Multi Party Computation: From Theory to Practice

Nigel P. Smart

Department of Computer Science,
University Of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB.

November 22, 2012

What if?

Take two drug companies.

Each has a database of molecules and toxicology test results.

They want to combine their results

Without revealing what molecules are in the databases.

What if?

A government wants to search network traffic for a specific anomolous behaviour.

But the network operator does not want to give access to the network to the government.

And the government does not want to reveal exactly what behaviour it is searching for.

Computing on Encrypted Data

These are just some ideas of applications for Computing On Encrypted Data.

There are two main ways of doing this:

Fully Homomorphic Encryption

- First scheme developed in 2009
- Party A sends encrypted data to party B.
- Party B does some computation and returns the encrypted result to party A
- Party A now decrypts to find out the answer.

Multi-Party Computation

- First schemes developed in mid 1980's.
- Parties jointly compute a function on their inputs using a protocol
- No information is revealed about the parties inputs.

Theory

In theory both such technologies can compute anything.

In FHE one has a huge computational cost, but zero communication.

In MPC one has virtually no computational cost, but huge communication.

In theory we can make either technology error tolerent

- Even against malicious players.

Practice

FHE is currently impractical for all but the simplest functions

- Although you can do some useful things with it.

MPC has been deployed for some operations

- Mainly against semi-honest adversaries.
- Tolerating only one baddie out of exactly three players.

We will show how to combine FHE and MPC to get something much better and practical.

Set up

Assume n parties of which $n-1$ can be malicious.

Assume a global (secret) key $\alpha \in \mathbb{F}_p$ is determined

Each party i holds α_i with

$$\alpha = \alpha_1 + \ldots + \alpha_n.$$

Secret Sharing

All data is represented by elements in $\mathbb{F}_p$.

A secret value $x \in \mathbb{F}_p$ is shared between the parties as follows

- Party i holds a data share x_i
- Party i holds a "MAC" share $\gamma_i(x)$

such that

$$x = x_1 + \cdots + x_n \quad \text{and} \quad \alpha \cdot x = \gamma_1(x) + \cdots + \gamma_n(x).$$

Note we can share a public constant x by

- Party 1 sets $x_1 = x$
- Party $i \neq 1$ sets $x_i = 0$
- Party i sets $\gamma_i(x) = \alpha_i \cdot x$.

Preprocessing Model

Such a sharing of x is denoted by $[x]$.

Our protocol works in the preprocessing model.

We (overnight say) generate a lot of data which is independent of the function to be computed, or its inputs.

In its basic form the data consists of triples of shared values

$$[a], [b], [c]$$

such that

$$c = a \cdot b.$$

We discuss how to produce these triples later.

The Computation

To perform the computation we utilize the following idea

Any computation can be represented by a series of additions and multiplications of elements in $\mathbb{F}_p$.

In other words $+$ and $\times$ are a set of Universal Gates over $\mathbb{F}_p$.

We assume the players inputs are shared first using the above sharing

- Will not explain how to do this, but it is easy

So all we need do is working out how to add and multiply shared values.

Addition will be easy, multiplication will be hard.

Addition

Suppose we have two shared values $[x]$ and $[y]$.

To compute the result $[z]$ of an addition gate the parties individually execute

- $z_i = x_i + y_i$
- $\gamma_i(z) = \gamma_i(x) + \gamma_i(y)$

Note this is a local operation and that we end up with

$$\begin{aligned} z &= \sum z_i = \sum (x_i + y_i) = \left(\sum x_i\right) + \left(\sum y_i\right) \\ &= x + y, \\ \alpha \cdot z &= \sum \gamma_i(z) = \sum (\gamma_i(x) + \gamma_i(y)) = \alpha \cdot x + \alpha \cdot y \\ &= \alpha \cdot (x + y). \end{aligned}$$

Linear Secret Sharing

The addition trick works because we have a Linear Secret Sharing Scheme.

We can locally compute any linear function of shared values

i.e. given constants a, b and c and shared values $[x]$ and $[y]$ we can compute

$$a \cdot [x] + b \cdot [y] + c = [a \cdot x + b \cdot y + c].$$

We will now use this in our method to perform multiplication.

Note: In what follows "partially opening" a share $[x]$ means revealing x_i but not the MAC share.

Multiplication

To multiply $[x]$ and $[y]$ to obtain $[z]$ we work as follows:

- Take a new triple $([a], [b], [c])$ off the precomputed list.
- Partially open $[x] - [a]$ to obtain $\epsilon = x - a$.
- Partially open $[y] - [b]$ to obtain $\rho = y - b$.
- Locally compute the linear function

$$[z] = [c] + \epsilon \cdot [b] + \rho \cdot [a] + \epsilon \cdot \rho.$$

Note

- Each multiplication requires interaction
- If a (resp. b) is random then ϵ (resp. ρ) is a one-time pad encryption of x (resp. y).

We get the correct result because

$$\begin{aligned}
&c + \epsilon \cdot b + \rho \cdot a + \epsilon \cdot \rho \\
&\quad = a \cdot b + (x - a) \cdot b + (y - b) \cdot a + (x - a) \cdot (y - b) \\
&\quad = (a \cdot b) + (x \cdot b - a \cdot b) + (y \cdot a - a \cdot b) + (x \cdot y - x \cdot b - y \cdot a + a \cdot b) \\
&\quad = x \cdot y.
\end{aligned}$$

Computation

So given we can add and multiply we can compute anything

At the end of the computation we check correctness by interactively checking the MAC values are all correct.

- Again this is easy to do.
- Does not require revealing α.

In practice our current implementation can perform around half a million multiplications per second

- About the speed of a 286 machine from 1981.
- Although comparison and other bit-operations take longer than on a 286.

Preprocessing and FHE

We return to the preprocessing, which we do using FHE

- Following is a naive version, the real version has lots of bells and whistles.

Assume a shared FHE public key pk for an FHE scheme.

- Party i holds a share sk_i
- Together they can decrypt a ciphertext ct via $\mathsf{Dec}_{\mathsf{sk}_1,\ldots,\mathsf{sk}_n}(\mathsf{ct})$.
- Adding and multiplying ciphertexts means underlying plaintexts get added and multiplied.
- Each party computes $\mathsf{Enc}_{pk}(\alpha_i)$ and broadcasts this.

Last step needed so that each party has $\mathsf{Enc}_{pk}(\alpha)$.

Reshare

Given a ciphertext ct encrypting a value m we can make each party obtain

- An additive share m_i, s.t. $m = \sum m_i$
- And (if needed) a new fresh ciphertext ct' encrypting m.

Reshare(ct)

- Party i generates a random f_i and transmits $\mathsf{ct}_{f_i} = \mathsf{Enc}_{\mathsf{pk}}(f_i)$.
- All compute $\mathsf{ct}_{m+f} = \mathsf{ct} + \sum \mathsf{ct}_{f_i}$.
- Execute $\mathsf{Dec}_{\mathsf{sk}_1,\ldots,\mathsf{sk}_n}(\mathsf{ct}_{m+f})$ to obtain $m + f$.
- Party 1 sets $m_1 = (m + f) - f_1$.
- Party $i \neq 1$ sets $m_i = -f_i$.
- Set $\mathsf{ct}' = \mathsf{Enc}_{\mathsf{pk}}(m + f) - \sum \mathsf{ct}_{f_i}$.

Use some “default” randomness for the last encryption.

Generating $[a]$ and $[b]$

We can generate our sharing $[a]$ as follows

- Party i generates a random a_i and transmits $\mathsf{ct}_{a_i} = \mathsf{Enc}_{\mathsf{pk}}(a_i)$.
- All compute $\mathsf{ct}_a = \sum \mathsf{ct}_{a_i}$.
- All compute $\mathsf{ct}_{\alpha \cdot a} = \mathsf{ct}_\alpha \cdot \mathsf{ct}_a$.
- Execute Reshare on $\mathsf{ct}_{\alpha \cdot a}$ so party i obtains $\gamma_i(a)$.

Note this can also be executed to obtain $[b]$.

Generating $[c]$

This is also easy

- We have ct_a and ct_b.
- All compute $\mathsf{ct}_c = \mathsf{ct}_{a \cdot b}$ from $\mathsf{ct}_a \cdot \mathsf{ct}_b$.
- Get shares c_i via executing Reshare on ct_c; also obtaining a fresh ciphertext ct'_c.
- All compute $\mathsf{ct}_{\alpha \cdot c} = \mathsf{ct}_\alpha \cdot \mathsf{ct}'_c$.
- Execute Reshare on $\mathsf{ct}_{\alpha \cdot c}$ so party i obtains $\gamma_i(c)$.

This is efficient despite using FHE technology because we only compute with depth one circuits.

Similar tricks with FHE allow us to perform other preprocessing making the computation phase even faster.

The Future

The above is called the SPDZ protocol

Very efficient and practical for some applications.

Better security properties than other MPC implementations

More flexible in terms of parameters than other MPC implementations.

Currently looking around for commercial applications and partners to take this forward.

Any Questions ?