Authenticated Triples

CS 507, Topics in Cryptography: Secure Computation

David Heath

Fall 2025

Last time were working on the problem of instantiating our preprocessing functionality for malicious MPC. Recall that our 2PC functionality performed three tasks:

- **Initialization.** Upon initialization, the functionality samples random $\Delta_A, \Delta_B \in_\$ \{0,1\}^\lambda$ and then sends these to A (resp. B).
- **Random bits.** Upon receiving instructions from both parties, the functionality samples a bit $\alpha \in_\$ \{0,1\}$, samples an authenticated sharing $\{\alpha\}$, and deals shares to the parties.
- **Random triples.** Upon receiving instructions from both parties, the functionality samples two bits $\alpha, \beta \in_\$ \{0,1\}$, samples authenticated sharings $\{\alpha, \beta, \alpha \cdot \beta\}$, and deals shares to the parties.

(Formally, above the adversary gets to choose its randomness on behalf of the corrupted party.) Recall that that in the two-party setting, we defined notation $\{x\}$ to mean a secret sharing of the following form:

$$\{x\} = [x \cdot (\Delta_A, \Delta_B, 1)]$$

Last time, we instantiated a maliciously-secure variant of the IKNP OT extension protocol, which achieved the following correlated OT functionality:

- On input $r \in \{0,1\}^n$ from a receiver and $\Delta \in \{0,1\}^\lambda$ from a sender, output $[r \otimes \Delta]$.

(Again, formally, the adversary gets to choose its randomness in the output secret shares on behalf of the corrupted party.)

We also saw how by using two invocations of correlated OT, we can construct authenticated random bits. Namely, we can invoke correlated OT once with A as the sender with key Δ_A, and once with B as the sender with key Δ_B. By doing so, we obtain numerous authenticated bits $\{\alpha_A\}, \{\alpha_B\}$ where α_A is known in the clear to A (resp. B). By summing two such bits, we can obtain a uniform authenticated bit known to neither party:

$$\{\alpha_A\} \oplus \{\alpha_B\} = \{\alpha\} \qquad \text{where } \alpha = \alpha_A \oplus \alpha_B$$

In the following, it will be useful to keep around the two "halves" of $\{\alpha\}$—namely, $\{\alpha_A\}, \{\alpha_B\}$.

Our goal today is to build on top of these authenticated random bits and ultimately obtain random authenticated AND triples, completing our functionality. We will follow present a technique similar to that of [KRRW18].

Ultimately, our goal is to construct random triples $\{\alpha\}, \{\beta\}, \{\alpha\beta\}$. Our presentation will proceed in two steps:

- First, we will build *leaky* AND triples. These are authenticated triples, but we will allow the adversary to learn one input of the triple with probability $1/2$.
- Second, we will show how to *combine* AND triples to eliminate the adversary's chance of winning.

Leaky AND Triples

Our first goal is to build a triple $\{\alpha\}, \{\beta\}, \{\alpha\beta\}$, but with the following caveat: The adversary is allowed to guess α. If the adversary guesses correctly, then they learn that their guess is correct, and the protocol proceeds silently without the honest party noticing. If the adversary guesses incorrectly, then the protocol aborts. Notice that because α is uniform, the adversary can only guess correctly with probability $1/2$. Notice also that if we make a large number of triples, and if the adversary tries to guess on too many of them, the protocol is all but certain to abort.

Of course, we do not ultimately *want* these leaky forms of triples; this concession is made to the adversary only for the purposes of efficiency.

As a starting point, let's generate two random authenticated bits $\{\alpha, \beta\}$. Moreover, let's recall that each bit comes in two parts, one known to each party. Let's keep these parts separate for wire α. Namely, the parties hold the following:

$$\{\alpha_A\} \quad \{\alpha_B\} \quad \{\beta\}$$

Bit β is uniform; Bit α_i is known to party P_i, and it is uniform if party P_i is honest. The main idea is that it suffices for parties to compute both $\{\alpha_A\beta\}$ and $\{\alpha_B\beta\}$, since the following holds:

$$\{\alpha_A\beta\} \oplus \{\alpha_B\beta\} = \{\alpha\beta\}$$

Let's focus on the computation $\{\alpha_A\beta\}$; $\{\alpha_B\beta\}$ is computed symmetrically. To achieve such an authenticated value, we will use a garbled-circuit-like trick.

In particular, notice that there are only two possible values for α_A; B will, roughly, send to A two "garbled truth table rows" that compute the appropriate function. It will indeed be the case that a malicious A can garble each such row incorrectly, but if she does and B evaluates it, this will break B's MAC key, and so she will notice and abort. As we will see, if A corrupts a garbled row and B *does not* abort, then A will use this to infer that B must not have decrypted the corrupted row, which leaks α_B to A. This matches our concession in security: A can learn α_B—and hence α—with probability one half.

Recall that the authenticated share $\{\alpha_B\}$ includes a component of form $[\alpha_B\Delta_A]$. If we think about what this means in more detail, it means that A holds some share of form $X \in \{0,1\}^\lambda$, and B holds a matching share $X \oplus \alpha_B\Delta_A$. Or, in other words, B holds one of two possible "keys":

$$X \oplus \alpha_B\Delta_A = \begin{cases} X & \text{if } \alpha_B = 0 \\ X \oplus \Delta_A & \text{otherwise} \end{cases}$$

A will use these two keys to encrypt two "garbled rows".

We can also think about $\{\beta\}$ as composed of its shares. Namely, A holds some string $Y \in \{0,1\}^{2\lambda+1}$ and B holds the matching share $Y \oplus \beta(\Delta_A, \Delta_B, 1)$.

Let us assume we have a random oracle $H : \{0,1\}^\lambda \to \{0,1\}^{2\lambda+1}$. A samples a random string $Z \in_\$ \{0,1\}^{2\lambda+1}$ and sends to B the following two strings:

$$r_0 = H(X) \oplus Z$$
$$r_1 = H(X \oplus \Delta_A) \oplus Y \oplus Z$$

Now, B holds either X or $X \oplus \Delta_A$, and he knows which of these two cases he is in, since he knows α_B.

He acts conditionally, depending on α_B. He computes:

$$\begin{aligned}
&\begin{cases} r_0 \oplus H(X) & \text{if } \alpha_B = 0 \\ r_1 \oplus H(X \oplus \Delta_A) \oplus (Y \oplus \beta(\Delta_A, \Delta_B, 1)) & \text{if } \alpha_B = 1 \end{cases} \\
= &\begin{cases} (H(X) \oplus Z) \oplus H(X) & \text{if } \alpha_B = 0 \\ (H(X \oplus \Delta_A) \oplus Y \oplus Z) \oplus H(X \oplus \Delta_A) \oplus (Y \oplus \beta(\Delta_A, \Delta_B, 1)) & \text{if } \alpha_B = 1 \end{cases} \\
= &\begin{cases} Z & \text{if } \alpha_B = 0 \\ Z \oplus \beta(\Delta_A, \Delta_B, 1) & \text{if } \alpha_B = 1 \end{cases} \\
= &\ Z \oplus \alpha_B \beta(\Delta_A, \Delta_B, 1)
\end{aligned}$$

Thus, A holds Z and B holds a matching share $Z \oplus \alpha_B \beta(\Delta_A, \Delta_B, 1)$; If the parties behave honestly, they hold $\{\alpha_B \beta\}$.

Why this is maliciously secure. Let's argue security at an intuitive level.

First, suppose B is malicious. Here, security is quite straightforward: B holds key $X \oplus \alpha_B \Delta_A$, and not the flipped key $X \oplus \neg\alpha_B \Delta_A$. He cannot guess this latter key, because guessing this would involve guessing Δ_A. Thus, malicious B can only decrypt the single row as we intend. And, again, he cannot flip the output sharing $\{\alpha_B \beta\}$ because he cannot guess Δ_A.

Now, suppose A is malicious. The intuition here is that A's only available action is to send corrupted rows r_0, r_1. But corrupting rows will not help A flip the value of the output wire sharing, because that would involve guessing Δ_B. As we stated earlier, there is one avenue of attack for A: A can corrupt *one* of the rows and see if this causes B to abort the protocol later on. If it does not abort, then that tells A the value of α_B. For instance, if r_0 is incorrectly constructed and B never aborts, then that must mean that $\alpha_B = 1$.

Corrupting such a row in this way amounts to simply guessing α_B, which will succeed with probability $1/2$. Crucially, if A guesses wrong, then the protocol aborts. Also crucially, only *one* of the AND triple's input bits can be leaked. Namely, while the adversary can learn α, they have no mechanism by which to learn β.

Thus, the parties can compute $\{\alpha_B \beta\}$, and they can of course symmetrically compute $\{\alpha_A \beta\}$. By XORing these, we can obtain $\{\alpha\beta\}$. Again, if the triple is well-formed, then the adversary knows α with probability at most $1/2$.

Bucketing triples

Now that we can construct leaky triples, we are almost to our full desired functionality. The main idea will be to *combine* multiple triples together. In particular we will show how to combine two AND triples to yield a single one. The benefit of this is that while each input triple can be "corrupted" with probability at most $1/2$, the output triple can be corrupted with probability at most $1/4$. By repeating this, we can yield a single triple with negligible probability of being corrupted. The following description is adjusted from [WRK17].

In particular, let us suppose we have preprocessed two triples:

$$\{\alpha^0, \beta^0, \alpha^0\beta^0\}$$
$$\{\alpha^1, \beta^1, \alpha^1\beta^1\}$$

It is crucial to recall that while the adversary might know α^0, α^1, it *does not* know β^0, β^1. Parties first compute the following locally:

$$\{\alpha\} = \{\alpha^0 \oplus \alpha^1\}$$
$$\{\beta\} = \{\beta^0 \oplus \beta^1\}$$

Our output triple will be the following:

$$\{\alpha\}, \{\beta^0\}, \{\alpha\beta^0\}$$

(Note that this might be different than your expectation that the triple would be $\{\alpha\}, \{\beta\}, \{\alpha\beta\}$.) Notice the following:

$$\alpha\beta^0 = (\alpha^0 \oplus \alpha^1)\beta^0 = \alpha^0\beta^0 \oplus \alpha^1\beta^0$$

The parties already hold $\{\alpha^0\beta^0\}$, so it suffices to compute $\{\alpha^1\beta^0\}$. Notice the following:

$$\{\alpha^1\beta^0\} = \{\alpha^1(\beta^0 \oplus \beta^1)\} \oplus \{\alpha^1\beta^1\} = \{\alpha^1\beta\} \oplus \{\alpha^1\beta^1\}$$

Again, the parties already hold $\{\alpha^1\beta^1\}$, so it suffices to compute $\{\alpha^1\beta\}$. But this is easy: the parties simply reveal to themselves β. This is secure because bit β^1 is secret, random, and independent of β^0, so it acts as a one-time pad. From here parties can simply compute $\{\alpha^1\} \cdot \beta = \{\alpha^1\beta\}$. So summarizing:

- The parties compute two triples.
- They linearly combine the parts and open the β component.
- They then linearly combine the triples to obtain $\{\alpha, \beta^0, \alpha\beta^0\}$.

Notice that the adversary gained no additional information about α, and learned nothing about β^0. Thus, we can claim that the adversary knows α with probability at most $1/4$. Repeating this λ times yields a true triple.

Decreasing the overhead of bucketing via batching

So far, we need to bucket $\Omega(\lambda)$ triples to achieve suitable security for our triples. However, this is quite expensive, since each triple itself requires $\Theta(\lambda)$ bits to construct. Thus, we currently require $\Omega(\lambda^2)$ bits of communication per triple, which is quite expensive.

There is a simple observation we can leverage to do much better: typically, we want to preprocess a *large* number of triples, not just one. So, before we bucket, we will need to preprocess large numbers of leaky triples, not just λ of them. Recall that for each leaky triple the adversary tries to corrupt, they are caught with probability $1/2$. So even if we create a huge batch of $n \gg \lambda$ leaky triples, the adversary cannot corrupt more than λ of them without being caught with overwhelming probability.

This leads to a very simple insight: before we bucket the triples, randomly shuffle them. Namely, after all leaky triples are computed, the two parties agree on a public, randomly-chosen permutation. Then, we bucket adjacent triples. The insight is that in order for the adversary to actually attack the protocol, they must manage to completely fill a particular bucket with corrupted leaky triples. But their probability of doing so successfully is now *considerably worse*.

Namely, if our goal is to construct n total multiplication triples, some basic combinatoric analysis shows it now suffices to construct only $\Theta(n\lambda/\log n)$ leaky triples, rather than $\Theta(n\lambda)$.

Next Time

We have now successfully achieved the preprocessing protocol for our maliciously-secure variant of GMW. From here, we will learn how to achieve similar techniques in the constant round, garbled-circuit-based setting. In particular, next time we will study *authenticated garbling*, which combines our existing preprocessing protocol with garbled circuit techniques we have already seen. The result will be a constant-round, maliciously-secure 2PC protocol.

References

[KRRW18] Jonathan Katz, Samuel Ranellucci, Mike Rosulek, and Xiao Wang. Optimizing authenticated garbling for faster secure two-party computation. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 365–391. Springer, Cham, August 2018.

[WRK17] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 21–37. ACM Press, October / November 2017.