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

Peter Rindal† Mike Rosulek†

June 17, 2016

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 *5.1ms offline + 1.3ms 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 74ms offline + 7ms 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 *2.4ms offline + 1.0ms 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 [Yao82] and shown to be feasible in principle using a construction now known as *garbled circuits*. Later, the Fairplay project [MNPS04] 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.

---

*This is the full version of a paper appearing in USENIX 2016.

†Oregon State University, {rindalp,rosulekm}@eecs.oregonstate.edu. Supported by NSF award 1149647. The first author is also supported by an ARCS foundation fellowship.

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 [LP07]. This overhead was later reduced to $3s$ circuits [LP11, sS11, sS13] and then s circuits [Lin13].

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 pre-processing (which does not depend on the inputs). In this *online/offline setting*, far fewer garbled circuits are needed per execution. The idea, due to [HKK$^+$14, LR14], 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 [MF06], 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.* [KMRR15] 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 [KMRR15] 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 [KMRR15] 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 straight-forward, some non-trivial technical changes are necessary to adapt [KMRR15] 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 [KMRR15] is incompatible with offline pre-processing, whereas the method from [LR15] 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 [KMRR15], the parties evaluate garbled circuits and then use *active-secure* private set intersection (PSI) to reconcile their outputs. We improve the analysis of [KMRR15] 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 [PSZ14]) 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 [ZRE15]; the OT-extension protocol of [KOS15] instantiated with the base OTs of [CO15]. 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 co-located 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 [AMPR14, FJN14, KSS12, LR15, PSSW09, sS11, sS13]. Except for [LR15], none of these protocols are in the online/offline settings so their performance is naturally much lower (100-1000$\times$ slower than online/offline protocols). Among them, the fastest reported secure evaluation of AES is that of [FJN14], 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 [HKE13, MR13]. The protocol of [HKE13] leaks more than one bit when the adversary successfully cheats during cut-and-choose.

Our protocol is most closely related to that of [LR15], which also achieves fast, active-secure 2PC in the online/offline setting. [LR15] is an implementation of the protocol of [LR14], 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 achieveing malicious security. 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

	Input Labels	Reconciliation
LR [LR15]	$\lvert x\rvert(B+B')\kappa_c$	$\lvert x\rvert B'\kappa_c$
Us (Async PSI)	$\lvert x\rvert B\kappa_c$	$B^2\kappa_s\kappa_c$
Us (Sync PSI)		$B\kappa_s + B^2\kappa_c$

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 [LR15]; $|x|$ is length of sender's inputs; κ_s is the statistical security parameter; κ_c is the computational security parameter.

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 κ_s denote a statistical security parameter, meaning that the protocol allows the adversary to completely break security with probability $1/2^{\kappa_s}$. 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 κ_s.

Our protocol has an additional parameter κ_b, where the protocol leaks (only) a single bit to the adversary with probability $1/2^{\kappa_b}$. In our protocol (as in [KMRR15]), the number of garbled circuits is proportional to κ_b. When instantiated with $\kappa_b = \kappa_s = 40$, our protocol gives an equivalent guarantee to the LR protocol with $\kappa_s = 40$. From this baseline, our protocol allows either κ_s to be increased (strictly improving the security guarantee without involving more garbled circuits) or κ_b to be decreased (trading performance for a small chance of a single bit leaking).[1]

(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 [Lin13]: 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 proprotional 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 $\kappa_s = 40$, 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[2] and delivers output only to one party. If both parties

[1] 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 $1/2^{20}$ or $1/2^{30}$ chance of leaking a single bit about this key might be permissible.

[2] 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.

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 [DLT14, DPSZ12, DZ15, DZ13, NNOB12]. 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 [DPSZ12] can securely evaluate AES with online latency 20ms [Sma15]. Of special note is the implementation of the [DZ13] protocol reported in [DZ15], 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 [DPSZ12] protocol require 156 seconds of offline time for securely computing AES [Sma15]; 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 evalauted, whereas ours does.

3 Preliminaries

Secure computation. We use the standard notion of universally composable (UC) security [Can01] 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 $\mathcal{F}_{\mathsf{multi\text{-}sfe}}$ 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 $\mathcal{L}$ 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 ϵ. Further, the leakage is "risky" in the sense that the honest party detects cheating when the leaked bit is zero.

Building blocks. In Figures 3 & 4 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 $r \leftarrow \{0,1\}^{\kappa_c}$ and sending $c = H(r\|v)$.

Setup stage: On common input $(\textsc{setup}, f, N, \epsilon)$ from both parties, where f is a boolean circuit:

- If neither party is corrupt, set $L = 0^N$. Otherwise, wait for input $(\textsc{cheat}, \mathcal{L})$ where $\mathcal{L}$ is a distribution over $\{0,1\}^N \cup \{\perp\}$ with the property that for every i, $\Pr_{L \leftarrow \mathcal{L}}[L_i = 1] \leq \epsilon$. Sample $L \leftarrow \mathcal{L}$ using random coins χ and give $(\textsc{cheatresult}, \chi)$ to the adversary. If $L = \perp$ then give output $(\textsc{cheating!})$ to the honest party and stop responding.
- Send output $(\textsc{ready})$ to both parties. Initialize counter $ctr = 1$. Proceed to the execution stage.

Execution stage: Upon receiving inputs $(\textsc{input}, x_1)$ from P_1 and $(\textsc{input}, x_2)$ from P_2:

- Compute $z = f(x_1, x_2)$. If both parties are honest, give $(\textsc{output}, ctr, z)$ to both parties.
- If any party is corrupt, give $(\textsc{output}, ctr, z)$ to the adversary.
- If $L_{ctr} = 1$, wait for a command $(\textsc{leak}, P)$ from the adversary, where P is a boolean predicate. Compute $p = P(x_1, x_2)$ and give $(\textsc{leakresult}, p)$ to the adversary.
- If any party is corrupt, then on input $(\textsc{deliver})$ from the adversary, if $p = 0$ above, then give output $(\textsc{cheating!})$ to the honest party, else give output $(\textsc{output}, ctr, z)$ to the honest party.
- If $ctr = N$ then stop responding; otherwise set $ctr = ctr + 1$ and repeat the execution stage.

Figure 2: The (ϵ-leaking) secure function evaluation functionality $\mathcal{F}_{\text{multi-sfe}}$.

We use and adapt the Garbled Circuit notation and terminology of [BHR12b]; for a formal treatment, consult that paper. In Appendix A we define the syntax and security requirements, highlighting the differences we adopt compared to [BHR12b].

4 The Dual Execution Paradigm

We now give a high-level outline of the (non-online/offline) 2PC protocol paradigm of [KMRR15], 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 $\mathcal{F}_{\text{psi}}^{n,\ell}$ functionality in Figure 5.

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_b 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.

Parameters: A sender P_1 and receiver P_2.

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

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

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

Figure 3: Random OT functionality $\mathcal{F}_{\text{ot}}$.

Parameters: A sender P_1 and receiver P_2.

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

Reveal: On input $(\textsc{open}, \mathsf{sid}, r')$ from P_2: if a tuple (sid, r', v) is stored for some v, then give $(\textsc{opened}, \mathsf{sid}, v)$ to P_2. Otherwise, give $(\textsc{error}, \mathsf{sid})$ to P_2.

Figure 4: Non-interactive commitment functionality $\mathcal{F}_{\mathsf{com}}$.

Parameters: Two parties: a sender P_1 and receiver P_2; ℓ = length of items; n = size of parties' sets.

First phase (input commitment): On input $(\textsc{input}, A_i)$ from party P_i ($i \in \{1,2\}$), with $A_i \subseteq \{0,1\}^\ell$ and $|A_i| = n$: If this is the first such command from P_i then internally record A_i and send message $(\textsc{input}, P_i)$ to both parties.

Second phase (output): On input $(\textsc{output})$ from P_i, deliver $(\textsc{output}, A_1 \cap A_2)$ to the other party.

Figure 5: Two-phase private set intersection (PSI) functionality $\mathcal{F}_{\mathsf{psi}}^{n,\ell}$.

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_y^b (b for a garbled output under Bob's encoding), let Y_y^a denote the encoding of y under Alice's garbled output encoding (which Alice can compute). Interpreting Y_y^a and Y_y^b as *sets* of individual wire labels, let R_y be the XOR of all items in $Y_y^a \cup Y_y^b$, which we write as $R_y = \bigoplus[Y_y^a \cup Y_y^b]$ 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 $2^{-\kappa_b}$ 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 $Y^b_{y^*}$ 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 $R_y = \bigoplus[Y^a_y \cup Y^b_y]$.

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 $2^{-\kappa_b}$ — 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 [KMRR15] 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 [LR15], we focus on choosing the number of circuits so that the probability of such an event in *any particular* bucket is $2^{-\kappa_b}$. We note that the analysis of parameters in [HKK+14, LR14] 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 ([LR15]). *If the parties plan to perform N executions, using a bucket of B circuits for each execution and a total of $\widehat{N} \geq NB$ garbled circuits generated for the overall cut-and-choose, then the probability that a specific bucket contains no good circuit is at most:*

$$\max_{t\in\{B,\dots,NB\}} \left\{ \frac{\binom{\widehat{N}-t}{NB-t}}{\binom{\widehat{N}}{NB}} \cdot \frac{\binom{t}{B}}{\binom{NB}{B}} \right\}.$$

Suppose the parties will perform N executions, using buckets of size B in the online phase, and wish for $2^{-\kappa_b}$ probability of leakage. We can use the formula to determine the smallest compatible $\widehat{N}$. In Figure 17 we show all reasonable parameter settings for $\kappa_b \in \{20, 40, 80\}$ and $N \in \{8, 16, 32, \ldots, 32768\}$.

By adapting [KMRR15] to the online/offline setting, we obtain the generic protocol outlined in Figure 6. Even with pre-processing, an online OT requires 2 rounds, one of which can be combined with the direct sending of garbled inputs. The protocol therefore requires 3 rounds plus the number of rounds needed for the PSI subprotocol (at least 2).

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 κ_b, the parties determine appropriate parameters $\widehat{N}, B$ according to the discussion in Section 4.2. Each party generates and sends $\widehat{N}$ garbled circuits, and chooses a random subset of $\widehat{N} - NB$ 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 6: High-level outline of the online/offline, dual-execution protocol paradigm.

4.3 Technicalities

We highlight which parts of the [KMRR15] protocol break down in the online/offline setting and require technical modification:

Same garbled output encoding. In [KMRR15] 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^*, W_1^* encoding false and true, respectively. Then if W_0^j, W_1^j are the output wire labels already chosen for the jth circuit in this bucket, the garbler is supposed to provide *translation values* $W_v^j \oplus W_v^*$

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 6) 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 [HKK+14, LR14]. See Appendix A for details.

Input consistency. To achieve security against active adversaries, GC-based protocols must ensure that parties provide the same inputs to all circuits that are evaluated. This is known as the problem of *input consistency.* The protocol of [KMRR15] uses the input consistency mechanism of shelat & Shen [sS13] which is unfortunately not compatible with the online/offline setting. More details follow in the next section.

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 [Bea95] 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^*, m_1^*), Alice can send $d = c \oplus c^*$ and Bob can reply with $m_0^* \oplus m_d$ and $m_1^* \oplus m_{1\oplus d}$.

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 ith 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 $(\textsc{commit}, (\mathsf{sid}, i), A_{i \oplus c})$ to $\mathcal{F}_{\mathsf{com}}$ for $i \in \{0, 1\}$.

Online: Alice send $d = c \oplus x$ to Bob and $(\textsc{open}, (\mathsf{sid}, d))$ to $\mathcal{F}_{\mathsf{com}}$. Bob receives $(\textsc{open}, (\mathsf{sid}, d), A_x)$ from $\mathcal{F}_{\mathsf{com}}$ and sends $(B_0 \oplus m_d, B_1 \oplus m_{1 \oplus d})$ to Alice who computes $B_x = m_c \oplus (B_{c \oplus d} \oplus m_c)$.

Figure 7: 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, m^*_1, Alice can send $d = c \oplus c^*$, and Bob can reply with $m^*_0 \oplus (m_d \oplus m'_{d \oplus \delta})$ and $m^*_1 \oplus (m_{1 \oplus d} \oplus m'_{1 \oplus d \oplus \delta})$.

The idea extends to aggregate any number B of different random OTs into a single one, with Alice sending $B - 1$ different δ 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 δ 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 jth circuit. Alice is then supposed to commit to the garbled inputs of her jth 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.[3] Hence with probability at least $1 - 2^{-\kappa_b}$, 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 jth OTs. To send her garbled input for the jth 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

[3] In fact, since the OT messages are long random strings, Alice can prove that she had particular choice bits in *many* OTs by simply reporting the $\textsc{xor}$ of all of the corresponding OT messages.

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 jth circuit is "good" and Alice provided honest aggregation strings, then Alice will have decommitted to inputs for the jth 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 $1/2^{\kappa_b}$), 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 ([LP07, sS13])**.** *Matrix $M \in \{0,1\}^{\ell \times n}$ is called k-**probe resistant** if for any $L \subseteq \{1, 2, ..., n\}$, the Hamming distance of $\bigoplus_{i \in L} M_i$ is at least k, where M_i denotes the ith 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 [KS08] (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 κ_s-probe resistant, where κ_s is the statistical security parameter. We refer the reader to [LR15] for the construction details of k-probe resistant matrices and their parameters.

6.1 Offlining 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 [LR14] suggest a technique that moves these computations associated with k-probe-resistant encodings to the offline phase. The parties will compute the related function $f'(\hat{x}_1, c, x_2) = f(\hat{x}_1 \oplus Mc, x_2)$. 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.

[4] Technically, the sender will commit to all garbled inputs, and then the OTs will be used to transfer the decommitment values. That way, the receiver can abort immediately if an incorrect decommitment value is received.

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 $\mathbb{Z}_2$-linearity of the encoding/decoding operation.

Consider a bucket of circuits $\{1, \dots, B\}$. For the jth 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 jth 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-zeroes derandomization message. This corresponds to her accepting the random c_1 as her choice bits. (Of course, an all-zeroes 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 jth 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 jth 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 jth circuit is "good" (*i.e.*, garbled correctly and input commitments arranged by Mc_j). As before, define her *effective* OT input to the jth OTs as $c^* = c_j \oplus \delta_j$ (which should be c_1 if Alice did not lie about δ_j). Even if Alice lied about the δ values she surely learns no more than she would have learned by being truthful about the δ 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 [KMRR15] 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 Π be a two-phase protocol for set intersection ($\mathcal{F}_{psi}^{n,\ell}$, Figure 5). We say that Π is* ***weakly malicious-secure*** *if it achieves UC-security with respect to environments that behave as follows:*

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

2. *The environment (privately) samples $A \leftarrow \mathcal{D}$ 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 ℓ 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. [PSZ14]. Their protocol is proven secure only against passive adversaries. We later discuss how to achieve weak malicous 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 ith one, the receiver uses choice bit $r[i]$ and the sender chooses random string inputs (m_0^i, m_1^i). The sender finally sends $S = \bigoplus_i m^i_{s[i]}$, and the receiver checks whether $S = \bigoplus_i m^i_{r[i]}$, 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 $\{s^1, \dots, s^t\}$ of strings, and receiver learns whether $r \in \{s^1, \dots, s^t\}$. We simply have the sender randomly permute the s^j values, compute for each one $S^j = \bigoplus_i F(m^i_{s^j[i]}, j)$ and send $\{S^1, \dots, S^t\}$, where F is a PRF.[5] The receiver can check whether $\bigoplus_i F(m^i_{r[i]}, j)$ matches S^j for any j. Finally, we can achieve a PSI

[5] Simply XORing the m_b^i values would reveal some linear dependencies; applying a PRF renders all of the S^j values independently random except the ones for which $r = s^j$.

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 $O(t^2)$, and [PSZ14] describe 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 [PSZ14] 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 $S, \{m_b^i\}$ and determining a value s such that $S = \bigoplus_i m^i_{s[i]}$. 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 t' of the t subprotocols, then the simulator can send s^* to the ideal PSI functionality with probability 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 2 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 2 PSI rounds are a bottleneck within the overall dual-execution protocol. We now describe a variant of the PSI protocol in which the 2 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^i, m_1^i. 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 $S = \bigoplus_i [m^i_{d[i]\oplus s[i]} \oplus z_i]$ 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):

$$\begin{array}{llllll} (\star) & [m^1_{s[1]} \quad \oplus z_1] & [m^2_{s[2]} \quad \oplus z_2] & \cdots & [m^n_{s[n]} \quad \oplus z_n] \\ & [m^1_{s[1]\oplus 1} \oplus z_1] & [m^2_{s[2]\oplus 1} \oplus z_2] & \cdots & [m^n_{s[n]\oplus 1} \oplus z_n] \end{array}$$

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 $\{m^i_{d[i]\oplus s[i]} \oplus z_i\}$; 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 $\bigoplus_i m^i_{d[i]\oplus s^*[i]}$.[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 ($\star$) 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 π. 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 $F(m^i_b, j) \oplus z^j_i$ 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 8 & 9.

Theorem 4. *The protocols $\Pi_{\textit{sync-psi}}$ and $\Pi_{\textit{async-psi}}$ described in Figures 8 & 9 are weakly-malicious secure (in the sense of Definition 3) when $\ell \geq \kappa_s$, the statistical security parameter.*

Proof. We prove security of the protocol in Figure 8; the proof of the other protocol is extremely similar. We first consider the case of a corrupt receiver P_2. In fact, the protocol is *fully secure* in this case.

Hybrid 0 (Real interaction) The simulator runs the protocol on behalf of the honest sender P_1 and simulates the ideal functionalities honestly. In the offline phase the simulator extracts the adversary's choice bits c_i in the OTs. In the input committing phase, the simulator obtains d_i from the adversary and computes $A_{2,i} = d_i \oplus c_i$.

Replacing d_i with $A_{2,i} \oplus c_i$, we can rewrite honest P_1's behavior as:

$$S_{i,j} = \bigoplus_t F(m^{i,t}_{c_i[t]\oplus A_{1,j}[t]\oplus A_{2,i}[t]}, j)$$

Hybrid 1 In the previous hybrid, values $m^{i,t}_{c_i[t]\oplus 1}$ are distributed independently of the adversary's view. By the PRF-security of F, we can replace any call to F that uses such a key with a random value. The resulting hybrid is indistinguishable from the previous one.

In doing so, we see that whenever $A_{1,j} \neq A_{2,i}$, the expression for $S_{i,j}$ involves such a term. Hence $S_{i,j}$ is uniformly distributed whenever $A_{1,j} \neq A_{2,i}$.

[6]Note: although we intend for the two parties' messages to be sent simultaneously, we must be able to simulate in the case that a corrupt sender waits for incoming message d before sending its commitments.

Hybrid 2 In the previous hybrid, the $S_{i,j}$ values can be simulated knowing only whether $A_{1,j} = A_{2,i}$. This information can be obtained by the simulator sending $A_2 = \{A_{2,1}, \ldots, A_{2,n}\}$ to the ideal PSI functionality and receiving output $A^* \subseteq A_2$. This hybrid defines our final simulation.

Next, we consider the case of a corrupt sender P_1. In this case, the protocol is only *weakly* secure:

Hybrid 0 (Real interaction) The simulator obtains a^* and $\mathcal{D}$ from the adversary, then samples the honest party's input $\mathcal{D}$ and runs the protocol on behalf of the honest receiver P_2. The simulator also simulates the ideal functionalities honestly.

Hybrid 1 Note that P_1's view is independent of P_2's input. The only protocol messages from P_2 are the d_i values in the first step. In this hybrid, the simulator simulates these to be random messages, rather than using honest P_2's actual input.

Hybrid 2 From the adversary's view, P_2 determines its output via

$$\{A_{2,i} \mid \exists j : \bigoplus_t F(m^{i,t}_{c_i[t]}, j) = S_{i,j}\}$$

$$\{A_{2,i} \mid \exists j : \bigoplus_t F(m^{i,t}_{A_{2,i}[t] \oplus d_i[t]}, j) = S_{i,j}\}$$

We modify this hybrid so that P_2's output is computed in the following way: The simulator chooses a random $i^* \in [n]$. Then P_2 outputs $\{a^*\}$ if $\exists j : \bigoplus_t F(m^{i^*,t}_{a^*[t] \oplus d_{i^*}[t]}, j) = S_{i^*,j}$; otherwise P_2 outputs $\emptyset$.

Recall that P_2's PSI input consists of a^* along with values chosen from the "unpredictable" distribution $\mathcal{D}$. Hence, the hybrids differ only in the case where there is some $A_{2,i}$ chosen from $\mathcal{D}$ (so $i \neq i^*$) satisfying:

$$\exists j : \bigoplus_t F(m^{i,t}_{A_{2,i}[t] \oplus d_i[t]}, j) = S_{i,j} \tag{1}$$

For different choices of $A_{2,i}$, the left hand side of this expression is indistinguishable from a uniformly distributed value. This holds because the $m^{i,t}_b$ values are chosen uniformly (by the simulator), and even though they are public, they are chosen independently of the argument j to the PRF. Hence PRF security applies. It is therefore only negligibly likely that there exist $a \neq a'$ satisfying:

$$\bigoplus_t F(m^{i,t}_{a[t] \oplus d_i[t]}, j) = \bigoplus_t F(m^{i,t}_{a'[t] \oplus d_i[t]}, j)$$

In other words, any $S_{i,j}$ value produced by the adversary can be consistent with at most a *single* candidate $A_{2,i}$ value of P_2, except with negligible probability. Since the $A_{2,i}$ values are unpredictable (in the sense of Definition 3) and chosen independently of the adversary's view, the probability of event (1) is therefore negligible. Hence the hybrids are indistinguishable.

Hybrid 3 In the previous hybrid, the simulated P_2 chooses a random i^* and simply checks whether $\exists j : \bigoplus_t F(m^{i^*,t}_{a^*[t] \oplus d_{i^*}[t]}, j) = S_{i^*,j}$. However, since both a^* and the $m^{i,t}_b$ messages are known to the simulator, the simulator can perform this check. If the check succeeds, the simulator sends $\{a^*\}$ to the ideal functionality, otherwise it sends $\emptyset$ to the ideal functionality. Since honest (ideal) P_2 always includes a^* in its PSI input, the ideal output always matches the simulator's input. This hybrid defines our final simulation.

□

Parameters: Two parties: a sender P_1 and receiver P_2; ℓ = bit-length of items in the set; n = size of parties' sets; F = a PRF.

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

Input committing phase:

- On input $(\textsc{input}, \{A_{2,1}, \ldots, A_{2,n}\})$ 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 $(\textsc{input}, \{A_{1,1}, \ldots, A_{1,n}\})$ for P_1, P_1 randomly permutes its input and then computes $S_{i,j} = \bigoplus_t F(m^{i,t}_{d_i[t] \oplus A_{1,j}[t]}, j)$ for $i, j \in [n]$.
- P_1 sends $(\textsc{commit}, \mathsf{sid}, (S_{1,1}, \ldots, S_{n,n}))$ to $\mathcal{F}_{\mathsf{com}}$.

Output phase: On input $(\textsc{output})$, P_1 sends $(\textsc{open}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{com}}$ and P_2 receives $(\textsc{opened}, \mathsf{sid}, (S_{1,1}, \ldots, S_{n,n}))$. P_2 then outputs $\{A_{2,i} \mid \exists j : \bigoplus_t F(m^{i,t}_{c_i[t]}, j) = S_{i,j}\}$.

Figure 8: Weakly-malicious-secure, synchronous (3-round), two-phase PSI protocol $\Pi_{\mathsf{sync\text{-}psi}}$.

Parameters: Two parties: a sender P_1 and receiver P_2; ℓ = bit-length of items in the set; n = size of parties' sets; F = a PRF.

Offline phase: Parties perform random OTs, resulting in P_1 holding strings $m^{i,t}_{\{0,1\}} \leftarrow \{0,1\}^{\kappa_c}$; and P_2 holding c_i and $m^{i,t}_{c_i[t]}$. 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 $z^{i,j}_t \leftarrow \{0,1\}^\ell$ subject to $\bigoplus_t z^{i,j}_t = 0$
- for $t \in [\ell], b \in \{0,1\}$; P_1 sends $(\textsc{commit}, (\mathsf{sid}, i, j, t, b), F(m^{i,t}_{\pi_j[t] \oplus b}, j) \oplus z^{i,j}_t)$ to $\mathcal{F}_{\mathsf{com}}$.

Input committing phase: On input $(\textsc{input}, \{A_{1,1}, \ldots, A_{1,n}\})$ for P_1 and $(\textsc{input}, \{A_{2,1}, \ldots, A_{2,n}\})$ for P_2, the parties randomly permute their inputs and asynchronously do:

- P_1 sends $d_{1,j} := A_{1,j} \oplus \pi_j$ for each $j \in [n]$
- P_2 sends $d_{2,i} := A_{2,i} \oplus c_i$ for each $i \in [n]$

Output phase: On input $(\textsc{output})$: for $i, j \in [n], t \in [\ell]$, party P_1 sends $(\textsc{open}, (\mathsf{sid}, i, j, t, d_{1,j}[t] \oplus d_{2,i}[t]))$ to $\mathcal{F}_{\mathsf{com}}$ and P_2 expects to receive $(\textsc{opened}, (\mathsf{sid}, i, j, t, d_{1,j}[t] \oplus d_{2,i}[t]), \rho^{i,j}_t)$.

P_2 outputs $\{A_{2,i} \mid \exists j : \bigoplus_t F(m^{i,t}_{c_i[t]}, j) = \bigoplus_t \rho^{i,j}_t\}$

Figure 9: Weakly-malicious-secure, asynchronous (2-round), two-phase PSI protocol $\Pi_{\mathsf{async\text{-}psi}}$.

8 Protocol Details & Implementation

The full details of our protocol are given in Figure 11 and the c++ implementation may be found at `https://github.com/osu-crypto/batchDualEx`. The protocol uses three security parameters:

Setup stage: On common input $(\mathsf{sid}, \textsc{setup}, f, N, \epsilon)$, where f is a boolean circuit, N is the number of executions. The parties agree on parameters $B, \widehat{N}$ as specified by Figure 17. Let $M \in \{0,1\}^{\mu \times n}$ be a k_s-probe resistant matrix for each party's input of size n. Let $a \in \{0,1\}$ denote the role of the current party and $b = a \oplus 1$. *Note:* the protocol is symmetric where both parties simultaneously play the roles of P_a and P_b.

- **Cut-and-Choose Commit:** P_a chooses at random the cut and choose set $\sigma_a \subset [\widehat{N}]$ of size $\widehat{N} - NB$. P_a send $(\textsc{commit}, (\mathsf{sid}, \textsc{cut-and-choose}, a), \sigma_a)$ to $\mathcal{F}_{\mathsf{com}}$. For $j \in [\widehat{N}]$:
  - **OT Init:** P_a sends $(\textsc{init}, (\mathsf{sid}, \textsc{ot}, a, j))$ to $\mathcal{F}^{\mu}_{\mathsf{rot}}$ and receives choice bits c^a_j in response.
  - **Send Circuit:** P_a chooses random output wire labels d_j, computes $(F^a_j, e_j) \leftarrow \widehat{\mathsf{Gb}}(f', d_j)$ and sends the F^a_j to P_b where $f'(x_a, r, \tilde{x}_b) = f(x_a, Mr \oplus \tilde{x}_b)$ and $r, \tilde{x}_b$ are P_b's inputs. Let e^a_j, e^b_j, e^r_j respectively be the labels encoding $x_a, \tilde{x}_b, r$, for circuit F^a_j and $e^*_{j,t,h}$ index the label of the t^{th} wire with value h in the set e^*_j
  - **Input Commit:** P_a sends the following to $\mathcal{F}_{\mathsf{com}}$:
    - $(\textsc{commit}, (\mathsf{sid}, x_a\text{-}\textsc{input}, a, j, t, h), e^a_{j,t,Mc[t]\oplus h})_{t\in[n],h\in\{0,1\}}$.
    - $(\textsc{commit}, (\mathsf{sid}, x_b\text{-}\textsc{input}, a, j, t, h), e^b_{j,t,h})_{t\in[n],h\in\{0,1\}}$
    - $(\textsc{commit}, (\mathsf{sid}, r\text{-}\textsc{input}, a, j, t, h), e^r_{j,t,h})_{t\in[\mu],h\in\{0,1\}}$
  - **Output Commit:** P_a sends $(\textsc{commit}, (\mathsf{sid}, \textsc{output}, a, j),\ d_j)$ to $\mathcal{F}_{\mathsf{com}}$.
- **Cut-and-Choose:** P_b sends $(\textsc{open}, (\mathsf{sid}, \textsc{cut-and-choose}, b))$ to $\mathcal{F}_{\mathsf{com}}$ and P_a receives σ_b. For $j \in \sigma_b$:
  - **OT Decommit:** P_a sends $(\textsc{open}, (\mathsf{sid}, \textsc{ot}, a, j))$ to $\mathcal{F}^{\mu}_{\mathsf{rot}}$ and P_b receives choice bits c^b_j.
  - **Check Circuit:** P_a sends P_b the d_j and coins used to garble F^a_j. P_b verifies the correctness of F^a_j.
  - **Input Decommit:** Let e^a_j, e^b_j, e^r_j be the verified labels as above.
    - P_a sends $(\textsc{open}, (\mathsf{sid}, x_a\text{-}\textsc{input}, a, j, t, h))_{\forall t,h}$ to $\mathcal{F}_{\mathsf{com}}$ and P_b receives labels e'^a.
    - P_a sends $(\textsc{open}, (\mathsf{sid}, x_b\text{-}\textsc{input}, a, j, t, h))_{\forall t,h}$ to $\mathcal{F}_{\mathsf{com}}$ and P_b receives labels e'^b.
    - P_a sends $(\textsc{open}, (\mathsf{sid}, r\text{-}\textsc{input}, a, j, t, h))_{\forall t,h}$ to $\mathcal{F}_{\mathsf{com}}$ and P_b receives labels e'^r.
    - If there exists a $e'^a_{t,h} \neq e^a_{j,t,Mc^b_j[t]\oplus h}$, or $e'^b \neq e^b_j$, or $e'^r \neq e^r_j$, P_b returns $\textsc{abort}$.
  - **Output:** P_a sends $(\textsc{open}, (\mathsf{sid}, \textsc{output}, a, j))$ to $\mathcal{F}_{\mathsf{com}}$. P_b receives d' and return $\textsc{abort}$ if $d' \neq d_j$.
- **Bucketing:** P_b randomly maps the indices of $[\widehat{N}] - \sigma_b$ into sets $\beta^a_1, ..., \beta^a_N$ s.t. $|\beta^a_i| = B$. For $i \in [N]$:
  - **Bucket Labels:** P_a generates random output labels O^a_i for bucket β^a_i. For $j \in \beta^a_i$, P_a send the output translation $T^a_j := \{O^a_{i,t,h} \oplus d_{j,t,h}\}_{t,h}$ and $H(O^a_{i,t,h})$ to P_b, where d_j are the output labels of F^a_j.
  - **Offline Inputs:**
    - P_a sends $(\textsc{aggregate}, (\mathsf{sid}, \textsc{ot-ag}, a, i), \{(\mathsf{sid}, \textsc{ot}, a, j) | j \in \beta^b_i\})$ to $\mathcal{F}^{\mu}_{\mathsf{rot}}$ and P_b receives the OT aggregation strings δ^a_j for $j \in \beta^a_i$.
    - P_b sends $(\textsc{deliver}, (\mathsf{sid}, \textsc{ot-ag}, a, i), \{e^r_j, w_j | j \in \beta^b_i\})$ to $\mathcal{F}^{\mu}_{\mathsf{rot}}$ where w_j are the decommitment strings to $\{(\mathsf{sid}, r\text{-}\textsc{input}, b, j, t, h)\}_{t,h}$.
    - For $j \in \beta^b_i$, P_a receives X^r_j and W_j from $\mathcal{F}^{\mu}_{\mathsf{rot}}$. P_a send $(\textsc{open}, (\mathsf{sid}, r\text{-}\textsc{input}, b, j, t, c^a_j[t]), W_{j,t})_{\forall t}$ to $\mathcal{F}_{\mathsf{com}}$ and receives X'^r_j. P_a returns $\textsc{abort}$ if $X'^r_j \neq X^r_j$.

Figure 10: Malicious secure online/offline dual-execution 2PC protocol $\Pi_{\mathsf{multi\text{-}sfe}}$.

Execution stage: On common bucket index i and P_a's input x_a.

- **Receiver's Inputs:** Let j' be the first index in β_i^a. P_a sends $\tilde{x}_a := x_a \oplus Mc^a_{j'}$ to P_b where $c^a_{j'}$ are the choice bits of $(\mathsf{sid}, \textsc{ot}, a, j')$. For all $j \in \beta_i^b$:
  - P_b sends $X_j^a := \{e^a_{j,t,(\tilde{x}_a \oplus M\delta_j^a)[t]}\}_t$ and $W_j^a := \{w_{j,(\tilde{x}_a \oplus M\delta_j^a)[t]}\}_t$ to P_a where e_j^a encodes $\tilde{x}_a$ for F_j^b and w_j are the decommitments string to $\{(\mathsf{sid}, x_a\text{-}\textsc{input}, b, j, t, h)\}_{t,h}$.
  - P_a receives X_j^a, W_j^a and sends $(\textsc{open}, (\mathsf{sid}, x_a\text{-}\textsc{input}, b, j, t, (x_a \oplus Mc_j^a)[t]), W^a_{j,t})_{\forall t}$ to $\mathcal{F}_{\mathsf{com}}$ and receives X'^a_j. P_a returns $\textsc{abort}$ if $X_j^a \neq X'^a_j$.
- **Sender's Inputs:** For $j \in \beta_i^b$, P_b sends $(\textsc{open}, (\mathsf{sid}, x_b\text{-}\textsc{input}, b, j, t, (x_b \oplus Mc_j^b)[t]))_{\forall t}$ to $\mathcal{F}_{\mathsf{com}}$. P_a receives the labels X_j^b. P_a returns $\textsc{abort}$ if $\tilde{x}_b \oplus M\delta_j^b \neq (x_b \oplus Mc_j^b)$.
- **Evaluate:** For $j \in \beta_i^b$, let $Y_j := \widehat{\mathsf{Ev}}(F_j^b, (X_j^b, X_j^r, X_j^a))$ with semantic value y_j.
- **PSI Commit:** For $\forall j, t$, if $(H(Y_{j,t}) \neq H(O^b_{i,t,y_j[t]}))$, then $Y_{j,t} \leftarrow \{0,1\}^{k_c}$. Let $I := \{\bigoplus_t Y_{j,t} \oplus T^b_{j,t,y_j[t]} \oplus O^a_{i,t,y_j[t]}\}_{j \in \beta_i^b}$. P_a pads I to size B with random values and sends $(\textsc{input}, (\mathsf{sid}, \textsc{psi}, i), I)$ to $\mathcal{F}_{\mathsf{psi}}$ and receives $(\textsc{input}, P_b)$.
- **Output Decommit:** For $j \in \beta_i^b$, P_b sends $(\textsc{open}, (\mathsf{sid}, \textsc{output}, b, j))$ to $\mathcal{F}_{\mathsf{com}}$ and P_a receives d'_j. If there exists j, j' s.t. $d'_{j,t,h} \oplus T^b_{j,t,h} \neq d'_{j',t,h} \oplus T^b_{j',t,h}$, P_a returns $\textsc{abort}$.
- **PSI Decommit:** P_b sends $(\textsc{open}, (\mathsf{sid}, \textsc{psi}, i))$ to $\mathcal{F}_{\mathsf{psi}}$ and P_a receives the intersection R. If $|R| \neq 1$, P_a returns $\textsc{cheating!}$, else P_A returns y_j s.t. $I_j \in R$.

Figure 11: Malicious secure online/offline dual-execution 2PC protocol $\Pi_{\mathsf{multi\text{-}sfe}}$.

κ_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.

κ_s 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 $2^{-\kappa_s}$.

κ_c is the computational security parameter, that controls the key sizes for OTs, commitments and garbled circuits.

In our evaluations we consider $\kappa_c = 128$, $\kappa_s \in \{40, 80\}$ and $\kappa_b \in \{20, 40, 80\}$. We now prove the security of our protocol:

Theorem 5. *Our protocol (Figure 11) securely realizes the $\mathcal{F}_{\mathsf{multi\text{-}sfe}}$ functionality, in the presence of malicious adversaries.*

Proof. We will show a series of hybrids resulting in a simulator which only depends on the output $f(x, y)$ of each execution and possibly a single bit of leakage. Due to the protocol being symmetric, we will focus on a malicious adversary $\mathcal{A}$.

Hybrid 0 (Real interaction) The simulator runs the protocol of Figure 11 and plays the role of the ideal functionalities. After $\mathcal{A}$ sends their circuits, the simulator extracts the commitments made to the input/output labels and the function computed by each garbled circuit. This combined with the simulator extracting the $\mathcal{F}_{\mathsf{rot}}$ choice bits fully determines which circuits were maliciously generated and where $\mathcal{A}$ provided inconsistent OT aggregation strings and output translation values.

In particular, let a *good bundle* be defined as a garbled circuit and its associated input, output commitments such that if opened in the Cut-and-Choose step the honest party does not abort. That is, there exists a set of random coins and decoding information d_j such that $(F_j^a, e_j) \leftarrow \widehat{Gb}(f, d_j)$ and F_j^a is the provided garbled circuit. The input commitments are to the corresponding encoding labels e_j where the Garbler's input commitments are permuted by their OT choice bits. The output commitments are to the decoding information d_j. Furthermore, let a *bad bundle* be defined to be any garbled circuit and associated input, output commitments which is not a good bundle.

Hybrid 1 (Rig the cut-and-choose challenge) Let $t \leq \widehat{N}$ denote the number of bad bundles and $\mathcal{L}$ be the distribution over $\{0,1\}^N \cup \{\perp\}$ defined as follows: Choose a random cut-and-choose challenge over the adversary's bundles (according to the parameters in the protocol) and random assignment of evaluation bundles into buckets. If any bad bundle is chosen as a checked bundle/circuit, then the outcome of $\mathcal{L}$ is $\perp$. Otherwise the outcome is a string L where $L_i = 1$ iff the ith bucket contains all bad bundles.

By Lemma 1, this distribution indeed satisfies $\Pr_{L \leftarrow \mathcal{L}}[L_i = 1] \leq \epsilon$ for appropriately chosen parameters. As such the simulator sends $(\textsc{cheat}, \mathcal{L})$ to the ideal functionality and receive $(\textsc{cheatresult}, \chi)$. Here χ are the coins used to sample from $\mathcal{L}$, hence χ is exactly the choice of cut-and-choose challenge and assignment into buckets. The simulator therefore uses χ as the honest party's cut-and-choose challenge and assignment into buckets.

The description of the distribution $\mathcal{L}$ exactly matches the distribution of the real interaction. The only change is that the ideal functionality itself chooses the cut-and-choose challenge, rather than the honest party. This hybrid is therefore identically distributed.

Hybrid 2 (Delay/equivocate commitments) The simulator provides empty commitments to the garbled circuits' output labels in the Output Commit step. When specified to open the commitments during the Output Decommit step, the simulator equivocates the commitments to the correct values. Note that this is identical to the previous hybrid except that these values are never used in the simulator until they are opened.

Hybrid 3 (Simulate garbled circuits) After the Cut-and-Choose Commit step, $\mathcal{A}$ has committed to the random coins which determine their cut-and-choose sets. Extract these coins and determine the set of circuits evaluated in the online phase. For all such evaluation circuits, generate simulated garbled circuits and make the corresponding input label commitments be to empty values.

In the online phase, extract $\mathcal{A}$'s effective OT input bits y. If $\mathcal{A}$ lies about the OT aggregation strings, take y to be any input which is consistent between all OT inputs, e.g. the first set of derandomized OT choice bits. Send $(\textsc{input}, y)$ to the ideal functionality and receive $(\textsc{output}, ctr, z)$ in response. Use the output z to generate simulated input labels where $\mathcal{A}$ provided honest OT aggregation values, i.e. do not sample input labels where $\mathcal{A}$ provided inconsistent OT aggregation bits. Equivocate the corresponding input label commitments made in the Input Commit phase and send the input labels to $\mathcal{A}$ using the aggregated $\mathcal{F}_{\mathsf{rot}}$ functionality. For all input wires where the simulator did not sample a label, send uniformly random values. This change is indistinguishable due to $\mathcal{A}$ not having at least one of the corresponding OT messages used to mask the output.

Note that the order in which $\mathcal{A}$'s input is extracted and garbled circuit material is generated matches the garbling scheme security of Definition 6. Furthermore, all output labels which encode output values other than z have not yet been sampled. The simulator will therefore delay the sampling of these output wire labels until they are used in the PSI Commit step and at which point the corresponding commitments will be equivocated. By the security of the garbling scheme, this change is indistinguishable.

Hybrid 4 (Weak PSI Malicious Security) It is now the case that the simulator knows the ideal output value z but still must use the honest parties input x to evaluate the malicious circuits to compute the PSI reconciliation set I. Let us define the distribution D and value a^* by the following process of cases.

- *Good bucket - there exist at least one good bundle*: Set a^* to be the reconciliation value encoding z. Sampling D is then defined as, evaluate the malicious circuits using the honest parties input.

For each set of output labels, translate them to the PSI reconciliation values as specified in the PSI Commit step. Define A to be this set of reconciliation values which do not equal a^*. Pad A with uniform values to size $B-1$. Note that since *Hybrid 3*, the simulator delays the sampling of output wire labels encoding outputs other than z until the PSI Commit step. Therefore each element of A is either uniform or the sum of a uniform output label which is independent of $\mathcal{A}$'s view.

- *Bad bucket - there are all bad bundles:* The simulator asks the ideal functionality for a single bit of leakage by sending $(\textsc{leak}, \exists j : \widetilde{\mathsf{De}}(\mathsf{Ev}(F_j, \mathsf{En}(e_j, x))) = z)$ to the $\mathcal{F}_{\textsf{multi-sfe}}$ functionality where the F_j are the garbled circuit of this bucket and e_j are the corresponding input labels extracted from $\mathcal{F}_{\textsf{rot}}$. In response the simulator receives the leakage $(\textsc{cheatresult}, p)$. If $p = 1$ the simulator sets a^*, D to be as above. Otherwise, the simulator artificially sets a^* to be the PSI reconciliation value of the first circuit in the bucket and defines D to be as above. Note that when $p = 0$, a^* is the sum of at least one output label which is uniform and independent of $\mathcal{A}$'s view.

By the process above, it is indeed the case that "$A \leftarrow D$; output a uniformly chosen element of A" yields a uniformly distributed element over $\{0,1\}^\ell$. In particular, each such element of A is either uniform or the sum of bucket wide output labels for which at least one term is independent of $\mathcal{A}$'s view. Moreover, a^* is the only "predictable" value and only when it is the reconciliation value encoding z.

The Simulator will then use $I = A \cup \{a^*\}$ as input to the PSI subprotocols. Furthermore, this process of determining A, a^* matches the security requirements of a weakly malicious-secure PSI specified in Definition 3. We can therefore substitute the PSI subprotocol with the ideal two-phase PSI functionality $\mathcal{F}_{\textsf{psi}}^{B,\ell}$ defined in Figure 5.

Hybrid 5 (Removing PSI Inputs) Observe that the ideal PSI outputs either the singleton set containing the reconciliation value encoding z or the empty set. Moreover, the outcome of this is determined by the existence of a circuit which evaluates to z. The simulator therefore need not evaluate the circuits sent by $\mathcal{A}$ using the honest party's input. Instead the simulator will perform the following, if the current bucket has at least a single honest circuit, the simulator will input the reconciliation value encoding z and uniformly random values everywhere else. Otherwise the simulator will ask the ideal functionality for the single bit of leakage as in *Hybrid 4*. If leaked bit is $p = 1$, the simulator will provide PSI input as above. Otherwise the simulator provides PSI inputs which are uniformly random. By the security of the ideal PSI functionality and the intersection being identical to before, this change is indistinguishable.

Hybrid 6 (Uniform Inputs) The simulator is no longer evaluating any circuits and therefore can request uniformly distributed inputs to evaluate $\mathcal{A}$'s circuits on. Recall that the simulator's offline input Mc_j is uniformly distributed in $\mathcal{A}$'s view by the security of the k-probe resistant matrix. Requesting uniformly random inputs in the online phase is therefore indistinguishable.

Hybrid 7 (Inconsistent translation values) Consider the case where $\mathcal{A}$ provides malicious output label translation values T during the Bucket Labels phase. These values allow the evaluator to translate circuit output labels to common bucket labels in the PSI Commit Step. Since these values are provided after the cut and choose, $\mathcal{A}$ need not provide honest values. However, the simulator will always abort in the Output Decommit step of the online phase in this event. Therefore, the simulator need not send $\mathcal{A}$'s extracted input to the ideal functionality for any buckets with malicious translation values. Instead, the simulator can use the zero string for z. Due to the simulator aborting before the PSI Decommit phase, this simulator is indistinguishable from the previous.

Hybrid 8 (Ideal Simulation) Now observe that the simulation of the ith execution only depends on whether the simulated party's ith bucket holds a garbled circuit which if evaluated would result in a PSI input encoding with semantic value $f(x, y)$. It follows that the simulator knows this information: if $L_i = 1$ (bad bucket) then the ideal functionality informs the simulator if such a circuit exists. Otherwise, $L_i = 0$ and there does exist at least one good circuit in this bucket.

In summary, the simulator implicit in final hybrid only uses the output of the ideal functionality. In particular, $\mathcal{A}$ evaluates simulated garbled circuits with fixed output $f(x, y)$ and the simulator does not use the honest parties input. If $\mathcal{A}$ successfully cheats by generating a bad bucket, then only a single additional bit of information is leaked about the honest parties input x. Therefore the protocol leak no more information than the ideal functionality in Figure 2. □

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.* [KOS15] protocol for OT extension. Starting from 128 base OTs (computed using the protocol of [PVW08]), 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 [ZRE15]. The implementation utilizes the hardware accelerated AES-ni instruction set and uses fixed-key AES as the gate-level cipher, as suggested by [BHKR13]. 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:

PSI		Async		Sync	
κ_s	B	Time	Size	Time	Size
40	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	5	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

Figure 12: The running time (ms) and online communication size (bytes) of the two PSI protocols when executed with κ_s-bit strings and input sets of size B.

- 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'ed 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 specified in Figure 17. 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 3 rounds total, and an *asynchronous* protocol that uses 2 rounds total (at higher communication cost). We now discuss the tradeoffs between these two PSI protocols. A summary is given in Figure 12. 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 $O(B\kappa_s + B^2\kappa_c)$. In the remaining comparisons, we always use the PSI protocol with lowest latency, according to Figure 12.

9.2 Comparison to the LR protocol

We compare our prototype to that of [LR15] with 40-bit security. That is, we use $\kappa_b = \kappa_s = 40$; both protocols have identical security and use the same bucket size. We use identical AWS instances and a similar number of threads to those reported in [LR15].

Figure 13 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 [LR15].

In the WAN setting with small circuits such as AES where the input size is minimal we see [LR15] 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

[7] The circuit is not optimized; each call to AES recomputes the entire key schedule.

$\kappa_s = \kappa_b = 40$			[LR15]		This	
	Circuit	N	Offline	Online	Offline	Online
LAN	AES	32	197	12	45	1.7
		128	114	10	16	1.5
		1024	74	7	5.1	1.3
	SHA256	32	459	50	136	10.0
		128	275	40	78	8.8
		1024	206	33	48	8.4
WAN	AES	32	1,126	163	282	190
		128	919	164	71	191
		1,024	760	160	34	189
	SHA256	32	3,638	290	777	194
		128	3,426	256	399	192
		1,024	2,992	207	443	191

Figure 13: Amortized running times per execution (reported in ms) for [LR15] and our prototype. We used bucket size $B = 6, 5, 4$ for $N = 32, 128, 1024$.

		$\kappa_b = \kappa_s = 80$			$\kappa_b = \kappa_s = 40$			$\kappa_b = 20; \kappa_s = 40$		
Circuit	N	Storage	Offline	Online	Storage	Offline	Online	Storage	Offline	Online
AES	32	0.21	69	2.3	0.12	45	1.7	0.06	40	1.1
	128	0.88	25	2.1	0.32	16	1.4	0.38	16	1.1
	1,024	6.8	16	1.8	1.6	5.1	1.3	0.76	2.4	1.0
SHA-256	32	6.8	234	15.7	1.3	136	10.0	0.68	65	7.6
	128	8.7	190	12.3	3.5	78	8.8	4.4	95	6.4
	1,024	62.1	131	11.4	15.6	48	8.4	8.8	24	6.3
2048 CBC-MAC	32	3.8	621	22.7	2.4	655	14.9	1.2	247	11.1
	128	15.4	450	18.1	6.2	191	13.4	7.9	246	10.6
	1,024	109.5	378	15.8	31.0	95	12.3	15.6	71	10.6

Figure 14: Amortized running times per execution (reported in ms) and total offline storage (reported in GB) for our prototype in the LAN configuration. The peak offline storage occurs before the cut and choose, consisting of the circuits, commitments, and OT messages. For $\kappa_b = 80$ we use parameters $(N, B) \in \{(32, 12), (128, 9), (1024, 7)\}$. For $\kappa_b = 40$ we use parameters $(N, B) \in \{(32, 6), (128, 5), (1024, 5)\}$. For $\kappa_b = 20$ we use parameters $(N, B) \in \{(32, 3), (128, 2), (1024, 2)\}$.

that of [LR15] by 16 to 100ms per execution and we achieve a much more efficient 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 [LR15] 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 14 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 [LR15]. 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

		LAN		WAN	
κ_s	B	Time	Bandwidth	Time	Bandwidth
40	2	0.26	327	0.63	144
	3	0.41	353	0.72	206
	4	0.56	381	1.01	213
	6	0.82	465	1.32	293
80	5	0.75	568	1.39	300
	7	1.01	725	2.02	366
	9	2.42	465	3.41	331

Figure 15: Maximum amortized throughput (ms/execution) and resulting bandwidth (Kbps) when performing many parallel evaluations of AES with the given bucket size B and statistical security κ_s.

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 κ_s only incurs an approximate 25% to 50% increase in running time. This is contrasted by [LR15] 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. [LR15] 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 15 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 0.26ms 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 0.72ms per AES evaluation with 40-bit security.

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.

[Bea95] Donald Beaver. Precomputing oblivious transfer. In Don Coppersmith, editor, *CRYPTO'95*, volume 963 of *LNCS*, pages 97–109. Springer, Heidelberg, August 1995.

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

[BHR12a] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 134–153. Springer, Heidelberg, December 2012.

[BHR12b] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12*, pages 784–796. ACM Press, October 2012.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[CG13] Ran Canetti and Juan A. Garay, editors. *CRYPTO 2013, Part II*, volume 8043 of *LNCS*. Springer, Heidelberg, August 2013.

[CO15] Tung Chou and Claudio Orlandi. The simplest protocol for oblivious transfer. In Kristin E. Lauter and Francisco Rodríguez-Henríquez, editors, *Progress in Cryptology - LATINCRYPT 2015*, volume 9230 of *Lecture Notes in Computer Science*, pages 40–58. Springer, 2015.

[DLT14] Ivan Damgaard, Rasmus Lauritsen, and Tomas Toft. An empirical study and some improvements of the MiniMac protocol for secure computation. Cryptology ePrint Archive, Report 2014/289, 2014. http://eprint.iacr.org/2014/289.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Safavi-Naini and Canetti [SNC12], pages 643–662.

[DZ13] Ivan Damgård and Sarah Zakarias. Constant-overhead secure computation of boolean circuits using preprocessing. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 621–641. Springer, Heidelberg, March 2013.

[DZ15] Ivan Damgård and Rasmus Winther Zakarias. Fast oblivious AES: a dedicated application of the MiniMac protocol. Cryptology ePrint Archive, Report 2015/989, 2015. ia.cr/2015/989.

[FJN14] Tore Kasper Frederiksen, Thomas P. Jakobsen, and Jesper Buus Nielsen. Faster maliciously secure two-party computation using the GPU. In Michel Abdalla and Roberto De Prisco, editors, *SCN 14*, volume 8642 of *LNCS*, pages 358–379. Springer, Heidelberg, September 2014.

[GG14] Juan A. Garay and Rosario Gennaro, editors. *CRYPTO 2014, Part II*, volume 8617 of *LNCS*. Springer, Heidelberg, August 2014.

[HKE13] Yan Huang, Jonathan Katz, and David Evans. Efficient secure two-party computation using symmetric cut-and-choose. In Canetti and Garay [CG13], pages 18–35.

[HKK+14] Yan Huang, Jonathan Katz, Vladimir Kolesnikov, Ranjit Kumaresan, and Alex J. Malozemoff. Amortizing garbled circuits. In Garay and Gennaro [GG14], pages 458–475.

[KMRR15] Vladimir Kolesnikov, Payman Mohassel, Ben Riva, and Mike Rosulek. Richer efficiency/security trade-offs in 2PC. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, *TCC 2015, Part I*, volume 9014 of *LNCS*, pages 229–259. Springer, Heidelberg, March 2015.

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

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

[KSS12] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. Billion-gate secure computation with malicious adversaries. In Tadayoshi Kohno, editor, *Proceedings of the 21th USENIX Security Symposium*, pages 285–300. USENIX Association, 2012.

[Lin13] Yehuda Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In Canetti and Garay [CG13], pages 1–17.

[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.

[LR14] Yehuda Lindell and Ben Riva. Cut-and-choose Yao-based secure computation in the online/offline and batch settings. In Garay and Gennaro [GG14], pages 476–494.

[LR15] Yehuda Lindell and Ben Riva. Blazing fast 2PC in the offline/online setting with security for malicious adversaries. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 579–590. ACM, 2015.

[MF06] Payman Mohassel and Matthew Franklin. Efficiency tradeoffs for malicious two-party computation. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 458–473. Springer, Heidelberg, April 2006.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In Matt Blaze, editor, *Proceedings of the 13th USENIX Security Symposium*, pages 287–302. USENIX, 2004.

[MR13] Payman Mohassel and Ben Riva. Garbled circuits checking garbled circuits: More efficient and secure two-party computation. In Canetti and Garay [CG13], pages 36–53.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Safavi-Naini and Canetti [SNC12], pages 681–700.

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

[PSZ14] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on OT extension. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium*, pages 797–812. USENIX Association, 2014.

[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.

[Sma15] Nigel Smart. Personal communication, November 2015.

[SNC12] Reihaneh Safavi-Naini and Ran Canetti, editors. *CRYPTO 2012*, volume 7417 of *LNCS*. Springer, Heidelberg, August 2012.

[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.

[sS13] abhi shelat and Chih-Hao Shen. Fast two-party secure computation with minimal assumptions. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13*, pages 523–534. ACM Press, November 2013.

[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.

A Adaptively Secure Garbling Schemes

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

- $\mathsf{Gb}(f, d) \to (F, e)$; 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 $\{0,1\}^{\kappa_c}$). We deviate from [BHR12b] in requiring the output wire labels d to be chosen by the caller of Gb, rather than chosen by Gb itself. In the notation of [BHR12b], 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.

- $\mathsf{En}(e, x) \to X$ 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 $X_i = e[i, x_i]$.

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

- $\widetilde{\mathsf{De}}(Y) \to y$. We assume a way to decode a garbled output to a plaintext value. It is a deviation from [BHR12b] 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 $\mathsf{Ev}(F, \mathsf{En}(e, x)) = \mathsf{En}(d, f(x))$ and $\widetilde{\mathsf{De}}(\mathsf{Ev}(F, \mathsf{En}(e, x))) = f(x)$ 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 ciruict 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 En. Since En is projective, we write $\mathsf{En}(e, i, b)$ to denote the component $e_{i,b}$ — that is, the garbled input for the ith wire corresponding to truth value b. Recall that we also garble a circuit with output wire labels d specified (rather than chosen by the Gb algorithm). Our security definition lets the adversary choose d.

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

$\underline{\mathcal{G}_{\text{real}}^{\text{Adv}}}$:	$\underline{\mathcal{G}_{\text{ideal}}^{\text{Adv},S}}$:

get f, d from Adv^H
$(F, e) \leftarrow \mathsf{Gb}(f, d)$
give F to Adv^H
for $i = 1$ to m:
    get x_i from Adv^H
    $X_i \leftarrow \mathsf{En}(e, i, x_i)$
    give X_i to Adv^H
Adv^H outputs a bit
	get f, d from Adv^{S_0}
	$F \leftarrow S_1(f)$
	give F to Adv^{S_0}
	for $i = 1$ to $m - 1$:
	    get x_i from Adv^{S_0}
	    $X_i \leftarrow S_2(i)$
	    give X_i to Adv^{S_0}
	get x_m from Adv^{S_0}
	$y = f(x_1 \cdots x_m)$
	$Y \leftarrow \mathsf{En}(d, y)$
	$X_m \leftarrow S_2(m, y, Y)$
	give X_m to Adv^{S_0}
	Adv^{S_0} outputs a bit

In $\mathcal{G}_{\textit{ideal}}$*,* H *is a random oracle. In* $\mathcal{G}_{\textit{ideal}}$*, 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 Adv, we have that*

$$\left| \Pr[\mathcal{G}_{\textit{real}}^{\textit{Adv}} \textit{ outputs } 1] - \Pr[\mathcal{G}_{\textit{ideal}}^{\textit{Adv},S} \textit{ outputs } 1] \right|$$

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* $\mathsf{En}(d, f(x))$. 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 16 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 [BHR12a], with some small changes. First, since we know in advance which order 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.

$$
\begin{array}{l}
\underline{\widehat{\mathsf{Gb}}(f, \widehat{d}):} \\
\quad (F, e, d) \leftarrow \mathsf{Gb}(f) \\
\quad R \leftarrow \{0,1\}^\kappa \\
\quad \text{for each output wire } i\text{:} \\
\quad\quad \delta_i^b \leftarrow H(R \| \mathtt{out} \| i \| b \| d_i^b) \oplus \widehat{d}_i^b \\
\quad \widehat{F} \leftarrow (F \oplus H(R \| \mathtt{gc}), \{\delta_i^b\}) \\
\quad \widehat{e} \leftarrow (e_1^0, e_1^1, e_2^0, e_2^1, \ldots, e_m^0 \| R, e_m^1 \| R) \\
\quad \text{return } (\widehat{F}, \widehat{e}) \\
\\
\underline{\widehat{\mathsf{Ev}}(\widehat{F}, \widehat{X}):} \\
\quad \text{parse } \widehat{X}_m \text{ as } X_m \| R \text{ and } \widehat{F} \text{ as } (F', \delta) \\
\quad X \leftarrow (\widehat{X}_1, \widehat{X}_2, \ldots, X_m) \\
\quad Y \leftarrow \mathsf{Ev}(F' \oplus H(R \| \mathtt{gc}), X) \\
\quad y \leftarrow \widetilde{\mathsf{De}}(Y) \\
\quad \text{for each output wire } i\text{:} \\
\quad\quad \widehat{Y}_i = \delta_i^{y_i} \oplus H(R \| \mathtt{out} \| i \| y_i \| Y_i) \\
\quad \text{return } \widehat{Y}
\end{array}
$$

Figure 16: Transformation from a static-secure doubly-projective garbling scheme $(\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De}, \widetilde{\mathsf{De}})$ to one satisfying Definition 6.

Theorem 7. *If* $(\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De}, \widetilde{\mathsf{De}})$ *is a doubly-projective garbling scheme satisfying the (static) prv and aut properties of [BHR12b] then the scheme in Figure 16 satisfies adaptive security notion of Definition 6 in the random oracle model.*

The proof is very similar to analogous proofs in [BHR12a]. The main idea is that the simulator can choose the "masked" $\widehat{F}$ and δ 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.*, [ZRE15].

κ_b = 40	B = 3	4	5	6	7	8	9	10	11
N = 8	–	–	–	307 (38.38)	203 (25.38)	161 (20.12)	143 (17.88)	136 (17.00)	135 (16.88)
16	–	–	–	344 (21.50)	253 (15.81)	222 (13.88)	214 (13.38)		
32	–	–	708 (22.12)	430 (13.44)	361 (11.28)	349 (10.91)			
64	–	2268 (35.44)	836 (13.06)	615 (9.61)	583 (9.11)				
128	–	2321 (18.13)	1137 (8.88)	995 (7.77)					
256	–	2730 (10.66)	1764 (6.89)	1760 (6.88)					
1024	18756 (18.32)	5664 (5.53)	5593 (5.46)						
2048	20066 (9.80)	9730 (4.75)							
4096	25099 (6.13)	17906 (4.37)							
8192	36743 (4.49)	34282 (4.18)							
16384	60961 (3.72)								
32768	109908 (3.35)								

κ_b = 20	B = 2	3	4	5	6
N = 8	–	–	84 (10.50)	70 (8.75)	69 (8.62)
16	–	186 (11.62)	113 (7.06)	109 (6.81)	
32	–	224 (7.00)	176 (5.50)		
64	–	313 (4.89)	303 (4.73)		
128	1449 (11.32)	500 (3.91)			
256	1476 (5.77)	882 (3.45)			
1024	2865 (2.80)				
2048	4883 (2.38)				
4096	8962 (2.19)				
8192	17146 (2.09)				
16384	33525 (2.05)				
32768	66291 (2.02)				

Figure 17: For each N, B, κ_b, the table shows the minimum number $\widehat{N}$ of circuits that must be generated in the offline phase to ensure that the probability of a particular execution leaking a bit is at most $2^{-\kappa_b}$. The number in parentheses is the ratio $\widehat{N}/N$, *i.e.*, the amortized number of total (offline+online) circuits needed per execution. An empty entry offers no benefit over the configurations to the left (both N and B are higher than another configuration). An "–" entry means that the ratio $\widehat{N}/N > \kappa_b$, so that batching N executions is no better (in terms of total offline+online cost) than doing N executions in isolation. (Figure continues on next page.)

κ_b = 80	B = 4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
N = 8	–	–	–	–	–	–	–	–	575 (71.88)	460 (57.50)	390 (48.75)	345 (43.12)	315 (39.38)	296 (37.00)	283 (35.38)	275 (34.38)	270 (33.75)	268 (33.50)
16	–	–	–	–	–	–	1203 (75.19)	845 (52.81)	660 (41.25)	556 (34.75)	496 (31.00)	461 (28.81)	441 (27.56)	430 (26.88)	426 (26.62)			
32	–	–	–	–	–	2041 (63.78)	1327 (41.47)	1004 (31.38)	843 (26.34)	759 (23.72)	717 (22.41)	699 (21.84)	695 (21.72)					
64	–	–	–	–	3930 (61.41)	2263 (35.36)	1623 (25.36)	1344 (21.00)	1220 (19.06)	1171 (18.30)	1163 (18.17)							
128	–	–	–	9320 (72.81)	4325 (33.79)	2798 (21.86)	2244 (17.53)	2040 (15.94)	1984 (15.50)									
256	–	–	–	9832 (38.41)	5239 (20.46)	3917 (15.30)	3513 (13.72)	3443 (13.45)										
1024	–	–	32393 (31.63)	14637 (14.29)	11263 (11.00)	10795 (10.54)												
2048	–	–	37215 (18.17)	21644 (10.57)	19426 (9.49)													
4096	–	170735 (41.68)	48680 (11.88)	35882 (8.76)	35795 (8.74)													
8192	–	181418 (22.15)	72691 (8.87)	64499 (7.87)														
16384	–	214407 (13.09)	121490 (7.42)															
32768	2320882 (70.83)	291603 (8.90)	219599 (6.70)															

(Figure 17 continued)