Authenticated Triples

CS 507, Topics in Cryptography: Secure Computation

David Heath

Fall 2026

Last time we 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 $\Delta_A, \Delta_B \leftarrow \{0,1\}^\lambda$ and then sends these to A (resp. B).
- **Random bits.** Upon receiving instructions from both parties, the functionality samples a bit $\alpha \leftarrow \{0,1\}$, samples an authenticated sharing $\{\alpha\}$, and deals shares to the parties.
- **Random triples.** Upon receiving instructions from both parties, the functionality samples two bits $\alpha, \beta \leftarrow \{0,1\}$, 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 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 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_A is known to A, and it is uniform if party A is honest (and respectively for bit α_B). 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 of $\{\alpha_B\beta\}$; $\{\alpha_A\beta\}$ is computed symmetrically. We use a garbled-circuit-like trick. There are only two possible values of α_B, so A sends B two "garbled rows", one per value, and B can decrypt only the row matching his α_B. In effect, A garbles a single AND gate whose input wire α_B has labels X and $X \oplus \Delta_A$. A malicious A can garble a row incorrectly, but an equality check will then force her to predict which row B decrypted: if she is wrong, the protocol aborts; if she is right, she has learned α_B, and hence α. This matches our concession.

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 $H : \{0,1\}^* \to \{0,1\}^{2\lambda+1}$. A samples a random string $Z \leftarrow \{0,1\}^{2\lambda+1}$ and sends to B the following two strings:

$$\begin{aligned} r_0 &= H(i, X) \oplus Z \\ r_1 &= H(i, X \oplus \Delta_A) \oplus Y \oplus Z \end{aligned}$$

Here i is some unique index per leaky triple, needed to argue formal security.

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:

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

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 $\{\alpha_B\beta\}$.

The parties compute $\{\alpha_A\beta\}$ symmetrically, with B sending the rows, and set $\{\alpha\beta\} := \{\alpha_A\beta\} \oplus \{\alpha_B\beta\}$.

A consistency check. One step remains. A malicious A could corrupt just one of her two rows, say by adding an error E to the Δ_A-part of r_0. Then B's share is off by E exactly when $\alpha_B = 0$. B cannot notice, since he does not know Δ_A, but A can: whenever a value derived from this share is later opened to her, B sends his Δ_A-part, and she sees whether E is present. She would learn α_B with no risk of abort.

To rule this out, the parties check that their shares of $\{\alpha\beta\}$ are consistent. Let $\mathcal{F}_{\text{EQ}}$ be an *equality functionality*: each party inputs a string, and both learn whether the two strings are equal. (This functionality is easy to realize with a random oracle. For instance, have one party commit to a hash of its string, the other send a hash of its own, and the first then open.)

1. The parties take a fresh authenticated random bit $\{\rho\}$ and locally compute $\{u\} := \{\alpha\beta\} \oplus \{\rho\}$. Write A's share as (T_A, T_B, t) and B's as (S_A, S_B, s); if both are honest, $(S_A, S_B, s) = (T_A, T_B, t) \oplus u(\Delta_A, \Delta_B, 1)$.

2. The parties exchange t and s and set $u := t \oplus s$. This reveals nothing about $\alpha\beta$, which is masked by ρ.

3. The parties call $\mathcal{F}_{\text{EQ}}$: A inputs $(T_A \oplus u\Delta_A,\ T_B)$ and B inputs $(S_A,\ S_B \oplus u\Delta_B)$. If the strings differ, the parties abort.

Roughly speaking, this check forces each party to declare what the other's MAC part should be under its own key. The bit $\{\rho\}$ is then discarded.

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. She cannot change the shared value, since that would require guessing Δ_B. Thus, her only option is to send corrupted rows, and the only corruption that matters is one that differs between the two rows (the same error in both rows is just a different choice of Z). After such a corruption, B's share depends on which row he decrypted, i.e. on α_B. To pass the equality check A must input exactly what B holds, so she must know which error he received: she must guess α_B. If she guesses wrong, the protocol aborts. If she guesses right, she has learned α_B.

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 hold $\{\alpha\beta\}$, and a cheating party learns α only by making a guess that aborts the protocol with probability 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 $\{\alpha^1\} \cdot \beta = \{\alpha^1\beta\}$.

Exercise 1 (Why not combine the β's too?)**.** *The combined triple is on* $(\alpha^0 \oplus \alpha^1), \beta^0$ *rather than the more symmetric* $(\alpha^0 \oplus \alpha^1), (\beta^0 \oplus \beta^1)$*. What goes wrong with the symmetric choice?*

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*.

To see why, let B be the bucket size, so there are nB leaky triples. If the adversary attacks more than λ of them, she survives the equality checks with probability at most $2^{-\lambda}$. Otherwise at most λ triples are attacked, and after the random permutation a fixed bucket consists entirely of attacked triples with probability at most $(\lambda/nB)^B \le (\lambda/n)^B$. By a union bound over the n buckets, the adversary knows some bucket completely with probability at most $n \cdot (\lambda/n)^B$. For $n \ge \lambda^2$ this is at most $n^{1-B/2}$, which is below $2^{-\lambda}$ once $B \ge 2\lambda/\log_2 n + 2$. So buckets of size $O(\lambda/\log n)$ suffice, for a total of $O(n\lambda/\log n)$ leaky triples. (A tighter analysis can further reduce the bucket size by $\approx 2\times$.)

Exercise 2. *Suppose the parties bucket adjacent leaky triples without the random permutation. Give an attack that learns* α *for one output triple, and state its success probability. What bucket size is then needed to enforce negligible advantage for the adversary?*

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.