (m^0_c, m^1_c) \). Alice can send \( d = c \oplus c^* \) and Bob can reply with \( m^0_c \oplus m_d \) and \( m^1_c \oplus m_1 \oplus d \).

In the offline phase of our protocol, the parties perform a random OT for each Alice-input wire of each circuit, where Alice acts as the receiver. These will be later used for Alice to pick up her garbled input for Bob’s circuit. Let \( c \) denote the string denoting Alice’s random choice bits for this collection of OTs.

Also in the offline phase, we will have Alice commit to all of the possible garbled input labels for the circuits that she generated. Suppose she commits to them in an order determined by the bits of \( c \); that is, the wire label commitments for the first input wire are in the order (false,true) if the first bit of \( c \) is 0 and (true,false) otherwise.

In the online phase with input \( x \), Alice sends the OT “derandomized” message \( d = x \oplus c \). She also sends her garbled inputs for the circuits she generated by opening the commitments indexed by \( d \); that is, she opens the first or second wire label of the \( i \)th pair, depending on whether \( d_i = 0 \) or \( d_i = 1 \), respectively. Bob will abort if Alice does not open the correct commitments.

Alice’s effective OT input is \( x = d \oplus c \), so she picks up garbled input corresponding to \( x \). If Alice did indeed commit to her garbled inputs arranged according to \( c \), then she opens the commitments whose truth values are also \( x = d \oplus c \). More formally,

### Offline:

Alice garbles the labels \( A_0, A_1 \) and Bob garbles \( B_0, B_1 \) for Alice’s input. Alice receives OT message \( m_c \) and Bob holds \( m_0, m_1 \). Alice sends \((\text{COMMIT}, (\text{sid}, i), A_{\text{side}})\) to \( F_{\text{com}} \) for \( i \in \{0, 1\} \).

### Online:

Alice sends \( d = c \oplus x \) to Bob and \((\text{OPEN}, (\text{sid}, d))\) to \( F_{\text{com}} \). Bob receives \((\text{OPEN}, (\text{sid}, d), A_0)\) from \( F_{\text{com}} \) and sends \((B_0 \oplus m_d, B_1 \oplus m_{1 \oplus d})\) to Alice who computes \( B_c = m_c \oplus (B_{\text{side}} \oplus m_c) \).

Figure 4: Input consistency on a single bit of Alice’s input for “classic” dual-execution.

Looking ahead, we will use cut-and-choose to guarantee that there is at least one circuit for which Alice’s garbled input commitments are correct in this way.

5.2 Aggregating Several OTs

In our protocol, both parties evaluate a bucket of several circuits. Within the bucket, each of Alice’s circuits is paired with one of Bob’s, as above. However, this implies that Alice uses separate OTs to pick up her garbled inputs in each of Bob’s circuits. To address this, we aggregating several OTs together to form a single OT.

Suppose Alice & Bob have performed two random string OTs, with Alice receiving \( c, c', m_c, m_c' \), and Bob receiving \( m_0, m_1, m'_0, m'_1 \), for random \( c, c' \in \{0, 1\} \). Suppose further that Alice sends \( \delta = c \oplus c' \) to Bob in an offline phase. To aggregate these two random OTs into a single chosen-input OT with inputs \( c^*, m^0_c, m^1_c \). Alice can send \( d = c \oplus c^* \), and Bob can reply with \( m^0_c \oplus (m_d \oplus m^1_c) \).

The idea extends to aggregate any number \( B \) of different random OTs into a single one, with Alice sending \( B - 1 \) different \( \delta \) difference values. In our protocol, we aggregate in this way the OTs for the same wire across different circuits. Intuitively, Alice either receives wire labels for the same value on each of these wires (by reporting correct \( \delta \) values), or else she receives nothing for this wire on any circuit.

5.3 Combining Everything with Cut-and-Choose

Now consider a bucket of \( B \) circuits. In the offline phase Alice acts as receiver in many random OTs, one collection of them for each of Bob’s circuits. Let \( c_j \) be her (string of) choice bits for the OTs associated with the \( j \)th circuit. Alice is then supposed to commit to the garbled inputs of her \( j \)th circuit arranged according to \( c_j \). Bob will check this property for all circuits that are opened during the cut-and-choose phase by Alice showing the
corresponding OT messages. Hence with probability at least $1 - 2^{-k_0}$, at least one circuit in any given bucket has this property. Alice also reports aggregation values $\delta_j = c_1 \oplus c_j$ for these OTs.

In the online phase Alice chooses her input $x$ and sends $d_1 = c_1 \oplus x$ as the OT-derandomization message. This is equivalent to Alice sending $d_j = \delta_j \oplus d_1$ as the message to derandomize the $j$th OTs. To send her garbled input for the $j$th circuit, Alice is required to open her commitments indexed by $d_j$.

If Alice lies in any of the aggregation strings, then she will be missing at least one of the $B$-out-of-$B$ secret shares which mask her possible inputs. Intuitively, Alice’s two strategies are either to provide honest aggregation strings or not obtain any garbled inputs in the position that she lied. In the latter case, the simulator can choose an arbitrary input for Alice in that position.

If we then consider the likely case where Bob’s $j$th circuit is “good” and Alice provided honest aggregation strings, then Alice will have decommitted to inputs for the $j$th circuit that are consistent with her effective OT input $x^*_1$. From the discussion in Section 4.1, this is enough to guarantee that the reconciliation phase leaks nothing.

Even if there are no “good” circuits in the bucket (which happens with probability $1/2^{k_0}$), it is still the case that Alice learns no more than if she had received consistent garbled input $x^*_1$ for all of Bob’s circuits. So the reconciliation phase can be simulated knowing only whether Bob evaluates any circuit resulting in $f(x^*_1, x_2)$. This is a single bit of information about Bob’s input $x_2$.

6 Selective Failure Attacks

In the garbled circuit paradigm, suppose Alice is acting as evaluator of some garbled circuits. She uses OT to pick up the wire labels corresponding to her input. A corrupt Bob could provide incorrect inputs to these OTs, so that (for instance) Alice picks up an invalid garbled input if and only if the first bit of her input is 0. By observing whether the evaluator aborts (or produces otherwise unexpected behavior), Bob can deduce the first bit of Alice’s input. This kind of attack, where the adversary causes the honest party to abort/fail with probability depending on its private input is called a selective failure attack.

A common way to prevent selective failure is to use what is called a $k$-probe-resistant input encoding:

**Definition 2** ([20, 32]). Matrix $M \in \{0, 1\}^{k \times n}$ is called $k$-probe resistant if for any $L \subseteq \{1, 2, \ldots, n\}$, the Hamming distance of $\bigoplus_{i \in L} M_i$ is at least $k$, where $M_i$ denotes the $i$th row vector of $M$.

The idea is for Alice to choose a random encoding $\tilde{x}_1$ of her logical input $x_1$ satisfying $M \tilde{x}_1 = x_1$. Then the parties evaluate the function $f'(\tilde{x}_1, x_2) = f(M \tilde{x}_1, x_2)$. This additional computation of $M \tilde{x}_1$ involves only XOR operations, so it does not increase the garbled circuit size when using the Free-XOR optimization [17] (it does increase the number OTs needed).

Alice will now use $\tilde{x}_1$ as her choice bits to the OTs. The adversary can probe any number of bits of $\tilde{x}_1$, by inserting invalid inputs to the OT in those positions, and seeing whether the other party aborts. For each position probed, the adversary incurs a $1/2$ probability of being caught.4

The property of $k$-probe-resistance implies that probing $k$ bits of the physical input $\tilde{x}_1$ leaks no information about the logical input $M \tilde{x}_1$. However, probing $k$ bits incurs a $1 - 2^{-k}$ probability of being caught. Hence, our protocol requires a matrix that is $k_0$-probe resistant, where $k_0$ is the statistical security parameter. We refer the reader to [23] for the construction details of $k$-probe resistant matrices and their parameters.

6.1 Offline the $k$-probe computations

Using $k$-probe-resistant encodings, the encoded input $\tilde{x}_1$ is significantly longer than the logical input $x_1$. While the computation of $M \tilde{x}_1$ within the garbled circuit can involve no cryptographic operations (using Free-XOR), it still involves a quadratic number of XOR operations.

Lindell & Riva [22] suggest a technique that moves these computations associated with $k$-probe-resistant encodings to the offline phase. The parties will compute the related function $f'((\tilde{x}_1, c), x_2) = f(\tilde{x}_1 \oplus Mc, x_2)$. In the offline phase, Alice will use OT to obtain wire labels for a random string $c$. She can also begin to partially evaluate the garbled circuit, computing wire labels for the value $Mc$.

In the online phase, Alice announces $\hat{x}_1 = x_1 \oplus Mc$ where $x_1$ is her logical input. Then Bob directly sends the garbled inputs corresponding to $\hat{x}_1$. This introduces an asymmetry into our input consistency technique. The most obvious solution to maintain compatibility is to always evaluate circuits of the form $f'(\hat{x}_1, c_1, \hat{x}_2, c_2) = f(\hat{x}_1 \oplus Mc_1, \hat{x}_2 \oplus Mc_2)$, so that Alice uses the same physical input $(c_1, \hat{x}_1)$ in both hers and Bob’s circuits. However, we would prefer to let Alice use logical input $x_1$ rather than its (significantly longer) $k$-probe-encoded input, to reduce the concrete overhead. It turns out that we can accommodate this by exploiting the $Z_2$-linearity of the encoding/decoding operation.

Consider a bucket of circuits $\{1, \ldots, B\}$. For the $j$th
circuit, Alice acts as receiver in a set of random OTs, and receives random choice bits $c_j$. The number of OTs per circuit is the number of bits in a $k$-probe-resistant encoding of Alice’s input.

For Alice’s $j$th circuit, she must commit to her garbled inputs in the order given by the string $Mc_j$ (rather than just $c_j$ as before). This condition will be checked by Bob in the event that this circuit is opened during cut-and-choose. To assemble a bucket, Alice reports aggregation values $\delta_j = c_1 \oplus c_j$ as before. Imagine Alice derandomizing these OTs by sending an all-zeroses derandomization message. This corresponds to her accepting the random $c_1$ as her choice bits. (Of course, an all-zeroses message need not be actually sent.) Bob responds and uses the aggregated OTs to send Alice the garbled inputs for $c_1$ for all of his garbled circuits (indeed, even in the $j$th circuit Alice receives garbled inputs corresponding to $c_1$).

In the online phase, Alice decides her logical input $x_1$, and she sends $\hat{x}_1 = Mc_1 \oplus x_1$. This value derandomizes the offline $k$-probe-resistant encoding. Then in her own $j$th circuit, Alice must open the garbled input commitments indexed by the (public) string $\hat{x}_1 \oplus M\delta_j$.

To see why this solution works, suppose that Alice’s $j$th circuit is “good” (i.e., garbled correctly and input commitments arranged by $Mc_j$). As before, define her effective OT input to the $j$th OTs as $c^* = c_j \oplus \delta_j$ (which should be $c_1$ if Alice did not lie about $\delta_j$). Even if Alice lied about the $\delta$ values she surely learns no more than she would have learned by being truthful about the $\delta$ values and using effective input $c^*$ in all OTs. Hence, we can imagine that she uses logical input $x^*_1 = \hat{x}_1 \oplus Mc^*$ in all of Bob’s garbled circuit.

Alice is required to open garbled inputs indexed by $\hat{x}_1 \oplus M\delta_j = \hat{x}_1 \oplus M(c^* \oplus c_j) = x^*_1 \oplus Mc_j$. These are exactly the garbled inputs corresponding to logical input $x^*_1$, since the commitments were arranged according to $Mc_j$. We see that Bob evaluates at least one correctly garbled circuit with Alice using input $x^*_1$, which is all that is required for weak input consistency.

7 Optimizing PSI Reconciliation

7.1 Weaker security.

Our main insight is that our PSI reconciliation step does not require a fully (UC) secure PSI protocol. Instead, a weaker security property suffices. Recall that the final steps of the [16] protocol proceed as follows:

- Alice & Bob commit to their PSI inputs.
- The garbled-output translations are opened and checked.
- The parties either abort or release the PSI output.

For simplicity, assume for now that only one party receives the final PSI output. We will address two-sided output later.

Suppose Alice is corrupt and Bob is honest. Following from the discussion of security in Section 4, Bob will use as PSI input a collection of valid reconciliation values. At the time Alice provides her PSI inputs, the authenticity property of the garling scheme is in effect. This means that Alice can predict a valid reconciliation value only for the “correct” output $y^*$. All other valid reconciliation values that might be part of Bob’s PSI input are unpredictable.

Below we formalize a weak notion of security for input distributions of this form:

**Definition 3.** Let $\Pi$ be a two-phase protocol for set intersection ($\mathcal{F}^{n,\ell}_{\text{psi}}$, Figure 12). We say that $\Pi$ is weakly malicious-secure if it achieves UC-security with respect to environments that behave as follows:

1. The adversary sends a value $a^* \in \{0,1\}^\mathcal{L}$ to the environment along with the description of a distribution $D$ whose support is cardinality-$(n - 1)$ subsets of $\{0,1\}^\mathcal{L}$. We further require that $D$ is unpredictable in the sense that the procedure “$A \leftarrow D$; output a uniformly chosen element of $A$” yields the uniform distribution over $\{0,1\}^\mathcal{L}$ (the joint distribution of all elements of $A$ need not be uniform).

2. The environment (privately) samples $A \leftarrow D$ and gives input $A \cup \{a^*\}$ to the honest party for the first phase of PSI.

3. After the first phase finishes (i.e., both parties’ inputs are committed), the environment gives the coins used to sample $A$ to the adversary.

4. The environment then instructs the honest party to perform the second phase of PSI to obtain output.

In this definition, the adversary knows only one value in the honest party’s set, while all other values are essentially uniform. We claim that when $\ell$ is large, the simulator for this class of environments does not need to fully extract the adversary’s PSI input! Rather, the following are enough to ensure weakly-malicious security:

- The adversary is indeed committed to some (unknown to the simulator) effective input during the commit phase.
- The simulator can test whether the adversary’s effective PSI input contains the special value $a^*$. With overwhelming probability, no effective input element other than $a^*$ can contribute to the PSI output. Any other values in the adversary’s effective input can simply be ignored; they do not need to be extracted.

For technical reasons and convenience in the proof, we have the environment give the adversary the coins used to sample $A$, but only after the PSI input phase.

7.2 PSZ protocol paradigm.

We now describe an inexpensive protocol paradigm for PSI, due to Pinkas et al. [30]. Their protocol is proven
secure only against passive adversaries. We later discuss how to achieve weak malicious security.

The basic building block is a protocol for private equality test (PEQT) based on OT. A benefit of using OT-based techniques is that the bulk of the effort in generating OTs can be done in the offline phase, again leading to a lightweight online phase for the resulting PSI protocol.

Suppose a sender has input $s$ and receiver has input $r$, with $r, s \in \{0, 1\}^n$, where the receiver should learn whether $r = s$ (and nothing more). The PEQT protocol requires $n$ string OTs; in the $i$th one, the receiver uses choice bit $r[i]$ and the sender chooses random string inputs $(m'_0, m'_1)$. The sender finally sends $S = \bigoplus_i m'_i[j]$, and the receiver checks whether $S = \bigoplus_i m'_i[j]$, which is the XOR of his OT outputs.

The PEQT can be extended to a private set membership test (PSMT), in which the sender has a set $\{s^1, \ldots, s^t\}$ of strings, and receiver learns whether $r \in \{s^1, \ldots, s^t\}$. We simply have the sender randomly permute the $s^j$ values, compute for each one $S^j = \bigoplus_i F(m'^i[j], j)$ and send $\{S^1, \ldots, S^t\}$, where $F$ is a PRF.\(^5\) The receiver can check whether $\bigoplus_i F(m'^i[r], j)$ matches $S^j$ for any $j$. Finally, we can achieve a PSI where the receiver has strings $\{r^1, \ldots, r^t\}$ by running independent PSMTs of the form $r^j \in \{s^1, \ldots, s^t\}$ for each $r^j$ (in random order).

The overhead of this approach is $O(r^2)$, and [30] describes ways to combine hashing with this basic PSI protocol to obtain asymptotically superior PSI protocols for large sets. However, we are dealing with very small values of $t$ (typically at most 5), so the concrete cost of this simple protocol is very low.

To make the PSI protocol two-phase, we run the OTs and commit to the $S$ values in the input-committing phase. Then the output phase consists simply of the sender opening the commitments to $S$.

### 7.3 Achieving weakly-malicious security and double-sided output.

We use the [30] protocol but instantiate it with malicious-secure OTs. This leads to the standard notion of security against an active receiver since the simulator can extract the receiver’s input from its choice bits to the OTs.

However, the protocol does not achieve full security against a malicious sender. In the simple PEQT building block, the simulator cannot extract a malicious sender’s input. Doing so would require inspecting $S, \{m'_i\}$ and determining a value $s$ such that $S = \bigoplus_i m'_i[j]$. Such an $s$ may not exist, and even if it did, the problem seems closely related to a subset-sum problem.

However, if the simulator knows a candidate $s^*$, it can certainly check whether the corrupt sender has sent the corresponding $S$ value. This is essentially the only property required for weakly malicious security.

We note that a corrupt sender could use inconsistent sets $\{s^1, \ldots, s^t\}$ in the parallel PSMT instances. However, the simulator can still extract whether the candidate $s^*$ was used in each of them. If the sender used $s^*$ in $r$ of the $t$ subprotocols, then the simulator can send $s^*$ to the ideal PSI functionality with probability $r^t/t$, which is a sound simulation for weakly-malicious security.

Regarding double-sided output, it suffices to simply run two instances of the one-sided-output PSI protocol, one in each direction, in parallel. Again, this way of composing PSI protocols is not sound in general, but it is sound for the special case of weakly-malicious security.

### 7.4 Trading computation for lower round complexity.

Even when random OTs are pre-processed offline, the PSI protocol as currently described requires two rounds to commit to the outputs, and one round to release the output. The two input-committing rounds are (apparently) inherently sequential, stemming from the sequential nature of OT derandomization.

In terms of round complexity, these two PSI rounds are a bottleneck within the overall dual-execution protocol. We now describe a variant of the PSI protocol in which the two input-committing messages are asynchronous and can be sent simultaneously. The modified protocol involves (a nontrivial amount of) additional computation but reduces the number of rounds in the overall 2PC online phase by one. This tradeoff does not always reduce the overall latency of the 2PC online phase — only sometimes, depending on the number of garbled circuits being evaluated and the network latency. The specific break-even points are discussed in Section 9.

In our PEQT protocol above, the two parties have pre-processed random OTs, with choice bits $c$ and random strings $m'_0, m'_1$. To commit to his PSI input, the receiver’s first message is $d = c \oplus r$, to which the sender responds with $S = \bigoplus_i m'_i[d[i] \oplus s^i]$.

Consider randomizing the terms of this summation as $S = \bigoplus_i m'_i[d[i] \oplus z_i] \oplus z_i$ where $z_i$ are random subject to $\bigoplus_i z_i = 0$. Importantly, (1) each term in this sum depends only on a single bit of $d$; (2) revealing all terms in the sum reveals no more than $S$ itself. We let the sender commit to all the potential terms of this sum and reveal them individually in response to $d$. In more detail, the sender commits to the following values (in this order):

\[
\begin{align*}
&[m'_1[1] \oplus z_1] & [m'_2[1] \oplus z_2] & \cdots & [m'_{n}[1] \oplus z_n] \\
&[m'_1[2] \oplus z_1] & [m'_2[2] \oplus z_2] & \cdots & [m'_{n}[2] \oplus z_n] \\
&[m'_1[3] \oplus z_1] & [m'_2[3] \oplus z_2] & \cdots & [m'_{n}[3] \oplus z_n] \\
&\vdots & \vdots & \ddots & \vdots \\
&[m'_1[n] \oplus z_1] & [m'_2[n] \oplus z_2] & \cdots & [m'_{n}[n] \oplus z_n]
\end{align*}
\]

\(^5\)Simply XORing the $m'_i$ values would reveal some linear dependencies; applying a PRF renders all of the $S^j$ values independently random except the ones for which $r = s^j$. 

---

306 25th USENIX Security Symposium USENIX Association
Importantly, these commitments can be made before \( d \) is known. In response to the message \( d \) from the receiver, the sender is expected to release the output by opening the commitments indexed by the bits of \( d \). The sender will open the commitments \( \{m_i \mid \oplus x_i \oplus z_i \} \); the receiver will compute their XOR \( S \) and proceed as before.

The simulator for a corrupt sender simulates a random message \( d \) and then checks whether the sender has used a candidate input \( s^* \) by extracting the commitments indexed by \( d \) to see whether their XOR is \( \bigoplus_i m_i \oplus s^* \).\(^6\)

We can further move the commitments to the offline phase, since there are two commitments per bit of \( s \) per PEQT. Observe that the commitments in (⋆) are arranged according to the bits of \( s \), which are not known until the online phase. Instead, in the offline phase the sender can commit to these values arranged according to a random string \( \pi \). In the online phase, the sender commits to its input \( s \) by sending \( s \oplus \pi \). Then in response to receiver message \( d \), the sender must open the commitments indexed by the bits of \( d \oplus (s \oplus \pi) \).

When extending the asynchronous PEQT to a PSMT protocol, the sender commits to an array of \( F(m_i, j) \oplus z^j_i \) values for each \( j \).

### 7.5 Final Protocols

For completeness, we provide formal descriptions of the final PSI protocols (synchronous 3-round and asynchronous 2-round) in Figures 13 & 14.

We defer the proof of their security to the full version.

**Theorem 4.** The protocols \( \Pi_{\text{async-psi}} \) and \( \Pi_{\text{async-psi}} \) described in Figures 13 & 14 are weakly-malicious secure (in the sense of Definition 3) when \( \ell \geq \kappa_r \), the statistical security parameter.

### 8 Protocol Details & Implementation

The full details of our protocol are given in Figure 15 and the C++ implementation may be found at https://github.com/osu-crypto/batchDualEx. The protocol uses three security parameters:

- \( \kappa_b \) is chosen so that the protocol will leak a bit to the adversary with probability at most \( 2^{-\kappa_b} \). This parameter controls the number of garbled circuits used per execution.

- \( \kappa_r \) is the statistical security parameter, used to determine the length of the reconciliation strings used as PSI input (the PSI protocol scales with the length of the PSI input values). The adversary can guess an unknown reconciliation value with probability at most \( 2^{-\kappa_r} \).

\( \kappa_c \) is the computational security parameter, that controls the key sizes for OTs, commitments and garbled circuits.

In our evaluations we consider \( \kappa_c = 128 \), \( \kappa_b \in \{40, 80\} \) and \( \kappa_r \in \{20, 40, 80\} \). In the full version we prove the security of our protocol:

**Theorem 5.** Our protocol (Figure 15) securely realizes the \( \mathcal{F}_{\text{multi-set}} \) functionality, in the presence of malicious adversaries.

#### 8.1 Implementation & Architecture

In the offline phase, the work is divided between \( p \) parallel sets of 4 threads. Within each set, two threads generate OTs and two threads garble and receive circuits and related commitments. Parallelizing OT generation and circuit generation is key to our offline performance; we find that these two activities take roughly the same time.

We generate OTs using an optimized implementation of the Keller et al. [15] protocol for OT extension. Starting from 128 base OTs (computed using the protocol of [28]), we first run an OT extension to obtain 128 \( \cdot p \) OT instances. We then distribute these instances to the \( p \) different thread-sets, and each thread-set uses its 128 OT instances as base OTs to perform its own independent OT extension.

We further modified the OT extension protocol to process and finalize OT instances in blocks of 128 instances. This has two advantages: First, OT messages can be used by other threads in the offline phase as they are generated. Second, OT extension involves CPU-bound matrix transposition computations along with I/O-bound communication, and this approach interlaces these operations.

The offline phase concludes by checking the circuits in the cut-and-choose, bucketing the circuits, and exchanging garbled inputs for the random \( k \)-probe-encoded inputs.

The online phase similarly uses threading to exploit the inherently parallel nature of the protocol. Upon receiving input, a primary thread sends the other party their input correction value as the first protocol message. This value is in turn given to \( B \) sub-threads (where \( B \) is the bucket size) that transmit the appropriate wire labels. Upon receiving the labels, the \( B \) threads (in parallel) each evaluate a circuit. Each of the \( B \) threads then executes (in parallel) one of the set-membership PSI sub-protocols. After the other party has committed to their PSI inputs, the translation tables of each circuit is opened and checked in parallel. The threads then obtain the intersection and the corresponding output value.

#### 8.2 Low-level Optimizations

We instantiate the garbled circuits using the state-of-the-art half-gates construction of [35]. The implementation
utilizes the hardware accelerated AES-ni instruction set and uses fixed-key AES as the gate-level cipher, as suggested by [3]. Since circuit garbling and evaluation is the major computation bottleneck, we have taken great care to streamline and optimize the execution pipeline.

The protocol requires the bucket’s common output labels to be random. Instead, we can optimize the online phase choose these labels as the output of a hash at a random seed value. The seed can then be sent instead of sending all of the common output labels. From the seed the other party regenerates the output labels and proceed to validate the output commitments.

9 Performance Evaluation

We evaluated the prototype on Amazon AWS instances c4.8xlarge (64GB RAM and 36 virtual 2.9 GHz CPUs). We executed our prototype in two network settings: a LAN configuration with both parties in the same AWS geographic region and 0.2 ms round-trip latency; and a WAN configuration with parties in different regions and 75 ms round-trip latency.

We demonstrate the scalability of our implementation by evaluating a range of circuits:

- The AES circuit takes a 128-bit key from one party and a 128-bit block from another party and outputs a 128-bit block to both. The circuit consists of 6800 AND gates and 26,816 XOR gates.

- The SHA256 circuit takes 512 bits from both parties, XORS them together and returns the 256-bit hash digest of the XOR’d inputs. The circuit consists of 90,825 AND gates and 145,287 XOR gates.

- The AES-CBC-MAC circuit takes a 16-block (2048-bit) input from one party and a 128-bit key from the other party and returns the 128-bit result of 16-round AES-CBC-MAC. The circuit consists of 156,800 AND gates and 430,976 XOR gates.\(^7\)

In all of our tests, we use system parameters derived from Lemma 1. \(N\) denotes the number of executions, and \(B\) denotes the bucket size (number of garbled circuits per execution) and we use \(\sim B\) online threads.

9.1 PSI protocol comparison

In Section 7 we describe two PSI protocols that can be used in our 2PC protocol — a synchronous protocol that uses three rounds total, and an asynchronous protocol that uses two rounds total (at higher communication cost). We now discuss the tradeoffs between these two PSI protocols. A summary is given in Figure 5. For small parameters in the LAN setting, the 2-round asynchronous protocol is faster overall, but for larger parameters the 3-round synchronous protocol is faster. This is due to the extra data sent by the 2-round protocol. Specifically, the asynchronous protocol sends \(O(B^2 \kappa_s + \kappa_c)\) bytes while the synchronous one sends \(O(B \kappa_s + \kappa_c)\). In the remaining comparisons, we always use the PSI protocol with lowest latency, according to Figure 5.

9.2 Comparison to the LR protocol

We compare our prototype to that of [23] with 40-bit security. That is, we use \(\kappa_s = \kappa_c = 40\); both protocols have identical security and use the same bucket size. We used identical AWS instances and a similar number of threads to those reported in [23].

Figure 6 shows the results of the comparison in the LAN setting. It can be seen that our online times are 5 to 7 times faster and our offline times are 4 to 15 times faster. Indeed, for \(N = 1024\) our total (online plus offline) time is less than the online time of [23].

In the WAN setting with small circuits such as AES where the input size is minimal we see [23] achieve faster online times. Their protocol has one fewer round than ours protocol, which contributes 38ms to the difference in performance. However, for the larger SHA256 circuit our implementation outperforms that of [23] by 16 to 100ms per execution and we achieve a much more ef-

\[\begin{array}{|c|c|c|c|c|}
\hline
\kappa_s & B & \text{Time} & \text{Size} & \text{Time} & \text{Size} \\
\hline
2 & 0.31 & 2,580 & 0.35 & 138 \\
3 & 0.34 & 5,790 & 0.39 & 303 \\
4 & 0.42 & 10,280 & 0.46 & 532 \\
6 & 0.65 & 32,100 & 0.55 & 1,182 \\
80 & 0.55 & 23,100 & 0.51 & 850 \\
7 & 0.83 & 62,860 & 0.66 & 1,638 \\
9 & 1.39 & 103,860 & 0.83 & 2,682 \\
\hline
\end{array}\]

Figure 5: The running time (ms) and online communication size (bytes) of the two PSI protocols when executed with \(\kappa_s\)-bit strings and input sets of size \(B\).

\[\begin{array}{|c|c|c|c|c|c|}
\hline
\kappa_s & \kappa_c & 40 & \text{[23]} & \text{This} \\
\hline
\text{Circuit} & \text{LAN} & \text{WAN} & \text{LAN} & \text{WAN} & \text{LAN} & \text{WAN} \\
\hline
\text{AES} & 32 & 197 & 12 & 45 & 1.7 \\
& 128 & 114 & 10 & 16 & 1.5 \\
& 1024 & 74 & 7 & 5.1 & 1.3 \\
\text{SHA256} & 32 & 459 & 50 & 136 & 10.0 \\
& 128 & 275 & 40 & 78 & 8.8 \\
& 1024 & 206 & 33 & 48 & 8.4 \\
\text{AES} & 32 & 1,126 & 163 & 282 & 190 \\
& 1,024 & 760 & 160 & 34 & 189 \\
\text{SHA256} & 32 & 3,638 & 290 & 777 & 194 \\
& 128 & 3,426 & 256 & 399 & 192 \\
& 1,024 & 2,992 & 207 & 443 & 191 \\
\hline
\end{array}\]

Figure 6: Amortized running times per execution (reported in ms) for [23] and our prototype. We used bucket size \(B = 6, 5, 4\) for \(N = 32, 128, 1024\).
sufficient offline phase ranging from 4 to 22 times faster for both circuits.

As discussed in Section 2.3, our protocol has asymptotically lower online communication cost, especially for computations with larger inputs. Since both protocols are more-or-less I/O bound in these experiments, the difference in communication cost is significant. Concretely, when evaluating AES with $N = 1024$ and $B = 4$ our protocol sends 16,384 bytes of wire labels and just 564 bytes of PSI data. The online phase of [23] reports to use 170,000 bytes with the same parameters. Even using our asynchronous PSI sub-protocol, the total PSI cost is only 10,280 bytes.

### 9.3 Effect of security parameters

We show in Figure 7 how our prototype scales for different settings of security parameters in the LAN setting. In particular, the security properties of our protocol allow us to consider smaller settings of parameters than are advised with traditional cut-and-choose protocols such as [23]. As a representative example, we consider $\kappa_B = 20$ and $\kappa_s = 40$ which means that our protocol will leak a single bit only with probability $1/2^{20}$ but guarantee all other security properties with probability $1 - 1/2^{40}$.

Our protocol scales very well both in terms of security parameter and circuit size. Each doubling of $\kappa_s$ only incurs an approximate 25% to 50% increase in running time. This is contrasted by [23] reporting a 200% to 300% increase in running time for larger security parameters. Our improvement is largely due to reducing the number of cryptographic steps and no cheat-recovery circuit which consume significant online bandwidth.

We see a more significant trend in the total storage requirement of the offline phase. For example, when performing $N = 1024$ AES evaluations for security parameter $\kappa_B = 20$ the protocol utilizes a maximum of 0.76 GB of storage while $\kappa_B = 40$ requires 1.6 GB of storage. This further validates $\kappa_B = 20$ as a storage and bandwidth saving mechanism. [23] reports that 3.8 GB of offline communication for $N = 1024$ and 40-bit security.

### 9.4 Throughput & Bandwidth

In addition to considering the setting when executions are performed sequentially, we tested our prototype when performing many executions in parallel to maximize throughput. Figure 8 shows the maximum average throughput for AES evaluations that we were able to achieve, under different security parameters and bucket sizes. The time reported is the average number of milliseconds per evaluation.

In the LAN setting, 8 evaluations were performed in parallel and achieved an amortized time of 0.26 ms per evaluation for bucket size $B = 2$. A bucket size of 2 can be obtained by performing a modest number (say $N = 256$) of executions with $\kappa_B = 20$, or a very large number of executions with $\kappa_B = 40$. We further tested our prototype in the WAN setting where we obtain a slightly decreased throughput of 0.72 ms per AES evaluation with 40-bit security.

### References

[27] Nielsen, J. B., Nordholt, P. S., Orlandi, C., and Burra, S. S. A new approach to practical active-secure


[33] Smart, N. Personal communication, November 2015.


A Adaptively Secure Garbling Schemes

A garbling scheme is a tuple of algorithms \((\text{Gb}, \text{En}, \text{Ev}, \text{De})\) with the following syntax and semantics. All algorithms accept a security parameter as explicit input, which we leave implicit.

- \(\text{Gb}(f, d) \rightarrow (F, e)\): Here \(f\) is a boolean circuit with \(m\) inputs and \(n\) outputs; \(d\) is an \(n \times 2\) array of (output) wire labels; \(F\) is a garbled circuit; and \(e\) is an \(m \times 2\) array of input wire labels.

By wire labels, we simply mean strings (i.e., elements of \(\{0, 1\}^\infty\)). We deviate from [5] in requiring the output wire labels \(d\) to be chosen by the caller of \(\text{Gb}\), rather than chosen by \(\text{Gb}\) itself. In the notation of [5], we assume that the scheme is projective in both its input and output encodings, meaning that \(e\) and \(d\) consist of two possible wire labels for each wire.

- \(\text{En}(e, x) \rightarrow X\) takes an \(m \times 2\) array of wire labels \(e\) and a plaintext input \(x \in \{0, 1\}^m\) and outputs a garbled encoding \(X\) of \(x\). By assuming that the scheme is projective, we assume that \(X = (X_1, \ldots, X_m)\) where \(X_i = e[i, x_i]\).

- \(\text{Ev}(F, X) \rightarrow Y\); takes a garbled circuit \(F\) and garbled encoding \(X\) of an input, and returns a garbled encoding of the output \(Y\).

- \(\text{De}(Y) \rightarrow y\). We assume a way to decode a garbled output to a plaintext text. It is a deviation from [5] to allow this to be done without the decoding information \(d\). Rather, we may assume that the garbled outputs contain the plaintext value, say, as the last bit of each wire label.

Our correctness condition is that for the variables defined above, we have \(\text{Ev}(F, \text{En}(e, x)) = \text{En}(d, f(x))\) and \(\text{De}(\text{Ev}(F, \text{En}(e, x))) = f(x)\) for all inputs \(x\) to the circuit \(f\). In other words, evaluating the garbled circuit should result in the garbled output that encodes \(f(x)\) under the encoding \(d\).

In our construction, an adversary sees the garbled circuit \(F\) first, then it receives some of the garbled inputs (corresponding to the \(k\)-probe matrix encoded inputs). Finally in the online phase it is allowed to choose the rest of its input to the circuit and receive the rest of the garbled inputs. Hence, our security game considers an adversary that can obtain the information in this order.

We overload the syntax of the encoding algorithm \(\text{En}\). Since \(\text{En}\) is projective, we write \(\text{En}(e, i, b)\) to denote the component \(e_{i,b}\) — that is, the garbled input for the \(i\)th wire corresponding to truth value \(b\). Recall that we also garble a circuit with output wire labels \(d\) specified (rather than chosen by the \(\text{Gb}\) algorithm). Our security definition lets the adversary choose \(d\).

Definition 6. For a garbling scheme \((\text{Gb}, \text{En}, \text{Ev}, \text{De})\), an interactive oracle program \(\text{Adv}\), and algorithms \(S = (S_0, S_1, S_2)\), we define the following two games/interactions:

In \(\mathcal{G}_{\text{ideal}}\), \(H\) is a random oracle. In \(\mathcal{G}_{\text{ideal}}\), the tuple \(S = (S_0, S_1, S_2)\) all share state. All algorithms receive the security parameter as implicit input.
Then the garbling scheme is **adaptively secure** if there exists a simulator \( S \) such that for all polynomial-time adversaries \( \text{Adv} \), we have that

\[
| \Pr[\mathcal{G}^{\text{Adv}}_{\text{real}} \text{ outputs } 1] - \Pr[\mathcal{G}^{\text{Adv}, S}_{\text{ideal}} \text{ outputs } 1] |
\]

is negligible in the security parameter.

Note that in the \( \mathcal{G}_{\text{ideal}} \) game, the simulator receives no information about the input \( x \) as it produces the garbled circuit \( F \) and all but one of the garbled input components. Finally when producing the last garbled input component, the simulator learns \( f(x) \) and its **garbled output encoding** \( \text{En}(d, f(x)) \). In particular, the simulator receives no information about \( x \), so its outputs carry no information about \( x \) beyond \( f(x) \). The game also implies an authenticity property for garbled outputs of values other than \( f(x) \) — the simulator’s total output contains no information about the rest of the garbled outputs \( d \).

In **Figure 9** we describe a generic, random-oracle transformation from a standard (static-secure) garbling scheme to one with this flavor of adaptive security. The construction is quite similar to the transformations in [4], with some small changes. First, since we know in advance which the adversary will request its garbled inputs, we include the random oracle nonce \( R \) in the last garbled input value (rather than secret-sharing across all garbled inputs). Second, since we garble a circuit with particular garbled output values in mind, we provide “translation values” that will map the garbled outputs of the static scheme to the desired ones. These translation values also involve the random oracle, so they can be equivocated by the simulator.

**Theorem 7.** If \( (\text{Gb}, \text{En}, \text{Ev}, \text{De}, \tilde{\text{De}}) \) is a doubly-projective garbling scheme satisfying the (static) \( \text{prv} \) and \( \text{aut} \) properties of [5] then the scheme in **Figure 9** satisfies adaptive security notion of Definition 6 in the random oracle model.

The proof is very similar to analogous proofs in [4]. The main idea is that the simulator can choose the “masked” \( \widehat{F} \) and \( \delta \) translation values upfront. Then it is only with negligible probability that an adversary will call the random oracle on the secret nonce \( R \), so the relevant parts of the oracle are still free to be programmed by the simulator. When the adversary provides the final bit of input, the simulator gets \( f(x) \) and can obtain a simulated garbled circuit \( F \) and garbled outputs \( d \) from the static-secure scheme. Then it can program the random oracle to return the appropriate masks.\(^8\)

---

\(^8\)Technically, the proof assumes that the simulator for the static-secure scheme can set the (simulated) garbled input encoding arbitrarily. This is true for common existing schemes; e.g., [35].
Setup stage: On common input (sid, Setup, f, N, e), where f is a boolean circuit, N is the number of executions. The parties agree on parameters B, N derived from Lemma 1. Let M ∈ {0, 1}^{n×ε} be a k₂-probe resistant matrix for each party’s input of size n. Let a ∈ {0, 1} denote the role of the current party and b = a ⊕ 1. Note: the protocol is symmetric where both parties simultaneously play the roles of Pa and Pb.

- **Cut-and-Choose Commit:** Pa chooses at random the cut and choose set σa ⊂ [N] of size N − NB. Pa send (commit, (sid, cut-and-choose, a), σa) to F_{com}. For j ∈ [N]:
  - **OT Init:** Pa sends (init, (sid, ot, a, j)) to F_{rot} and receives choice bits e'_{j,h} in response.
  - **Send Circuit:** Pa chooses random output wire labels d_j, computes (F^a_j, e_j) ← Gb(f, d_j) and sends the F^a_j to Pb where f(x_a, r, x_b) = f(x_a, M_r ⊕ x_b) and r, x_b are Pa’s inputs. Let e'_{j,h}, e''_{j,h} respectively be the labels encoding x_a, x_b, r for circuit F^{a}_{j,h} and e'_{j,t,h} the index of the t^{th} wire with value h in the set e'_{j,h}.
  - **Input Commit:** Pa sends the following to F_{com}:
    - (commit, (sid, x_a-input, a, j, t, h)), e'_{j,t,h}^{a}, \{ \mu | h ∈ \{0, 1\} \}
    - (commit, (sid, x_b-input, a, j, t, h)), e''_{j,t,h}^{b} \{ \mu | h ∈ \{0, 1\} \}
  - **Output Commit:** Pb sends (commit, (sid, output, a, j)) to F_{com}.

- **Cut-and-Choose:** Pb sends (open, (sid, cut-and-choose, b)) to F_{com} and Pb receives σ_b. For j ∈ σ_b:
  - **OT Decommit:** Pb sends (open, (sid, ot, a, j)) to F_{rot} and Pb receives choice bits e'_{j,h}.
  - **Check Circuit:** Pb sends Pb the d_j and coins used to garble F^{b}_{j}. Pb verifies the correctness of F^{b}_{j}.

- **Input Commit:** Let e'_{j,h}, e''_{j,h} be the verified labels as above.
  - Pb sends (open, (sid, x_a-input, a, j, t, h)), \{ \mu | h ∈ \{0, 1\} \} to F_{com} and Pb receives labels e'_{a,h}.
  - Pb sends (open, (sid, x_b-input, a, j, t, h)), \{ \mu | h ∈ \{0, 1\} \} to F_{com} and Pb receives labels e''_{b,h}.
  - Pb sends (open, (sid, r-input, a, j, t, h)), e'_{j,t,h} \{ \mu | h ∈ \{0, 1\} \} to F_{com} and Pb receives labels e''_{j,h}.
  - If there exists a e'_{a,h} ≠ e'_{j,t,h} or e''_{b,h} ≠ e''_{j,t,h}, Pb returns ABORT.
  - **Output:** Pb sends (open, (sid, output, a, j)) to F_{com}. Pb receives d' and return ABORT if d' ≠ d_j.

- **Bucketing:** Pb randomly maps the indices of [N] − σa into sets \beta_1^{a}, ..., \beta_\epsilon^{a} s.t. \beta_\epsilon^{a} = B. For i ∈ [N]:
  - **Bucket Labels:** Pa generates random output labels O^{a}_{i} for bucket \beta^{a}_{i}. For j ∈ \beta^{a}_{i}, Pb sends the output translation T_{i,j} := \{ O^{a}_{i,j} ⊕ d_{j} \},_h and H(O^{a}_{i,j}), to Pb, where d_{j} are the output labels of F^{b}_{j}.
  - **Offline Inputs:**
    - Pb sends (aggregate, (sid, ot-ag, a, i), \{ (sid, ot, a, j) | j ∈ \beta^{b}_{i} \}) to F_{rot} and Pb receives the OT aggregation strings \beta^{b}_{i} for j ∈ \beta^{b}_{i}.
    - Pb sends (deliver, (sid, ot-ag, a, i), \{ e'_{j,h} | j ∈ \beta^{b}_{i} \}) to F_{rot} where w_j are the decommitment strings to \{ (sid, r-input, b, j, t, h) \},_h.
    - For j ∈ \beta^{b}_{i}, Pb receives X^{e}_{j, a} and Y^{w}_{j} from F_{rot}. Pb sends (open, (sid, x_a-input, b, j, t, e'_{j,h}^{a})) \{ W^{j}_{a} \},_h to F_{com} and receives X^{a}_{j, a}. Pb returns ABORT if X^{a}_{j, a} ≠ X^{w}_{j}.

Execution stage: On common bucket index i and Pb’s input x_a.

- **Receiver’s Inputs:** Let f be the first index in \beta^{b}_{i}. Pb sends \tilde{x}_a := x_a ⊕ M_e^{b} to Pb where e^{b}_{j} are the choice bits of (sid, ot, a, j). For all j ∈ \beta^{b}_{i}:
  - Pb sends X^{e}_{j, a} := \{ (e^{j}_{t,h}, (x_a ⊕ M_e^{b}), (x_b ⊕ M_e^{b}), \mu | h ∈ \{0, 1\} \},_h and W^{w}_{j} := \{ (w_{j,a} ⊕ M_e^{b}), \mu | h ∈ \{0, 1\} \},_h to Pb where e^{j}_{t,h} encodes \tilde{x}_a for F^{b}_{j} and w_{j,a} are the decommitment string to \{ (sid, x_a-input, b, j, t, h) \},_h.
  - Pb receives X^{a}_{j, a}, W^{a}_{j} and sends (open, (sid, x_a-input, b, j, t, (x_a ⊕ M_e^{b})), W^{j}_{a})\{ \} to F_{com} and receives X^{e}_{j, a}. Pb returns ABORT if X^{a}_{j, a} ≠ X^{w}_{j}.

- **Sender’s Inputs:** For j ∈ \beta^{b}_{i}, Pb sends (open, (sid, x_a-input, b, j, t, (x_a ⊕ M_e^{b}))), \{ \} to F_{com}. Pb receives the labels X^{a}_{j, a} and returns ABORT if x_a ⊕ M_e^{b} ≠ (x_a ⊕ M_e^{b}).

- **Evaluate:** For j ∈ \beta^{b}_{i}, let Y_{j} := \tilde{E}(F^{b}_{j}, (X^{e}_{j, a}, X^{a}_{j, a})) with semantic value y_{j}.

- **PSI Commit:** For \forall j, t, if (H(Y_{j,t}) ≠ H(\tilde{O}_{j,t,y_{j}})), then Y_{j,t} ← \{0, 1\}^\infty. Let I := \{ Y_{j,t} ⊕ T^{b}_{j,t,y_{j}} ⊕ O^{d}_{j,t,y_{j}} | j ∈ \beta^{b}_{i} \}. Pb pads I to size B with random values and sends (input, (sid, psi, i), I) to F_{psi} and receives (input, Pb).

- **Output Decommit:** For j ∈ \beta^{b}_{i}, Pb sends (open, (sid, output, b, j)) to F_{com} and Pb receives d'_{j}.
  - If there exists j, j' s.t. d'_{j,t,h} \neq d'_{j',t,h} \neq T^{b}_{j,h}, Pb returns ABORT.
  - **PSI Decommit:** Pb sends (open, (sid, psi, i)) to F_{psi} and Pb receives the intersection R. If |R| ≠ 1, Pb returns CHEATING!

Figure 15: Malicious secure online/offline dual-execution 2PC protocol \Pi_{multi-sfe}. 
Parameters: A sender $P_1$ and receiver $P_2$.

Setup: On common input $S$ from both parties, for every $s \in S$ choose random $m_0, m_1 \leftarrow \{0,1\}^\kappa$ and random $c \leftarrow \{0,1\}$. Internally store a tuple $(s, m_0, m_1, c)$.

$P_1$ output: On input $(\text{GET}, s)$ from $P_1$, if there is a tuple $(s, m_0, m_1, c)$ for some $m_0, m_1, c$ then give $(\text{OUTPUT}, s, m_0, m_1)$ to $P_1$.

$P_2$ output: On input $(\text{GET}, s)$ from $P_2$, if there is a tuple $(s, m_0, m_1, c)$ for some $m_0, m_1, c$ then give $(\text{OUTPUT}, s, c, m_1)$ to $P_2$.

Figure 10: Random OT functionality $\mathcal{F}_{ot}$.

Parameters: A sender $P_1$ and receiver $P_2$.

Commit: On input $(\text{COMMIT}, \text{sid}, v)$ from $P_1$: If a tuple of the form $(\text{sid}, \cdot, \cdot)$ is stored, then abort. If $P_1$ is corrupt, then obtain value $r$ from the adversary; otherwise choose $r \leftarrow \{0,1\}^\kappa$ and give $r$ to $P_1$. Internally store a tuple $(\text{sid}, r, v)$ and give $(\text{COMMITTED}, \text{sid})$ to $P_2$.

Reveal: On input $(\text{OPEN}, \text{sid}, r', v)$ from $P_2$: if a tuple $(\text{sid}, r', v)$ is stored for some $v$, then give $(\text{OPENED}, \text{sid}, v)$ to $P_2$. Otherwise, give $(\text{ERROR}, \text{sid})$ to $P_2$.

Figure 11: Non-interactive commitment functionality $\mathcal{F}_{com}$.

Parameters: Two parties: a sender $P_1$ and receiver $P_2$; $\ell = \text{bit-length of items in the set}; n = \text{size of parties’ sets}; F = a \text{ PRF}$.

Offline phase: Parties perform random OTs, resulting in $P_1$ holding strings $m_0^{d_i[0,1]} \leftarrow \{0,1\}^\kappa$; and $P_2$ holding $c_i$ and $m_i^{\ell[i]}$. Here, $c_i \in \{0,1\}^\ell$ and $i \in [n]$, $t \in [\ell]$.

Input committing phase:

- On input $(\text{INPUT}, \{A_{2,1}, \ldots, A_{2,n}\})$ to $P_2$, $P_2$ randomly permutes its input and then sends $d_i := A_{2,i} \oplus c_i$ for each $i \in [n]$.
- On input $(\text{INPUT}, \{A_{1,1}, \ldots, A_{1,n}\})$ to $P_1$, $P_1$ randomly permutes its input and then computes $S_{i,j} = \bigoplus_j F(m_{i,j}^{d_i[0,1] \oplus A_{1,j}[0,1]}; j) \text{ for } i,j \in [n]$.
- $P_1$ sends $(\text{COMMIT}, \text{sid}, (S_1, 1, \ldots, S_n, n))$ to $\mathcal{F}_{\text{com}}$.

Output phase: On input $(\text{OUTPUT})$, $P_1$ sends $(\text{OPEN}, \text{sid})$ to $\mathcal{F}_{\text{com}}$ and $P_2$ receives $(\text{OPENED}, \text{sid}, (S_1, 1, \ldots, S_n, n))$. $P_2$ then outputs $(A_{2,j} \mid \exists j: \bigoplus_i F(m_{i,j}^{\ell[0,1]}), j = S_{i,j})$.

Figure 13: Weakly-malicious-secure, synchronous (3-round), two-phase PSI protocol $\Pi_\text{sync-psi}$.

Parameters: Two parties: a sender $P_1$ and receiver $P_2$; $\ell = \text{bit-length of items in the set}; n = \text{size of parties’ sets}; F = a \text{ PRF}$.

Offline phase: Parties perform random OTs, resulting in $P_1$ holding strings $m_0^{d_i[0,1]} \leftarrow \{0,1\}^\kappa$; and $P_2$ holding $c_i$ and $m_i^{\ell[i]}$. Here, $c_i \in \{0,1\}^\ell$ and $i \in [n]$, $t \in [\ell]$.

For $i \in [n]$, $P_1$ chooses $\pi_i \leftarrow \{0,1\}^\ell$. Then for $i,j \in [n]$, party $P_1$ does the following:
- For $t \in \{0,1\}^\ell$, choose $z_i^{t,j} \leftarrow \{0,1\}^\ell$ subject to $\bigoplus_{t \in [\ell]} z_i^{t,j} = 0$
- For $t \in [\ell], b \in \{0,1\}$, $P_1$ sends $(\text{COMMIT}, (\text{sid}, i, j, t, b), F(m_i^{d_i[0,1] \oplus b, j}, j \oplus z_i^{t,j}))$ to $\mathcal{F}_{\text{com}}$.

Input committing phase: On input $(\text{INPUT}, \{A_{1,1}, \ldots, A_{1,n}\})$ for $P_1$ and $(\text{INPUT}, \{A_{2,1}, \ldots, A_{2,n}\})$ for $P_2$, the parties randomly permute their inputs and asynchrnously do:
- $P_1$ sends $d_{1,j} := A_{1,j} \oplus \pi_j$ for each $j \in [n]$
- $P_2$ sends $d_{2,j} := A_{2,j} \oplus c_i$ for each $i \in [n]$

Output phase: On input $(\text{OUTPUT})$: for $i,j \in [n]$, $t \in [\ell]$, party $P_1$ sends $(\text{OPEN}, (\text{sid}, i, j, t, d_{1,j}[0,1]) \oplus d_{2,i}[t]))$ to $\mathcal{F}_{\text{com}}$ and $P_2$ expects to receive $(\text{OPENED}, (\text{sid}, i, j, t, d_{1,j}[0,1] \oplus d_{2,i}[t], b_{1,j}^{t})$.

$P_2$ outputs $(A_{2,i} \mid \exists j: \bigoplus_i F(m_{i,j}^{\ell[0,1]}), j = B_{1,j})$.

Figure 14: Weakly-malicious-secure, asynchronous (2-round), two-phase PSI protocol $\Pi_\text{async-psi}$.