FHE-MPC Notes

Lecturer: Nigel Smart — Lecture #8
Scribe: Peter Scholl — 28/11/11

In the previous lecture, it was shown how to perform secure multiparty computation using a multiplicative Linear Secret Sharing Scheme (LSSS). The resulting scheme had two main drawbacks:

- Perfect security only for *semi-honest* adversaries.
- Required secure point-to-point communication between all players.

To allow active as opposed to semi-honest adversaries can be dealt with via VSS, but this is inefficient and still does not avoid the secure point-to-point communication issue.

The requirement for secure channels stems from the algorithm for multiplying x and y: each player had to compute the local product of shares $p_i = [x]_i[y]_i$ and then securely distribute the shares of p_i to all other players. To allow the use of publicly broadcasted channels, a new technique for multiplication is required. This was introduced by Beaver in 1992 [1].

Initially, it is required that a secret *multiplication triple* is shared between all players. That is, the i-th player has the shares $[a]_i, [b]_i, [c]_i$, such that upon reconstruction, $a \cdot b = c$. Suppose we have already done this initial setup stage, and now want to compute the product of x and y, where each player has the shares $[x]_i, [y]_i$. Firstly, every player computes and publicly broadcasts the values:

$$\begin{aligned}[d]_i &= [x]_i - [a]_i \\ [e]_i &= [y]_i - [b]_i.\end{aligned}$$

Since $[a]_i$ and $[b]_i$ are only initially known to the i-th player, it follows that these values hide the corresponding shares of x and y. We also have that when reconstructed, $d = x - a$ and $e = y - b$. Next, each player reconstructs the values of d and e, and then computes

$$[z]_i = de + d[b]_i + e[a]_i + [c]_i.$$

This corresponds to a share of

$$\begin{aligned}z &= de + db + ea + c \\ &= (x-a)(y-b) + (x-a)b + (y-b)a + ab \\ &= xy\end{aligned}$$

and so we have successfully shared the product of x and y.

Active adversaries

To achieve security against active adversaries, some kind of error correction is needed. We need to be able to correct any errors introduced by the adversary in the broadcast of the shares of e and d. In the case of information theoretic security this means that the underlying LSSS must support a Q^3 adversary structure, or equivalently can be extended to a scheme which is strongly multiplicative. By a result of Cramer *et al.* [2], these properties are essentially the same as the LSSS being derived from an error correcting code. In the case of threshold adversaries this equates to the treshold satisfying $t < n/3$.

For the case of dishonest majority and computational adversaries a different technique is needed. We could use MACs to provide authentication of the communications. Essentially, every piece of transmitted data is augmented with a MAC to ensure that nobody can cheat.

Sharing a multiplication triple

Using a threshold LSSS

To generate a shared multiplication triple we can use pseudo-random secret sharing (PRSS), as introduced in the previous lecture. When using a Shamir secret sharing based scheme, the steps for each player to carry out are as follows:

- Using PRSS, generate shares $[a]_i$, $[b]_i$, $[r]_i$, of pseudo-random numbers $[a]$, $[b]$ and $[r]$, which are represented by degree t polynomials.
- Use *pseudo-random zero sharing* (PRZS) to generate shares $[z_i]$ of 0, represented by a degree $2t$ polynomial z.
- Compute $[s]_i = [a]_i[b]_i - [r]_i + [z]_i$, giving a degree $2t$ share of $ab - r$.
- Broadcast $[s]_i$, so every player knows $s = ab - r$.
- Compute the share $[c]_i = [s] + [r]_i$, giving a share of the product $[a] \cdot [b]$.

Note that, whilst the zero shares $[z]_i$ are not required for correctness, they are necessary to properly mask s. Although the random polynomial r provides some masking, it is only of degree t and so does not fully hide the degree $2t$ polynomial.

In the case of semi-honest adversaries the above protocol will be correct. For the case of active adversaries in the information theoretic model we simply require that the honest players can detect when an error occurs (since then we just abort the offline phase and can start again). Since the broadcast of $[s]_i$ is of a $2t$ sharing we will be able to detect errors as long as the number of honest players is larger than $2t$; i.e. we require $n < t/3$.

Using somewhat homomorphic encryption

We want to be able to perform multiparty computation when faced with a *dishonest majority* of adversaries. Moreover, we want these adversaries to be *active.* We need to be able to produce the triples, and the associated MACs mentioned above. To overcome these obstacles, we make use of a *somewhat homomorphic encryption* scheme to compute the multiplication triples. This avoids the dependency on a strongly multiplicative LSSS, thus allowing security against a dishonest majority (at the expensive of assuming a computational assumption). The steps for computing a multiplication triple using SHE are as follows (see [3]).

- Generate random shares $[a]_i$ and $[b]_i$ as before (for dishonest majority this is trivial).

- Broadcast the *encryptions* under the SHE scheme of these shares, given by $\mathsf{Enc}([a]_i)$, $\mathsf{Enc}([b]_i)$.
- Each player homomorphically computes $\mathsf{Enc}(a) = \sum_i \mathsf{Enc}([a]_i)$, $\mathsf{Enc}(b) = \sum_i \mathsf{Enc}([b]_i)$ and $\mathsf{Enc}(c) = \mathsf{Enc}(a) \cdot \mathsf{Enc}(b)$.
- Perform a threshold decryption of $\mathsf{Enc}(c)$, so each player ends up with a share $[c]_i$ of the product $a \cdot b$.

How to add MACs to the above computation is explained in [3].

References

[1] Beaver, D. Efficient multiparty protocols using circuit randomization. *Advances in Cryptology - CRYPTO 1991* (1991).

[2] Cramer, R., Damgård, I., Maurer, U. General secure multi-party computation from any linear secret-sharing scheme. *Advances in Cryptology - EUROCRYPT 2000* (2000).

[3] Damgård, I., Pastro, V., Smart, N., and Zakarias, S. Multiparty Computation from Somewhat Homomorphic Encryption. *eprint.iacr.org* (2011).