Introduction to Secure Multi-Party Computation

Johan Wallén

November 7, 2001

The problem

Consider a random process f mapping m inputs to m outputs, and m parties each having an input x_i. The ith party wants to obtain the ith output of $f(x_1, \ldots, x_2)$.

Can this be securely implemented?

Using a trusted party

In an **ideal** model with a **trusted third party**, the problem can be solved:

- Each party sends its input x_i to the TTP using a secure channel.
- The TTP computes $f(x_1, \ldots, x_m)$ and sends the ith output to party i using a secure channel.

Even this model has some unavoidable problems.

Unavoidable problems

A coalition of the parties can always:

- Jointly agree on which inputs to use.
- Share all the received outputs between them.
- Abort the protocol. That is, refuse to participate.

A solution to the multi-party computation problem lets the parties **simulate** this ideal model in a **real** model without any TTP.

The malicious models

When translated to a real model, **abortion** becomes more involved.

The coalition might be able to abort the protocol at a later stage, say, when some of the parties have obtained their outputs.

If a majority of the players are honest, it turns out that early abort can effectively be prevented.

Thus, we have two models: one **allowing abort** and one with a **honest majority**

The malicious models (cont)

In the **first** malicious model, early abort is not considered a break of security. On the other hand, it is secure as long as **at least one** party is honest.

In the **second** malicious model, early abort is not allowed. On the other hand, it is secure (only) if **more than half** of the players are honest.

The secure multi-party computation problem can be solved for both models assuming the existence of **trapdoor permutations**.

Overview of the constructions

We will first solve the problem for a **semi-honest** model.

Then, we will construct two **compilers**.

The **first** compiler transforms a protocol secure in the semi-honest model to a protocol secure in the first malicious model.

The **second** compiler transforms a protocol secure in the first malicious model to a protocol secure in the second malicious model.

Security arguments are based on simulation: Any adversary in one of the three real models can effectively be simulated in the corresponding ideal model.

Constructions for the semi-honest model

In the semi-honest model, each party follows the protocol. We must only make sure that the protocol does not leak any information.

The functionality is described by an **arithmetic circuit** over $\mathrm{GF}(2)$.

For each gate with actual inputs a, b, each party hold **shares** a_i and b_i adding up to the correct input value. By a gate emulation protocol, the parties obtain shares c_i adding up to the correct output value.

Oblivious transfer

In an oblivious transfer protocol, the **sender** holds k input bits $b_1, \ldots, b_k$, and the **receiver** holds an integer $j \in \{1, \ldots, k\}$.

The objective is to transfer the bit b_j to the receiver, so that the sender does not learn which bit was received, and the receiver does not learn anything about the other bits.

This is easily implementable in the semi-honest model.

Multiplication gates

The main problem is the emulation of multiplication gates.

It turns out that the problem can be reduced to **two-party** multiplication gate emulation.

Thus, party 1 holds a_1, b_1 and party 2 holds a_2, b_2. They wants to compute uniformly distributed shares c_1, c_2 such that $c_1 + c_2 = (a_1 + a_2)(b_1 + b_2)$.

Party 1 picks uniformly c_1. Then they invoke an **oblivious transfer** protocol to transfer one of the bits $c_1 + a_1 b_1$, $c_1 + (a_1 + 1)b_1$, $c_1 + a_1(b_1 + 1)$ or $c_1 + (a_1 + 1)(b_1 + 1)$ to party 2.

The output share of party 1 is c_1 and the share of party 2 is the bit received from the oblivious transfer protocol.

Forcing semi-honest behavior

The first compiler will augment the previous protocol to force semi honest behavior.

How can a malicious party misbehave?

- Replace its input at a later stage.
- Use coins that are not random.
- Send invalid messages during the protocol emulation.

Forcing semi-honest behavior (cont)

These problems will be solved by:

Input commitment: Each party commit to its input and sends commitments to the other parties. After this, all the inputs are fixed.

Coin tossing: A coin tossing into the well protocol is used for coin generation. Each party obtains uniformly distributed coins unknown to the other parties. The other parties obtain commitments to these coins.

Authenticated computation: When a party sends a message in the underlying protocol, it proves in zero knowledge to the other parties that the message was computed according to the protocol (using the correct input and random coins).

Preventing abort

Each party shares its input using a **verifiable secret sharing scheme**. The other parties obtains a share of the input and additional information that can be used to verify the correctness of the share.

If less than half of the parties abort after this stage, the honest parties can recover the inputs of the aborting parties and complete the protocol without them.

Conclusions

If trapdoor permutation exists, any m-ary functionality can be securely implemented in both malicious models.

Thus, virtually any protocol problem can be solved in principle.

Unfortunately, the techniques presented here result in inefficient protocols. Thus, direct constructions are needed in practice.

These results can, however, be considered as an ultimate plausibility result.