6

Malicious Security

So far, we have focused on semi-honest protocols which provide privacy and security only against passive adversaries who follow the protocol as specified. The semi-honest threat model is very weak. It makes assumptions that underestimate the power of realistic adversaries, for most scenarios. This chapter discusses several protocols, summarized in Figure 6.1, that are designed to resist malicious adversaries.

6.1 Cut-and-Choose

Yao's GC protocol is not secure against malicious adversaries. In particular, P_1 is supposed to generate and send a garbled version of $\mathcal{F}$ to P_2. A malicious

Protocol	Parties	Rounds	Based on
Cut-and-Choose (§6.1–6.4)	2	constant	Yao's GC (§3.1)
GMW compiler (§6.5.1)	many	(inherited)	any semi-honest
BDOZ & SPDZ (§6.6)	many	circuit depth	preprocessing
Authenticated garbling (§6.7)	many	constant	BMR (§3.5)

Figure 6.1: Summary of malicious MPC protocols discussed in this chapter.

P_1 may send the garbling of a different circuit that P_2 had not agreed to evaluate, but P_2 has no way to confirm the circuit is correct. The output of the maliciously-generated garbled circuit may leak more than P_2 has agreed to reveal (for instance, P_2's entire input).

Main idea: check some circuits, evaluate others. The standard way to address this problem is a technique called *cut-and-choose*, a general idea that goes back at least to Chaum (1983), who used it to support blind signatures.

To use cut-and-choose to harden Yao's GC protocol, P_1 generates many independent garbled versions of $\mathcal{F}$ and sends them to P_2. P_2 then chooses some random subset of these circuits and asks P_1 to "open" them by revealing all of the randomness used to generate the chosen circuits. P_2 then verifies that each opened garbled circuit is a correctly garbled version of the agreed-upon circuit $\mathcal{F}$. If any of the opened circuits are found to be generated incorrectly, P_2 knows P_1 has cheated and can abort. If all of the opened circuits are verified as correct, P_2 continues the protocol. Since the opened circuits have had all of their secrets revealed, they cannot be used for the secure computation. However, if all of the opened circuits were correct, P_2 has some confidence that most of the unopened circuits are also correct. These remaining circuits can then be evaluated as in the standard Yao protocol.

Generating an output. Cut-and-choose cannot ensure with non-negligible probability that *all* unopened circuits are correct. Let's say that P_1 generates s garbled circuits, and each one is checked with independent probability $\frac{1}{2}$. Then if P_1 generates only one of the circuits incorrectly, then with (non-negligible) probability $\frac{1}{2}$ that circuit will not be chosen for checking and will become one of the evaluation circuits. In this event P_2 may get inconsistent outputs from the evaluated circuits.

If P_2 sees inconsistent outputs, then it is obvious to P_2 that P_1 is misbehaving. It is tempting to suggest that P_2 should abort in this case. However, to do so would be insecure! Suppose P_1's incorrect circuits are designed to be selectively incorrect in a way that *depends on* P_2*'s input.* For example, suppose an incorrect circuit gives the wrong answer if the first bit of P_2's input is 1. Then, P_2 will only see disagreeing outputs if its first bit of input is 1. If P_2 aborts in this case, the abort will then leak the first bit of its input. So we are in a situation where P_2 knows for certain that P_1 is cheating, but must continue as if there was no problem to avoid leaking information about its input.

Traditionally, cut-and-choose protocols address this situation by making P_2 consider only the *majority* output from among the evaluated circuits. The cut-and-choose parameters (number of circuits, probability of checking each circuit) are chosen so that

$$\Pr[\textit{majority of evaluated circuits incorrect} \ \wedge \textit{all check circuits correct}]$$

is negligible. In other words, if all of the check circuits are correct, P_2 can safely assume that the majority of the evaluation circuits are correct too. This justifies the choice to use the majority output.

Input consistency. In cut-and-choose-based protocols, P_2 evaluates several garbled circuits. The fact that there are several circuits presents an additional problem: a malicious party may try to use *different inputs* to different garbled circuits. *Input consistency* refers to the problem of ensuring that both parties use the same input to all circuits.

Ensuring consistency of P_2's inputs (i.e. preventing a malicious P_2 from submitting different inputs) is generally easier. Recall that in Yao's protocol, P_2 picks up garbled input using oblivious transfer (OT) as receiver. The parties can perform a single OT for each bit of P_2's input, whose payload is the garbled input corresponding to that bit, for *all* circuits. Because P_1 prepares the OT payloads, they can ensure that P_2 only receives garbled inputs corresponding to the same input for all circuits.

Ensuring input consistency for P_1 is more challenging. One approach (see Section 6.8 for others), proposed by shelat and Shen (2011), has the parties evaluate the function $((x, r), y) \mapsto (\mathcal{F}(x, y), H(x, r))$ where H is a 2-universal hash function. The 2-universal property is that for all $z \neq z'$, $\Pr[H(z) = H(z')] = \frac{1}{2^\ell}$, where the probability is over the random choice of H and ℓ is the output length of H in bits. The idea is to make P_1 first commit to its (garbled) inputs; then P_2 chooses a random H which determines the circuit to garble. Since P_1's inputs are fixed before H is chosen, any inconsistent inputs will lead to different outputs from H, which P_2 can detect. In order to ensure that the H-output does not leak information about P_1's input x, P_1 also includes additional randomness r as argument to H, which hides x when r

is sufficiently long. As shown by shelat and Shen (2011), multiplication by a random Boolean matrix is a 2-universal hash function. When such a function H is made public, the resulting circuit computing H consists of exclusively XOR operations, and therefore adds little cost to the garbled circuits when using the FreeXOR optimization.

Selective abort. Another subtle issue is even if all garbled circuits are correct, P_1 may still cheat by providing incorrect garbled inputs in the oblivious transfers. Hence it does not suffice to just check the garbled circuits for correctness. For instance, P_1 may select its inputs to the OT so that whenever P_2's first input bit is 1, P_2 will pick up garbage wire labels (and presumably abort, leaking the first input bit in the process). This kind of attack is known as a *selective abort* attack (sometimes called *selective failure*). More generally, we care about when P_1 provides *selectively* incorrect garbled inputs in some OTs (e.g., P_1 provides a correct garbled input for 0 and incorrect garbled input for 1), so that whether or not P_2 receives incorrect garbled inputs depends on P_2's input.

An approach proposed by Lindell and Pinkas (2007) (and improved in shelat and Shen (2013)) uses what are called *k-probe-resistant* matrices. The idea behind this technique is to agree on a public matrix M and for P_2 to randomly encode its true input y into $\tilde{y}$ so that $y = M\tilde{y}$. Then the garbled circuit will compute $(x, \tilde{y}) \mapsto \mathcal{F}(x, M\tilde{y})$ and P_2 will use the bits of $\tilde{y}$ (rather than y) as its inputs to OT. The k-probe-resistant property of M is that for any nonempty subset of rows of M, their XOR has Hamming weight at least k. Lindell and Pinkas (2007) show that if M is k-probe-resistant, then the joint distribution of any k bits of $\tilde{y}$ is uniform—in particular, it is independent of P_2's true input y. Furthermore, when M is public, the computation of $\tilde{y} \mapsto M\tilde{y}$ consists of only XOR operations and therefore adds no cost to the garbled circuit using FreeXOR (though the increased size of $\tilde{y}$ contributes more oblivious transfers).

The k-probe-resistant encoding technique thwarts selective abort attacks in the following way. If P_1 provides selectively incorrect garbled inputs in at most k OTs, then these will be selectively picked up by P_2 according to at most k specific bits of $\tilde{y}$, which are completely uniform in this case. Hence, P_2's abort condition is input-independent. If on the other hand P_1 provides incorrect garbled inputs in more than k OTs, then P_2 will almost surely abort—at least with probability $1 - 1/2^k$. If k is chosen so that $1/2^k$ is negligible (e.g., if

$k = \sigma$, the statistical security parameter), then any input-dependent differences in abort probability are negligible.

Concrete parameters. Cut-and-choose mechanisms involve two main parameters: the *replication factor* is the number of garbled circuits that P_1 must generate, and the *checking probability* refers to the probability with which a garbled circuit is chosen to be checked in the cut-and-choose phase. An obvious goal for any cut-and-choose protocol is to minimize the replication factor needed to provide adequate security.

In the cut-and-choose protocol described above, the only way an adversary can break security of the mechanism is to cause a majority of evaluation circuits to be incorrect, while still making all checked circuits correct. The adversary's task can be captured in the following abstract game:

- The player (arbitrarily) prepares ρ balls, each one is either red or green. A red ball represents an incorrectly-garbled circuit while a green ball represents a correct one.
- A random subset of exactly c balls is designated to be *checked.* If any checked ball is red, the player loses the game.
- The player wins the game if the majority of the unchecked balls are red.

We want to find the smallest ρ and best c so that no player can win the game with probability better than $2^{-\lambda}$. The analysis of shelat and Shen (2011) found that the minimal replication factor is $\rho \approx 3.12\lambda$, and the best number of items to check is $c = 0.6\rho$ (surprisingly, not 0.5ρ).

Cost-aware cut-and-choose. The results from shelat and Shen (2011) provide an optimal number of check and evaluation circuits, assuming the cost of each circuit is the same. However, some circuits are evaluated and others are checked, and these operations do not have equal cost. In particular, the computational cost of evaluating a garbled circuit is about 25–50% the cost of checking a garbled circuit for correctness since evaluating just involves executing one path through the circuit, while checking must verify all entries in the garble tables. Also, some variants of cut-and-choose (e.g., Goyal *et al.* (2008)) allow P_1 to send only a hash of a garbled circuit up-front, before P_2

chooses which circuits to open. To open a circuit, P_1 can simply send a short seed that was used to derive all the randomness for the circuit. P_2 can then recompute the circuit and compare its hash to the one originally sent by P_1. In protocols like this, the communication cost of a checked circuit is almost nothing—only evaluation circuits require significant communication.

Zhu *et al.* (2016) study various cut-and-choose games and derive parameters with optimal cost, accounting for the different costs of checking and evaluating a garbled circuit.

6.2 Input Recovery Technique

In the traditional cut-and-choose mechanisms we have described so far, the evaluator (P_2) evaluates many garbled circuits and reports the majority output. As previously mentioned, the overhead of this method is well understood—the replication factor for security $2^{-\lambda}$ is roughly 3.12λ. Reducing this replication factor requires a different approach to the entire cut-and-choose mechanism.

Lindell (2013) and Brandão (2013) independently proposed cut-and-choose protocols breaking this replication factor barrier. These protocols give P_2 a way of identifying the correct output in the event that some of the evaluated circuits disagree. Hence, the only way for a malicious P_1 to break security is to force *all* of the evaluated circuits to be incorrect in the same way (rather than simply forcing a *majority* of them to be incorrect as in the previous protocols). Suppose there are ρ circuits, and each one is checked with independent probability $\frac{1}{2}$. The only way to cheat is for all of the correct circuits to be opened and all of the incorrect circuits to be evaluated, which happens with probability $2^{-\rho}$. In short, one can achieve $2^{-\lambda}$ security with replication factor only $\rho = \lambda$ rather than $\rho \approx 3.12\lambda$. The protocol of Lindell (2013) proceeds in two phases:

1. The parties do a fairly typical cut-and-choose with P_1 generating many garbled circuits, and P_2 choosing some to check and evaluating the rest. Suppose P_2 observes different outputs from different garbled circuits. Then, P_2 obtains output wire labels for the same wire corresponding to opposite values (e.g., a wire label encoding 0 on the first output wire of one circuit, a label encoding 1 on the first output wire of another circuit). When P_1 is honest, it is infeasible to obtain such contradictory wire labels. Hence, the wire labels serve as "proof of cheating". But,

for the same reasons as mentioned above, P_2 must not reveal whether it obtained such a proof since that event may be input-dependent and leak information about P_2's private input.

2. In the second phase, the parties do a malicious-secure computation of an *input recovery* function: if P_2 can provide proof of cheating in phase 1, then the function "punishes" P_1 by revealing its input to P_2. In that case, P_2 has both parties' inputs and can simply compute the correct output locally. Otherwise, when P_2 does not provide proof of cheating, P_2 learns nothing from this second phase. Either way, P_1 learns nothing from this second phase.

There are many subtle details that enable this protocol to work. Some of the most notable are:

- The secure computation in the second phase is done by using a traditional (majority-output) cut-and-choose protocol. However, the size of this computation can be made to depend only on the size of P_1's input. In particular, it does not depend on the size of the circuit the parties are evaluating in phase 1.

- In order to make the circuits for the second phase small, it is helpful if all garbled circuits share the same output wire labels. When this is the case, opening any circuit would reveal all output wire labels for all evaluation circuits and allows P_2 to "forge" a proof of cheating. Hence the check circuits of phase 1 cannot be opened until the parties' inputs to phase 2 have been fixed.

- The overall protocol must enforce that P_1 uses the same input to all circuits *in both phases*. It is important that if P_1 uses input x in phase 1 and cheats, it cannot prevent P_2 from learning that same x in phase 2. Typical mechanisms for input consistency (such as the 2-universal hash technique described above) can easily be adapted to ensure consistency across both phases in this protocol.

- For the reasons described previously, P_2 cannot reveal whether it observed P_1 cheating in the first phase. Analogously, the protocol gives P_2 two avenues to obtain the final output (either when all phase-one circuits agree, or by recovering P_2's input in phase two and computing the output directly), but P_1 cannot learn which one was actually used. It follows that the parties must *always* perform the second phase, even when P_1 is caught cheating.

6.3 Batched Cut-and-Choose

As a motivating scenario, consider the case where two parties know in advance that they would like to perform N secure evaluations of the same function f (on unrelated inputs). In each secure computation, P_1 would be required to generate many garbled circuits of f for each cut-and-choose. The amortized costs for each evaluation can be reduced by performing a single cut-and-choose for all N evaluation instances.

Consider the following variant of the cut-and-choose abstract game:

1. The player (arbitrarily) prepares $N\rho + c$ balls, each one is either red or green.
2. A random subset of exactly c balls is designated to be *checked.* If any checked ball is red, the player loses the game.
3. [new step] The unchecked balls are randomly assigned into N buckets, with each bucket containing exactly ρ balls.
4. [modified step] The player wins if any bucket contains *only* red balls (in a different variant, one might specify that the player wins if any bucket contains a majority of red balls).

This game naturally captures the following high-level idea for a cut-and-choose protocol suitable for a batch of N evaluations of the same function. First, P_1 generates $N\rho + c$ garbled circuits. P_2 randomly chooses c of them to be checked and randomly assigns the rest into N buckets. Each bucket contains the circuits to be evaluated in a particular instance. Here we are assuming that each instance will be secure as long as it includes at least one correct circuit (for example, using the mechanisms from Section 6.2).

Intuitively, it is now harder for the player (adversary) to beat the cut-and-choose game, since the evaluation circuits are further randomly assigned to buckets. The player must get lucky not only in avoiding detection during checking, but also in having many incorrect circuits placed in the same bucket.

Zhu and Huang (2017) give an asymptotic analysis showing that replication $\rho = 2 + \Theta(\lambda/\log N)$ suffices to limit the adversary to success probability $2^{-\lambda}$. Compare this to single-instance cut-and-choose which requires replication factor $O(\lambda)$.[1] The improvement over single-instance cut-and-choose is not just asymptotic, but is significant for reasonable values of N. For instance, for $N = 1024$ executions, one achieves a security level of 2^{-40} if P_1 generates 5593 circuits, of which only 473 are checked. Then only $\rho = 5$ circuits are evaluated in each execution.

Lindell and Riva (2014) and concurrently Huang *et al.* (2014) described batch cut-and-choose protocols following the high-level approach described above. The former protocol was later optimized and implemented in Lindell and Riva (2015). The protocols use the input-recovery technique so that each instance is secure as long as at least one correct circuit is evaluated.

6.4 Gate-level Cut-and-Choose: LEGO

In batch cut-and-choose, the amortized cost per bucket/instance decreases as the number of instances increases. This observation was the foundation of the *LEGO paradigm* for malicious-secure two-party computation, introduced by Nielsen and Orlandi (2009). The main idea is to do a batch cut-and-choose on *individual garbled gates* rather than on entire garbled circuits:

1. P_1 generates a large number of independently garbled NAND gates, and the parties perform a batch cut-and-choose on them. P_2 chooses some gates to check and randomly assigns the remaining gates into buckets.

2. The buckets of gates are assembled into a garbled circuit in a process called *soldering* (described in more detail below).

    - The gates within a single bucket are connected so that they collectively act like a fault-tolerant garbled NAND gate, which correctly

[1]The replication factor in this modified game measures only the number of evaluation circuits, whereas for a single instance we considered the total number (check and evaluation) of circuits. In practice, the number of checked circuits in batch cut-and-choose is quite small, and there is little difference between amortized number of *total* circuits vs. amortized number of *evaluation* circuits.

computes the NAND function as long as a majority of gates in the bucket are correct.

- The fault-tolerant garbled gates are connected to form the desired circuit. The connections between garbled gates transfer the garbled value on the output wire of one gate to the input wire of another.

3. P_2 evaluates the single conceptual garbled circuit, which is guaranteed to behave like a correct garbled circuit with overwhelming probability.

We now describe the soldering process in more detail, using the terminology of Frederiksen *et al.* (2013). The paradigm requires a *homomorphic commitment*, meaning that if P_1 commits to values A and B independently, it can later either decommit as usual, or can generate a decommitment that reveals *only* $A \oplus B$ to P_2.

P_1 prepares many individual garbled gates, using the FreeXOR technique. For each wire i, P_1 chooses a random “zero-label” k_i^0; the other label for that wire is $k_i^1 = k_i^0 \oplus \Delta$, where Δ is the FreeXOR offset value common to all gates. P_1 sends each garbled gate, and commits to the zero-label of each wire, as well as to Δ (once and for all for all gates). In this way, P_1 can decommit to k_i^0 or to $k_i^1 = k_i^0 \oplus \Delta$ using the homomorphic properties of the commitment scheme.

If a gate is chosen to be checked, then P_1 cannot open all wire labels corresponding to the gate. This would reveal the global Δ value and break the security of all gates. Instead, P_2 chooses a one of the four possible input combinations for the gate at random, and P_1 opens the corresponding input and output labels (one label per wire). Then, P_2 can check that the gate evaluates correctly on this combination. An incorrectly-garbled gate can be therefore caught only with probability $\frac{1}{4}$ (Zhu and Huang (2017) provides a way to increase this probability to $\frac{1}{2}$). This difference affects the cut-and-choose parameters (e.g., bucket size) by a constant factor.

Soldering corresponds to connecting various wires (attached to individual gates) together, so that the logical value on a wire can be moved to another wire. Say that wire u (with zero-label k_u^0) and wire v (with zero-label k_v^0) are to be connected. Then P_1 can decommit to the solder value $\sigma_{u,v} = k_u^0 \oplus k_v^0$. This value allows P_2 to transfer a garbled value from wire u to wire v during circuit evaluation. For example, if P_2 holds wire label $k_u^b = k_u^0 \oplus b \cdot \Delta$, representing unknown value b, then xor-ing this wire label with the solder value $\sigma_{u,v}$ results in the appropriate wire label on wire v:

$$k_u^b \oplus \sigma_{u,v} = (k_u^0 \oplus b \cdot \Delta) \oplus (k_u^0 \oplus k_v^0) = k_v^0 \oplus b \cdot \Delta = k_v^b$$

Gates within a bucket are assembled into a fault-tolerant NAND gate by choosing the first gate as an "anchor" and soldering wires of other gates to the matching wire of the anchor (i.e., solder the left input of each gate to the left input of the anchor). With ρ gates in a bucket, this gives ρ ways to evaluate the bucket starting with the garbled inputs of the anchor gate—transfer the garbled values to another gate, evaluate the gate, and transfer the garbled value back to the anchor. If all gates are correct, all ρ of the evaluation paths will result in an identical output wire label. If some gates are incorrect, the evaluator takes the majority output wire label.

The LEGO paradigm can take advantage of the better parameters for batch cut-and-choose, even in the single-execution setting. If the parties wish to securely evaluate a circuit of N gates, the LEGO approach involves a replication factor of $O(1) + O(\lambda/\log N)$, where λ is the security parameter. Of course, the soldering adds many extra costs that are not present in circuit-level cut-and-choose. However, for large circuits the LEGO approach gives a significant improvement over circuit-level cut-and-choose that has replication factor λ.

Variations on LEGO. The LEGO protocol paradigm has been improved in a sequence of works (Frederiksen *et al.*, 2013; Frederiksen *et al.*, 2015; Zhu and Huang, 2017; Kolesnikov *et al.*, 2017b; Zhu *et al.*, 2017). Some notable variations include:

- Avoiding majority-buckets in favor of buckets that are secure if even one garbled gate is correct (Frederiksen *et al.*, 2015).
- Performing cut-and-choose at the level of component subcircuits consisting of multiple gates (Kolesnikov *et al.*, 2017b).
- Performing cut-and-choose with a fixed-size pool of gates that is constantly replenished, rather than generating all of the necessary garbled gates up-front (Zhu *et al.*, 2017).

6.5 Zero-Knowledge Proofs

An alternative to the cut-and-choose approach is to convert a semi-honest protocol into a malicious-secure protocol by incorporating a proof that the protocol was executed correctly. Of course, the proof cannot reveal the secrets used in the protocol. Goldreich *et al.* (1987) shows how to use zero-knowledge (ZK) proofs to turn any semi-honest MPC protocol into one that is secure against malicious adversaries (Section 6.5.1).

Zero-knowledge proofs are a special case of malicious secure computation, and were introduced in Section 2.4. ZK proofs allow a prover to convince a verifier that it knows x such that $C(x) = 1$, without revealing any additional information about x, where C is a public circuit.

6.5.1 GMW Compiler

Goldreich, Micali, and Wigderson (GMW) showed a compiler for secure multi-party computation protocols that uses ZK proofs (Goldreich *et al.*, 1987). The compiler takes as input any protocol secure against semi-honest adversaries, and generates a new protocol for the same functionality that is secure against malicious adversaries.

Let π denote the semi-honest-secure protocol. The main idea of the GMW compiler is to run π and prove in zero-knowledge that every message is the result of running π *honestly*. The honest parties abort if any party fails to provide a valid ZK proof. Intuitively, the ZK proof ensures that a malicious party can either run π honestly, or cheat in π but cause the ZK proof to fail. If π is indeed executed honestly, then the semi-honest security of π ensures security. Whether or not a particular message is consistent with honest execution of π depends on the parties' private inputs. Hence, the ZK property of the proofs ensures that this property can be checked without leaking any information about these private inputs.

Construction. The main challenge in transforming a semi-honest protocol into an analogous malicious-secure protocol is to precisely define the circuit that characterizes the ZK proofs. Two important considerations are:

1. Each party must prove that each message of π is consistent with honest execution of π, *on a consistent input*. In other words, the ZK proof

should prevent parties from running π with different inputs in different rounds.

2. The "correct" next message of π is a function of not only the party's private input, but also their private random tape. π guarantees security only when each party's random tape is chosen uniformly. In particular, the protocol may be insecure if the party runs honestly but on some adversarially-chosen random tape.

The first consideration is addressed by having each party commit to its input upfront. Then all ZK proofs refer to this commitment: e.g., the following message is consistent with an honest execution of π, on the input that is contained inside the public commitment.

The second consideration is addressed by a technique called *coin-tossing into the well*. For concreteness, we focus on the ZK proofs generated by P_1. Initially P_1 produces a commitment to a random string r. Then P_2 sends a value r' in the clear. Now P_1 must run π with $r \oplus r'$ as the random tape. In this way, P_1 does not have unilateral control over its effective random tape $r \oplus r'$ — it is distributed uniformly even if P_1 is corrupt. P_1's ZK proofs can refer to the commitment to r (and the public value r') to guarantee that π is executed with $r \oplus r'$ as its random tape.

The full protocol description is given in Figure 6.2.

6.5.2 ZK from Garbled Circuits

Jawurek, Kerschbaum, and Orlandi (JKO) presented an elegant zero-knowledge protocol based on garbled circuits (Jawurek *et al.*, 2013). Since zero-knowledge is a special case of malicious secure computation, one can obviously base zero-knowledge on any cut-and-choose-based 2PC protocol. However, these protocols require many garbled circuits. The JKO protocol on the other hand achieves zero-knowledge using only one garbled circuit.

The main idea is to use a single garbled circuit for both evaluation and checking. In standard cut-and-choose, opening a circuit that is used for evaluation would reveal the private input of the garbled circuit generator. However, the verifier in a zero-knowledge protocol has no private input. Thus, the verifier can play the role of circuit garbler.

PARAMETERS: Semi-honest-secure two-party protocol $\pi = (\pi_1, \pi_2)$, where $\pi_b(x, r, T)$ denotes the next message for party P_b on input x, random tape r, and transcript so far T. A commitment scheme Com.

PROTOCOL π^*: (P_1 has input x_1 and P_2 has input x_2)

1. For $b \in \{1, 2\}$, P_b chooses random r_b and generates a commitment c_b to (x_b, r_b) with decommitment δ_b.

2. For $b \in \{1, 2\}$, P_b chooses and sends random r'_{3-b} (a share of the counterpart's random tape).

3. The parties alternate between $b = 1$ and $b = 2$ as follows, until the protocol terminates:

   (a) Let T be the transcript of π-messages so far (initially empty). P_b computes and sends the next π-message, $t = \pi_b(x_b, r_b \oplus r'_b, T)$. If instead π_b terminates, P_b terminates as well (with whatever output π_b specifies).

   (b) P_b acts as prover in a ZK proof with private inputs x_b, r_b, δ_b, and public circuit $C[\pi_b, c_b, r'_b, T, t]$ that is defined as:

   ```
   $C[\pi, c, r', T, t](x, r, \delta)$:
       return 1 iff $\delta$ is a valid opening of commitment
       $c$ to $(x, r)$ and $t = \pi(x, r \oplus r', T)$.
   ```

   The other party P_{3-b} aborts if verification of the ZK proof fails.

Figure 6.2: GMW compiler applied to a semi-honest-secure protocol π.

Suppose the prover P_1 wishes to prove $\exists w : \mathcal{F}(w) = 1$ where $\mathcal{F}$ is a public function. The JKO protocol proceeds in the following steps:

1. The verifier P_2 generates and sends a garbled circuit computing $\mathcal{F}$.

2. The prover picks up garbled inputs for w, using oblivious transfer.

3. The prover evaluates the circuit and obtains the output wire label (corresponding to output 1) and generates a commitment to this wire label.

4. The verifier opens the garbled circuit and the prover checks that it was generated correctly. If so, then the prover opens the commitment to the output wire label.

5. The verifier accepts if the prover successfully decommits to the 1 output wire label of the garbled circuit.

The protocol is secure against a cheating prover because at the time P_1 generates a commitment in step 3, the garbled circuit has not yet been opened. Hence, if P_1 does not know an input that makes $\mathcal{F}$ output 1, it is hard to predict the 1 output wire label at this step of the protocol. The protocol is secure against a cheating verifier because the prover only reveals the result of the garbled circuit after the circuit has been confirmed to be generated correctly.

Because the garbled circuits used for this protocol only need to provide authenticity and not privacy, their garbled tables can be implemented less expensively than for standard Yao's. Zahur *et al.* (2015) show that the half-gates method can be used to reduce the number of ciphertexts needed for a privacy-free garbled circuit to a single ciphertext for each AND gate, and no ciphertexts for XOR gates.

6.6 Authenticated Secret Sharing: BDOZ and SPDZ

Recall the approach for secret-sharing based MPC using Beaver triples (Section 3.4). This protocol paradigm is malicious-secure given suitable Beaver triples and any sharing mechanism such that:

1. Sharings are additively homomorphic,

2. Sharings hide the underlying value against a (malicious) adversary, and

3. Sharings can be opened reliably, even in the presence of a malicious adversary.

In this section we describe two sharing mechanisms with these properties: BDOZ (Section 6.6.1) and SPDZ (Section 6.6.2).

6.6.1 BDOZ Authentication

The Bendlin-Damgård-Orlandi-Zakarias (BDOZ or BeDOZa) technique (Bendlin *et al.*, 2011) incorporates information-theoretic MACs into the secret shares. Let $\mathbb{F}$ be the underlying field, with $|\mathbb{F}| \geq 2^\kappa$ where κ is the security parameter. Interpreting $K, \Delta \in \mathbb{F}$ as a key, define $\mathsf{MAC}_{K,\Delta}(x) = K + \Delta \cdot x$.

This construction is an information-theoretic one-time MAC. An adversary who sees $\mathsf{MAC}_{K,\Delta}(x)$ for a chosen x cannot produce another valid MAC, $\mathsf{MAC}_{K,\Delta}(x')$, for $x \neq x'$. Indeed, if an adversary could compute such a MAC, then it could compute Δ:

$$\begin{aligned}
&(x - x')^{-1}\Big(\mathsf{MAC}_{K,\Delta}(x) - \mathsf{MAC}_{K,\Delta}(x')\Big) \\
&= (x - x')^{-1}\Big(K + \Delta x - K - \Delta x'\Big) \\
&= (x - x')^{-1}(\Delta(x - x')) = \Delta
\end{aligned}$$

But seeing only $\mathsf{MAC}_{K,\Delta}(x)$ perfectly hides Δ from the adversary. Hence, the probability of computing a MAC forgery is bounded by $1/|\mathbb{F}| \leq 1/2^\kappa$, the probability of guessing a randomly chosen field element Δ.

In fact, the security of this MAC holds even when an honest party has many MAC keys that all share the same Δ value (but with independently random K values). We refer to Δ as the global MAC key and K as the local MAC key.

The idea of BDOZ is to authenticate each party's shares with these information-theoretic MACs. We start with the two-party case. Each party P_i generates a global MAC key Δ_i. Then $[x]$ denotes the secret-sharing mechanism where P_1 holds x_1, m_1 and K_1 and P_2 holds x_2, m_2 and K_2 such that:

1. $x_1 + x_2 = x$ (additive sharing of x),
2. $m_1 = K_2 + \Delta_2 x_1 = \mathsf{MAC}_{K_2,\Delta_2}(x_1)$ (P_1 holds a MAC of its *share* x_1 under P_2's MAC key), and
3. $m_2 = K_1 + \Delta_1 x_2 = \mathsf{MAC}_{K_1,\Delta_1}(x_2)$ (P_2 holds a MAC of its *share* x_2 under P_1's MAC key).

Next, we argue that this sharing mechanism satisfies the properties required by the Beaver-triple paradigm (Section 3.4):

sharing	P_1 has	P_2 has
$[x]$	x_1	x_2
	K_1	K_2
	$\mathsf{MAC}_{K_2,\Delta_2}(x_1)$	$\mathsf{MAC}_{K_1,\Delta_1}(x_2)$
$[x']$	x'_1	x'_2
	K'_1	K'_2
	$\mathsf{MAC}_{K'_2,\Delta_2}(x'_1)$	$\mathsf{MAC}_{K'_1,\Delta_1}(x'_2)$
$[x + x']$	$x_1 + x'_1$	$x_2 + x'_2$
	$K_1 + K'_1$	$K_2 + K'_2$
	$\mathsf{MAC}_{K_2+K'_2,\Delta_2}(x_1 + x'_1)$	$\mathsf{MAC}_{K_1+K'_1,\Delta_1}(x_2 + x'_2)$

Figure 6.3: BDOZ authenticated sharing

- Privacy: the individual parties learn nothing about x since they only hold one additive share, x_p, and m_p reveals nothing about x without knowing the other party's keys (which are never revealed).

- Secure opening: To open, each party announces its (x_p, m_p), allowing both parties to learn $x = x_1 + x_2$. Then, P_1 can use its MAC key to check whether $m_2 = \mathsf{MAC}_{K_1,\Delta_1}(x_2)$ and abort if this is not the case. P_2 performs an analogous check on m_1. Note that opening this sharing to any different value corresponds exactly to the problem of breaking the underlying one-time MAC.

- Homomorphism: The main idea is that when all MACs in the system use the same Δ value, the MACs become homomorphic in the necessary way. That is,

  $$\mathsf{MAC}_{K,\Delta}(x) + \mathsf{MAC}_{K',\Delta}(x') = \mathsf{MAC}_{K+K',\Delta}(x + x')$$

  Here we focus on adding shared values $[x] + [x']$; the other required forms of homomorphism work in a similar way. The sharings of $[x]$ and $[x']$ and the resulting BDOZ sharing of $[x + x']$ is shown in Figure 6.3.

The BDOZ approach generalizes to n parties in a straightforward (but expensive) way. All parties have global MAC keys. In a single sharing $[x]$, the parties have additive shares of x and each party's share is authenticated under *every other party's* MAC key.

Generating triples. The BDOZ sharing method satisfies the security and homomorphism properties required for use in the abstract Beaver-triples approach. It remains to be seen how to generate Beaver triples in this format.

Note that BDOZ shares work naturally even when the payloads (i.e., x in $[x]$) are restricted to a subfield of $\mathbb{F}$. The sharings $[x]$ are then homomorphic with respect to that subfield. A particularly useful case is to use BDOZ for sharings of single bits, interpreting $\{0, 1\}$ as a subfield of $\mathbb{F} = GF(2^\kappa)$. Note that $\mathbb{F}$ must be exponentially large for security (authenticity) to hold.

The state of the art method for generating BDOZ shares of bits is the scheme used by Tiny-OT (Nielsen *et al.*, 2012). It uses a variant of traditional OT extension (Section 3.7.2) to generate BDOZ-authenticated bits $[x]$. It then uses a sequence of protocols to securely multiply these authenticated bits needed to generate the required sharings for Beaver triples.

6.6.2 SPDZ Authentication

In BDOZ sharing, each party's local part of $[x]$ contains a MAC for every other party. In other words, the storage requirement of the protocol scales linearly with the number of parties. A different approach introduced by Damgård, Pastro, Smart, and Zakarias (SPDZ, often pronounced "speeds") (Damgård *et al.*, 2012b) results in constant-sized shares for each party.

As before, we start with the two-party setting. The main idea is to have a global MAC key Δ that is not known to either party. Instead, the parties hold Δ_1 and Δ_2 which can be thought of as shares of a global $\Delta = \Delta_1 + \Delta_2$. In a SPDZ sharing $[x]$, P_1 holds (x_1, t_1) and P_2 holds (x_2, t_2), where $x_1 + x_2 = x$ and $t_1 + t_2 = \Delta \cdot x$. Thus, the parties hold additive shares of x and of $\Delta \cdot x$. One can think of $\Delta \cdot x$ as a kind of "0-time information-theoretic MAC" of x.

This scheme clearly provides privacy for x. Next, we show that it also provides the other two properties required for Beaver triples:

- Secure opening: We cannot have the parties simply announce their shares, since that would reveal Δ. It is important that Δ remain secret throughout the entire protocol. To open $[x]$ without revealing Δ, the protocol proceeds in 3 phases:

1. The players announce only x_1 and x_2. This determines the (unauthenticated) candidate value for x.
2. Note that if this candidate value for x is indeed correct, then

$$\begin{aligned}(\Delta_1 x - t_1) + (\Delta_2 x - t_2) &= (\Delta_1 + \Delta_2)x - (t_1 + t_2)\\ &= \Delta x - (\Delta x)\\ &= 0\end{aligned}$$

   Furthermore, P_1 can locally compute the first term $(\Delta_1 x - t_1)$ and P_2 can compute the other term. In this step of the opening, P_1 *commits* to the value $\Delta_1 x - t_1$ and P_2 commits to $\Delta_2 x - t_2$.
3. The parties open these commitments and abort if their sum is not 0. Note that if the parties had simply announced these values one at a time, then the last party could cheat by choosing the value that causes the sum to be zero. By using a commitment, the protocol forces the parties to know these values in advance.

To understand the security of this opening procedure, note that when P_1 commits to some value c, it expects P_2 to also commit to $-c$. In other words, the openings of these commitments are easily simulated, which implies that they leak nothing about Δ.

It is possible to show that if a malicious party is able to successfully open $[x]$ to a different x', then that party is able to guess Δ. Since the adversary has no information about Δ, this event is negligibly likely.

- Homomorphism: In a SPDZ sharing $[x]$, the parties' shares consist of additive shares of x and additive shares of $\Delta \cdot x$. Since each of these are individually homomorphic, the SPDZ sharing naturally supports addition and multiplication-by-constant.

  To support addition-by-constant, the parties must use their additive shares of Δ as well. Conceptually, they locally update $x \mapsto x + c$ and locally update $\Delta x \mapsto \Delta x + \Delta c$. This is illustrated in Figure 6.4.

Generating SPDZ shares. Since SPDZ shares satisfy the properties needed for abstract Beaver-triple-based secure computation, the only question remains

sharing	P_1 has	P_2 has	sum of P_1 and P_2 shares
$[x]$	x_1	x_2	x
	t_1	t_2	Δx
$[x+c]$	$x_1 + c$	x_2	$x + c$
	$t_1 + \Delta_1 c$	$t_2 + \Delta_2 c$	$\Delta(x+c)$

Figure 6.4: SPDZ authenticated secret sharing.

how to generate Beaver triples in the SPDZ format. The paper that initially introduced SPDZ (Damgård *et al.*, 2012b) proposed a method involving somewhat homomorphic encryption. Followup work suggests alternative techniques based on efficient OT extension (Keller *et al.*, 2016).

6.7 Authenticated Garbling

Wang *et al.* (2017b) introduced an *authenticated garbling* technique for multiparty secure computation that combines aspects of information-theoretic protocols (e.g., authenticated shares and Beaver triples) and computational protocols (e.g., garbled circuits and BMR circuit generation). For simplicity, we describe their protocol in the two-party setting but many (not all) of the techniques generalize readily to the multi-party setting (Wang *et al.*, 2017c).

A different perspective on authenticated shares of bits. One starting point is the BDOZ method for authenticated secret-sharing of *bits*. Recall that a 2-party BDOZ sharing $[x]$ corresponds to the following information:

sharing	P_1 has	P_2 has
$[x]$	x_1	x_2
	K_1	K_2
	$T_1 = K_2 \oplus x_1\Delta_2$	$T_2 = K_1 \oplus x_2\Delta_1$

Since we consider x, x_1, x_2 to be bits, the underlying field is $\mathbb{F} = \mathrm{GF}(2^\kappa)$ and we write the field addition operation as $\oplus$. An interesting observation is that:

$$\underbrace{(K_1 \oplus x_1\Delta_1)}_{\text{known to } \mathsf{P}_1} \oplus \underbrace{(K_1 \oplus x_2\Delta_1)}_{\text{known to } \mathsf{P}_2} = (x_1 \oplus x_2)\Delta_1 = x\Delta_1$$

Hence, a side-effect of a BDOZ sharing $[x]$ is that parties hold additive shares of $x\Delta_1$, where Δ_1 is P_1's global MAC key.

Distributed garbling. Consider a garbled circuit in which the garbler P_1 chooses wire labels k_i^0, k_i^1 for each wire i. Departing from the notation from Section 3.1.2, we will let the superscript b in k_i^b denote the public "point-and-permute" pointer bit of a wire label (that the evaluator learns), rather than its semantic value true/false. We let p_i denote the corresponding pointer bit, so that $k_i^{p_i}$ is the label representing false.

We focus on a single AND gate with input wires a, b and output wire c. Translating the standard garbled circuit construction into this perspective (i.e., organized according to the pointer bits), we obtain the following garbled table:

$$\begin{aligned}
e_{0,0} &= H(k_a^0 \| k_b^0) \oplus k_c^{p_c \oplus p_a \cdot p_b} \\
e_{0,1} &= H(k_a^0 \| k_b^1) \oplus k_c^{p_c \oplus p_a \cdot \overline{p_b}} \\
e_{1,0} &= H(k_a^1 \| k_b^0) \oplus k_c^{p_c \oplus \overline{p_a} \cdot p_b} \\
e_{1,1} &= H(k_a^1 \| k_b^1) \oplus k_c^{p_c \oplus \overline{p_a} \cdot \overline{p_b}}
\end{aligned}$$

Using FreeXOR, $k_i^1 = k_i^0 \oplus \Delta$ for some global value Δ. In that case, we can rewrite the garbled table as:

$$\begin{aligned}
e_{0,0} &= H(k_a^0 \| k_b^0) \oplus k_c^0 \oplus (p_c \oplus p_a \cdot p_b)\Delta \\
e_{0,1} &= H(k_a^0 \| k_b^1) \oplus k_c^0 \oplus (p_c \oplus p_a \cdot \overline{p_b})\Delta \\
e_{1,0} &= H(k_a^1 \| k_b^0) \oplus k_c^0 \oplus (p_c \oplus \overline{p_a} \cdot p_b)\Delta \\
e_{1,1} &= H(k_a^1 \| k_b^1) \oplus k_c^0 \oplus (p_c \oplus \overline{p_a} \cdot \overline{p_b})\Delta
\end{aligned}$$

One of the main ideas in authenticated garbling is for the parties to construct such garbled gates in a somewhat distributed fashion, in such a way that neither party knows the p_i permute bits.

Instead, suppose the parties only have *BDOZ sharings* of the form $[p_a], [p_b], [p_a \cdot p_b], [p_c]$, where neither party knows these p_i values in the clear. Suppose further that P_1 chooses the garbled circuit's wire labels so that $\Delta = \Delta_1$ (i.e., its global MAC key from BDOZ). The parties therefore have additive shares of $p_a\Delta$, $p_b\Delta$, and so on. They can use the homomorphic properties of additive secret sharing to locally obtain shares of $(p_c \oplus p_a \cdot p_b)\Delta$, $(p_c \oplus p_a \cdot \overline{p_b})\Delta$, and so on.

Focusing on the first ciphertext in the garbled table, we can see:

$$e_{0,0} = \underbrace{H(k_a^0 \| k_b^0) \oplus k_c^0}_{\text{known to } \mathsf{P}_1} \oplus \underbrace{(p_c \oplus p_a \cdot p_b)\Delta}_{\text{parties have additive shares}}$$

Hence, using only local computation (P_1 simply adds the appropriate value to its share), parties can obtain additive shares of $e_{0,0}$ and all other rows in the garbled table.

In summary, the distributed garbling procedure works by generating BDOZ-authenticated shares of random permute bits $[p_i]$ for every wire in the circuit, along with Beaver triples $[p_a]$, $[p_b]$, $[p_a \cdot p_b]$ for every AND gate in the circuit. Then, using only local computation, the parties can obtain additive shares of a garbled circuit that uses the p_i values as its permute bits. P_1 sends its shares of the garbled circuit to P_2, who can open it and evaluate.

Authenticating the garbling. As in Yao's protocol, the garbler P_1 can cheat and generate an incorrect garbled circuit — in this case, by sending incorrect additive shares. For example, P_1 can replace the "correct" $e_{0,0}$ value in some gate by an incorrect value, while leaving the other three values intact. In this situation, P_2 obtains an incorrect wire label whenever the logical input to this gate is (p_a, p_b). Even assuming P_2 can detect this condition and abort, this leads to a *selective abort* attack for a standard garbled circuit. By observing whether P_2 aborts, P_1 learns whether the input to this gate was (p_a, p_b).

However, this is not actually a problem for distributed garbling. While it is still true that P_2 aborts if and only if the input to this gate was (p_a, p_b), P_1 has no information about p_a, p_b — hiding these permute bits from P_1 causes P_2's abort probability to be input-independent!

Constructing a garbled circuit with secret permute bits addresses the problem of privacy against a corrupt P_1. However, P_1 may still break the correctness of the computation. For instance, P_1 may act in a way that flips one of the $p_c \oplus p_a \cdot p_b$ bits. To address this, Wang *et al.* (2017b) relies on the fact that the parties have BDOZ sharings of the p_i indicator bits. These sharings determine the "correct" pointer bits that P_2 should see. For example, if the input wires to an AND gate have pointer bits $(0, 0)$ then the correct pointer bit for the output wire is $p_c \oplus p_a \cdot p_b$. To ensure correctness of the computation, it suffices to ensure that P_2 always receives the correct pointer bits. As discussed previously, the parties can obtain authenticated BDOZ sharings of $p_c \oplus p_a \cdot p_b$. We therefore augment the garbled circuit so that each ciphertext contains not only the output wire label, but also P_1's authenticated BDOZ share of the "correct" pointer bit. The BDOZ authentication ensures that P_1 cannot cause P_2 to view pointer bits that are inconsistent with the secret p_i values, without aborting the protocol. Now as P_2 evaluates the garbled circuit, it checks for each gate that the visible pointer bit is authenticated by the associated MAC.

This protocol provided dramatic cost improvements. Wang *et al.* (2017b) reports on experiments using authenticated garbling to execute malicious secure protocols over both local and wide area networks. In a LAN setting, it can execute over 800,000 AND gates per second and perform a single private AES encryption in 16.6 ms (of which 0.93 ms is online cost) on a 10Gbps LAN and 1.4 s on a WAN (77 ms is online cost). In a batched setting where 1024 AES encryptions are done, the amortized total cost per private encryption drops to 6.66 ms (113 ms in a WAN). As a measure of the remarkable improvement in MPC execution, the fastest AES execution as a semi-honest LAN protocol in 2010 was 3300 ms total time (Henecka *et al.*, 2010), so in a matter of eight years the time required to execute a malicious secure protocol dropped to approximately $\frac{1}{200}$ that of the best semi-honest protocol!

6.8 Further Reading

Cut-and-choose existed as a folklore technique in the cryptographic literature. Different cut-and-choose mechanisms were proposed by Mohassel and Franklin (2006) and Kiraz and Schoenmakers (2006), but without security proofs. The first cut-and-choose protocol for 2PC with a complete security proof was due to Lindell and Pinkas (2007).

We presented one technique from Lindell and Pinkas (2007) for dealing with the selective abort attacks. The subtle nature of selective abort attacks was first observed by Kiraz and Schoenmakers (2006), and fixed using a different technique — in that work, by modifying the oblivious transfers into a variant called committed OT.

We presented one technique for the problem of input consistency in cut-and-choose (from shelat and Shen (2011)). Many other input-consistency mechanisms have been proposed including Lindell and Pinkas (2007), Lindell and Pinkas (2011), Mohassel and Riva (2013), and shelat and Shen (2013).

We described the BDOZ and SPDZ approaches to authenticated secret-sharing. Other efficient approaches include Damgård and Zakarias (2013) and Damgård *et al.* (2017). Various approaches for efficiently generating the authenticated shares needed for the SPDZ approach are discussed by Keller *et al.* (2016) and Keller *et al.* (2018).

The GMW paradigm transforms a semi-honest-secure protocol into a malicious-secure one. However, it generally does not result in a protocol with practical efficiency. This is due to the fact that it treats the semi-honest-secure protocol in a *non-black-box* way — the parties must prove (in zero knowledge) statements about the next-message function of the semi-honest-secure protocol, which in the general case requires expressing that function as a circuit. Jarecki and Shmatikov (2007) propose a malicious variant of Yao's protocol that is similar in spirit to the GMW paradigm, in the sense that the garbling party proves correctness of each garbled gate (although at a cost of public-key operations for each gate).

A black-box approach for transforming a semi-honest-secure protocol into a malicious-secure one (Ishai *et al.*, 2007; Ishai *et al.*, 2008) is known as "MPC in the head." The idea is for the actual parties to imagine a protocol interaction among "virtual" parties. Instead of running an MPC protocol for the desired functionality, the actual parties run an MPC that realizes the behavior of the virtual parties. However, the MPC that is used to simulate the virtual parties can satisfy a weaker security notion (than what the overall MPC-in-the-head approach achieves), and the protocol being run among the virtual parties needs to be only semi-honest secure. For the special case of zero-knowledge proofs, the MPC-in-the-head approach results in protocols that are among the most efficient known (Giacomelli *et al.*, 2016; Chase *et al.*, 2017; Ames *et al.*, 2017; Katz *et al.*, 2018).