Secure Three-Party Computation

CS 507, Topics in Cryptography: Secure Computation

David Heath

Fall 2026

Last time, we saw how to solve the problem of computing secure AND between two parties A and B by using a third party D as a dealer. Recall that there were two main ideas behind that protocol:

- **XOR secret sharing.** We defined the notion of a secret-shared bit $[\![x]\!]$. Recall that this notation means that our two parties A and B hold respective shares $x_0, x_1 \in \{0,1\}$ such that $x_0 \oplus x_1 = x$.
- **Correlated randomness.** We asked the dealer D to sample two uniform bits $\alpha, \beta \leftarrow \{0,1\}$, to compute $\alpha \cdot \beta$, and then to deal uniform shares $[\![\alpha]\!], [\![\beta]\!], [\![\alpha \cdot \beta]\!]$ to our two parties.

From here, the main idea was that we could use a particular mathematical identity to transform the AND operation we care about—$x \cdot y$—into one that involves seemingly random values $x \oplus \alpha$ and $y \oplus \beta$. The specific identity we looked at is as follows:

$$(x \oplus \alpha) \cdot [\![y]\!] \oplus (y \oplus \beta) \cdot [\![\alpha]\!] \oplus [\![\alpha \cdot \beta]\!] = [\![x \cdot y]\!].$$

To compute this, parties A, B needed to reveal to themselves $x \oplus \alpha$ and $y \oplus \beta$, but this is safe, since α, β are uniform, and hence $x \oplus \alpha$ and $y \oplus \beta$ look uniform to A (resp. B). Formally, we were able to simulate what A, B see in this protocol. To complete the AND operation, the parties simply exchanged their shares of $[\![x \cdot y]\!]$ and reconstructed $x \cdot y$. Thus, given a secret-shared multiplication triple, our two parties can compute the AND of two bits of their choice.

Today, we will build on the main ideas above to construct a protocol that allows our two parties (with the help of a dealer) to securely compute any deterministic, computable function f of their joint inputs.

Protocol Sketch

The main idea of our protocol Π is as follows:

1. Convert the function f into a Boolean circuit C.
2. Secret share the party inputs x and y.
3. Step through the Boolean circuit one gate at a time. At each gate, use secret sharings on input wires to compute a secret sharing on the output wire. This can be achieved with the help of correlated randomness dealt by D.
4. Once each gate is computed, exchange shares of the output wires and reconstruct the output.

Roughly speaking, Π is secure because all internal revealed values of the computation appear to be uniform. This allows us to simulate the view of A, B, and D.

Boolean Circuits

Let's recall the definition of a Boolean circuit. It will be useful for us to consider a particular kind of circuit consisting of AND, XOR, and NOT gates. For those that are familiar with Boolean circuits, the following definition is unsurprising, and it is included mostly for completeness:

Definition 1 (Boolean Circuit)**.** *A* Boolean circuit *C is a directed acyclic graph. Vertices in C are called* gates *and edges are called* wires*. Each gate computes a single output value, which is carried by every wire leaving that gate; thus a gate may have any number of output wires (its* fan-out*), all carrying the same value. We use $|C|$ to denote the number of wires in C. We consider circuits with five types of gates:*

- *An* AND *gate has two input wires. We denote an AND gate with input wires a, b and output value c by writing $c := a \cdot b$.*
- *An* XOR *gate has two input wires. We write $c := a \oplus b$.*
- *A* NOT *gate has one input wire. We write $b := \neg a$, or sometimes $b := \bar{a}$.*
- *An* INPUT *gate has no input wires. We write $a :=$ `INPUT`.*
- *An* OUTPUT *gate has one input wire and no output wires. We write* `OUTPUT`(a)*.*

If C has n INPUT gates and m OUTPUT gates, we write $C : \{0,1\}^n \to \{0,1\}^m$. Indeed, C defines a Boolean function $f : \{0,1\}^n \to \{0,1\}^m$ in a natural manner. We (1) assign each i-th input bit to the circuit's i-th INPUT wire, (2) for each XOR/AND/NOT gate assign the gate's output wire by computing the appropriate function of its input wires in a topological order, and (3) read each i-th output bit from the i-th OUTPUT gate.

When clear from context that a wire is an input/output, we will omit explicit INPUT/OUTPUT notation. Similarly, we will for brevity sometimes simply write Boolean expressions, e.g. $(x \oplus y) \cdot z$, without explicitly naming the intermediate wires.

Importantly, Boolean circuits are complete:

Fact 1 (Completeness)**.** *For every Boolean function $f : \{0,1\}^n \to \{0,1\}^m$, there exists a Boolean circuit C that computes f.*

More importantly for our purposes, there is a well known sense in which Boolean circuits are efficient. Namely, Boolean circuit size is polynomially-related to the runtime of Turing machines:

Fact 2 (Polynomial Efficiency)**.** *Let M be a Turing machine that, within T steps, halts on every input x of length n with $f(x)$ written on its output tape. There exists a Boolean circuit of size* $\mathrm{poly}(T)$ *that on input x computes $f(x)$.*

Hence, any poly-time computable function can be computed by a poly-size circuit. This means that if we can construct a protocol that securely computes a Boolean circuit C in polynomial time (with respect to $|C|$), then we can securely compute any poly-time computable function in polynomial time.

An Example Circuit

We will be playing with circuit design a bit throughout this course, so it's useful to consider some example Boolean circuits to see how they can be constructed. Let's consider an example where we wish to compare two-bit numbers $x = x_1x_0$ and $y = y_1y_0$. Here, x_1 (resp. y_1) is the most significant bit of x (resp. y). In particular, we wish to compute $x < y$.

First, let's recall the definition of XOR and AND for two bits a and b:

a	b	$a \oplus b$	$a \cdot b$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

One basic observation from this table is that $a \oplus b$ naturally encodes the bitwise inequality operation:

$$a \neq b \iff a \oplus b.$$

Thus, it is also easy to encode the equality operation:

$$a = b \iff \neg(a \oplus b).$$

It's also easy to observe that the following checks if bit a is less than bit b:

$$a < b \iff \neg a \cdot b.$$

From these operations, we can construct a two-bit comparator on two-bit strings x and y:

$$x < y \iff (x_1 < y_1) \oplus (x_1 = y_1) \cdot (x_0 < y_0).$$

Exercise 1. *Reason through the logic of the above comparator. Show it extends to a comparator for pairs of strings of arbitrary length n.*

A Perfectly-Secure 3PC Protocol

Now, we are ready to construct our three-party protocol. Roughly, parties will agree on some Boolean circuit C that they wish to compute. Then, the parties will execute that circuit C where instead of each wire in C holding a logical value, the wire will hold a secret share of the logical value. That is, A and B will evaluate C jointly. Each party will see only a secret share of the value on each wire, and hence they will learn nothing. To evaluate each gate, they will potentially need to communicate with one another and consume correlated randomness dealt by D.

Protocol 1. *We specify a protocol Π that runs between three parties A, B, and D.*

- ***Parameters.*** *Parties agree on a computable function encoded as a Boolean circuit $C : \{0,1\}^n \to \{0,1\}^m$. Parties agree on n_A (resp. n_B), the size of A's (resp. B's) input, such that $n = n_A + n_B$. We say that the first n_A INPUT gates of C belong to A, and the other n_B INPUT gates belong to B.*
- ***Input.*** *A inputs a string $x \in \{0,1\}^{n_A}$; B inputs a string $y \in \{0,1\}^{n_B}$. D inputs nothing.*
- ***Output.*** *A and B output $C(x, y)$. D outputs nothing.*
- ***Procedure.*** *Parties proceed gate by gate through C in a topological order, propagating the following key invariant: For each wire in C holding logical value a, parties A and B hold XOR secret shares $[\![a]\!]$. In particular, the parties handle each gate as follows:*
  - *If the next gate is the i-th INPUT gate belonging to A, then A fetches its i-th input bit x_i, constructs a uniform secret sharing $[\![x_i]\!]$, and sends one share to B. Parties act symmetrically for B's INPUT gates.*
  - *If the next gate is an XOR of form $c := a \oplus b$, then by the topological ordering the parties must already hold secret-shared gate inputs $[\![a]\!]$ and $[\![b]\!]$. They use the secret sharing scheme's XOR homomorphism to compute $[\![c]\!] = [\![a \oplus b]\!]$ and place this on the output wire.*
  - *If the next gate is a NOT of form $b := \neg a$, then the parties already hold $[\![a]\!]$. Let $(a_0, a_1) = [\![a]\!]$. Parties locally compute a share $[\![b]\!] = [\![\neg a]\!] = (a_0 \oplus 1, a_1)$. Some basic Boolean reasoning shows this computation indeed yields a secret share of $\neg a$. Parties place this sharing on the gate output wire.*
  - *If the next gate is an AND of form $c := a \cdot b$, then the parties already hold $[\![a]\!], [\![b]\!]$, but they will need to communicate and consume correlated randomness to execute this gate. D samples a uniformly random multiplication triple $[\![\alpha]\!], [\![\beta]\!], [\![\alpha \cdot \beta]\!]$ and deals these shares to the two parties. Note that this multiplication triple is gate-specific; i.e., if C has k AND gates, the parties will need k independent triples. Now, A and B locally compute $[\![a \oplus \alpha]\!]$ and $[\![b \oplus \beta]\!]$, then exchange shares to reconstruct $a \oplus \alpha$ and $b \oplus \beta$. They then locally compute*

$$(a \oplus \alpha) \cdot [\![b]\!] \oplus (b \oplus \beta) \cdot [\![\alpha]\!] \oplus [\![\alpha \cdot \beta]\!] = [\![a \cdot b]\!] = [\![c]\!].$$

  They place this sharing on the gate output wire.

– *If the next gate is of form* $\texttt{OUTPUT}(a)$*, then the parties hold* $[\![a]\!]$*. They exchange shares to reconstruct and learn* a*.*

Theorem 1. *Protocol 1 is a three-party protocol that correctly computes any circuit* C *between parties* A *and* B*. The protocol is perfectly secure against a semi-honest adversary that corrupts any single party.*

Recall that to prove the theorem, one must construct a simulator for each party that produces a view, then argue that the produced view is identically distributed to the party's real-world view. The structure of such simulators is almost identical to what we saw last time, where the parties securely executed a single AND gate. Because the simulators are so similar to what we have already seen, we will not write them out here, but it is an excellent exercise to do so yourself:

Exercise 2. *Write out simulators to prove the theorem.*

Just like last time, these simulators mostly just sample uniform randomness; also like last time, the main challenge is that the simulator for A (resp. B) must properly simulate messages received for OUTPUT gates, which are not uniform in the real-world protocol. The hint here is that the simulator for A (resp. B) will also step through the circuit gate-by-gate, maintaining secret shares that look like the ones that A (resp. B) sees in the real protocol. By doing so, the simulator can anticipate the message it should receive corresponding to a particular OUTPUT gate.

The Good, the Bad, and the Ugly

Let's think about some properties of Protocol 1. Recall that there are three main efficiency measures for a protocol:

- **Computation Complexity.** It's easy to see that in our protocol, the parties perform only $O(|C|)$ work, since for each gate each party does a constant number of bitwise operations and exchanges at most a constant number of bits. This is excellent, far better than many of the protocols we will see in the future.
- **Communication Complexity.** Similarly, the parties exchange only $O(|C|)$ bits. In fact, we can refine this somewhat. Let $\#\texttt{AND}(C)$ count the number of AND gates in C. Our actual communication complexity is $O(n + m + \#\texttt{AND}(C))$ bits. This is because XOR/NOT gates are free: they do not require communication. Concretely, each AND gate costs four bits between A and B (two shares each direction) plus six bits from D (three shares to each party). This communication complexity is very good, especially in terms of the hidden constants inside the O notation.
- **Round Complexity.** So far, our round complexity is abysmal: Since parties proceed gate by gate, they use $O(n + m + \#\texttt{AND}(C))$ rounds of communication.

The round complexity $O(n + m + \#\texttt{AND}(C))$ can be straightforwardly significantly improved. In particular, notice that there is no reason to literally proceed one gate at a time through C. Instead, what the parties should do is proceed through the circuit in layers, handling in parallel all AND gates that are ready. A simple rearrangement of Protocol 1 therefore allows us to achieve round complexity scaling only with the so-called multiplicative depth of C.

Definition 2 (Multiplicative Depth)**.** *Let* C *be a Boolean circuit and let* P *be a path through* C *from some INPUT gate to some OUTPUT gate. The multiplicative length of* P *is the number of AND gates along* P*. The multiplicative depth of* C*—written* $\texttt{mul-depth}(C)$*—is the maximum over multiplicative lengths of all paths through* C*.*

With this rearrangement, the protocol runs in $\texttt{mul-depth}(C) + O(1)$ rounds. Still, this round complexity leaves much to be desired, but improving further will require new techniques.

There are also qualitative properties of our protocol that we should consider:

- It requires a dealer. The assumption that D is available to provide correlated randomness and will not collude with A or B is quite strong. We would like to remove this assumption.

- It works only for exactly three parties. It might be desirable to have a protocol that admits more parties, even when many of them collude.
- It only works against a semi-honest adversary. The protocol described is fundamentally broken against an adversary who actively attempts to break security.

  Exercise 3. *Show how a malicious adversary can attack this protocol.*

- It requires the computation to be expressed as a Boolean circuit C. While we argued that all poly-time computations are expressible as poly-size circuits, we have not said anything about the polynomial cost factors incurred by the circuit representation. Indeed, Boolean circuits tend to require very large representations for certain natural problems. The canonical example where circuits become expensive is when we have a program that requires access to random access arrays. Unfortunately, compiling array accesses to Boolean gates is expensive, in the sense that huge numbers of gates are required.

Going forward, we will investigate ways to improve all of these qualitative problems.

Next Time

We will investigate a way to remove the dealer D. We will specifically introduce a cryptographic primitive called oblivious transfer. Based on this building block protocol, we will show a way that A and B can conjure multiplication triples without a dealer.