Secure MPC Based on Secret Sharing

N.P. Smart

COSIC,
KU Leuven, ESAT,
Kasteelpark Arenberg 10, bus 2452,
B-3001 Leuven-Heverlee,
Belgium.

August 31, 2018

Outline

What Questions Will We Answer?

What is Shamir secret sharing?

How can we obtain a passively secure protocol when $t < n/2$?

When is a secret sharing scheme multiplicative?

How to authenticate the computation via MACs.

How to generate random sharings quickly

How to perform reactive computation.

How to work with any LSSS, and not just multiplicative ones.

Reed–Solomon Codes

Shamir secret sharing and Reed–Solomon codes are highly related.

A Reed–Solomon code is defined by two integers (n, t) with $t < n$.

It can detect $n - t - 1$ errors, and correct $(n - t - 1)/2$ errors.

Take a finite field $\mathbb{F}_q$ with $q > n$.

Reed–Solomon Codes

Consider the set of polynomials of degree less than or equal to t over $\mathbb{F}_q$

$$\mathbb{P} = \{f_0 + f_1 \cdot X + \cdots + f_t \cdot X^t : f_i \in \mathbb{F}_q\}.$$

This defines the set of code-words in our code, equal to q^{t+1}.

The actual code words are given by

$$\mathbb{C} = \{(f(1), f(2), \ldots, f(n)) : f \in \mathbb{P}\}.$$

Think of f as the message and $c \in \mathbb{C}$ as the codeword.

- There is redundancy in this representation
- $t \cdot \log_2 q$ bits of information are represented by $n \cdot \log_2 q$ bits.

Reed–Solomon Codes

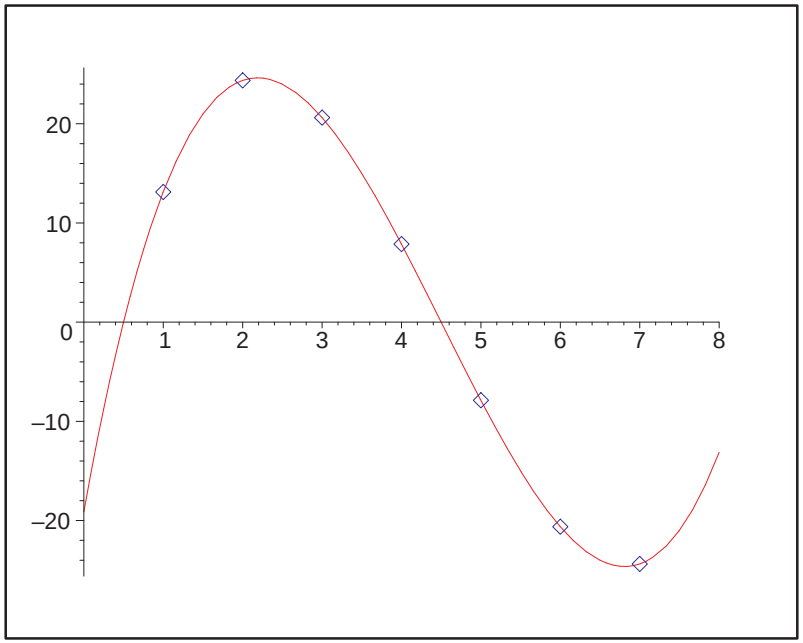

Figure : Cubic function evaluated at seven points

Reed–Solomon Codes: Data Recovery

Given a received code word $c = (c_1, \ldots, c_n) = (f(1), \ldots, f(n))$, we want to recover the message f.

This is equivalent to solving the system of equations

$$\begin{aligned} c_1 &= f_0 + f_1 \cdot 1 + \cdots + f_t \cdot 1^t, \\ &\vdots \qquad\qquad\qquad \vdots \\ c_n &= f_0 + f_1 \cdot n + \cdots + f_t \cdot n^t. \end{aligned}$$

So to solve for f_i we just need to invert this linear system.

Reed–Solomon Codes: Data Recovery

This can be (more easily done via Lagrange interpolation)

Take the values

$$\delta_i(X) \leftarrow \prod_{1 \leq j \leq n, i \neq j} \frac{X - j}{i - j}, \quad 1 \leq i \leq n.$$

Note that we have the following properties, for all i,

- $\delta_i(i) = 1$.
- $\delta_i(j) = 0$, if $i \neq j$.
- $\deg \delta_i(X) = n - 1$.

Lagrange interpolation takes the values c_i and computes

$$f(X) \leftarrow \sum_{i=1}^{n} c_i \cdot \delta_i(X).$$

Reed–Solomon Codes: Error Detection

If we get an error then when we perform the message recovery we will *definitely* not get a polynomial of degree t

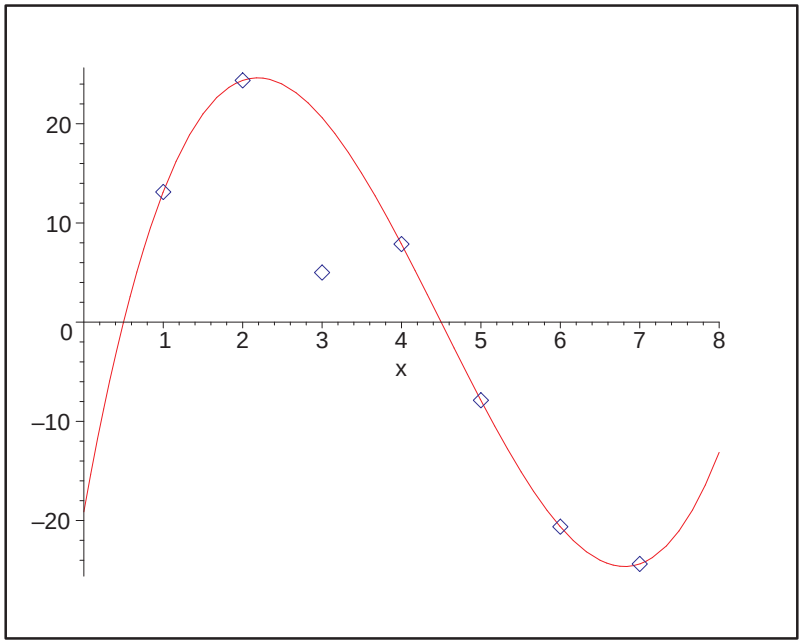

Figure : Cubic function going through six points and one error point

But if the number of errors $e \leq t < n/2$, then we detect all errors.

Reed–Solomon Codes: Error Correction

We would like to be able to recover from errors.

Which we can do if e is forced to be smaller $(n - t - 1)/2$.

A naive way of doing this is to try all subsets of size $n - e$ to try and find a polynomial of degree t.

A better method is to use the Berlekamp-Welch algorithm (see any decent textbook on coding theory).

Shamir Secret Sharing

We will use Reed–Solomon codes to define a secret sharing scheme

We map secrets $s \in \mathbb{F}_q$ to the set $\mathbb{P}$ by associating a polynomial with the secret given by the constant term

For n parties we then distribute the shares as the elements of the code word

- So party i gets $s_i = f(i)$ for $1 \leq i \leq n$.

Secret reconstruction is via

$$s \leftarrow f(0) = \sum_{i=1}^{n} s_i \cdot \delta_i(0).$$

Actually any $t + 1$ parties can recover the secret.

Shamir Secret Sharing

A set of honest parties do not reveal their shares to anyone unless they want to.

A passive adversary controlling a subset A wants to learn the secret from the honest parties.

- This means $t \geq |A|$ to ensure privacy.
- Shamir is said to be a *threshold secret sharing scheme*
- If $|A| \leq t$ the adversary learns nothing at all about the secret.

The number of honest parties must be able to recover the secret, so we have

$$n - |A| > t \geq |A|$$

i.e.

$$|A| < n/2.$$

Same condition as error detection of Reed–Solomon codes.

Shamir Secret Sharing

An active adversary is one which will lie about its shares

- In order for the honest parties to recover the wrong secret

To protect against this we use the error correcting property of Reed–Solomon codes.

If the adversary is of size $|A| \leq (n - t - 1)/2$ we can recover the secret, i.e.

$$t < n - 2 \cdot |A|$$

To maintain security we require $|A| \leq t$

So

$$|A| < n - 2 \cdot |A|$$

i.e.

$$|A| < n/3$$

Linear Secret Sharing Schemes

A LSSS is a secret sharing scheme over $\mathbb{F}_q$ given by the following construction.

We have a matrix $M \in \mathbb{F}_q^{m \times d}$ and a vector $\mathbf{v} \in \mathbb{F}_q^d$ and a map $\chi : \{1, \ldots, m\} \longrightarrow \{1, \ldots, n\}$.

To share a value $s \in \mathbb{F}_q$ one picks a vector $\mathbf{k} \in \mathbb{F}_q^d$ such that

$$s = \mathbf{v}^{\mathsf{T}} \cdot \mathbf{k}.$$

We also compute

$$\mathbf{s} = (s_1, \ldots, s_m)^{\mathsf{T}} = M \cdot \mathbf{k}$$

with party j getting share s_i if $\chi(i) = j$.

Linear Secret Sharing Schemes

A set of parties A can recover the secret if the span of the rows i of M indexed by $\chi(i) = j$ for $j \in A$ contain $\mathbf{v}$

i.e. (with an obvious notation) there is an $\mathbf{x}_A$ such that

$$\mathbf{v}^T = \mathbf{x}_A^T \cdot M_A$$

Note we have $\mathbf{s}_A = M_A \cdot \mathbf{k}$ and so we have

$$\mathbf{x}_A^T \cdot \mathbf{s}_A = \mathbf{x}_A^T \cdot (M_A \cdot \mathbf{k}) = (\mathbf{x}_A^T \cdot M_A) \cdot \mathbf{k} = \mathbf{v}^T \cdot \mathbf{k} = s.$$

Linear Secret Sharing Schemes

A LSSS allows parties to compute arbitrary linear functions of the underlying secrets *without interaction*

Let s and s' be shared by $\mathbf{s} = M \cdot \mathbf{k}$ and $\mathbf{s}' = M \cdot \mathbf{k}'$ then

- $s + s'$ is shared by $\mathbf{s} + \mathbf{s}' = M \cdot (\mathbf{k} + \mathbf{k}')$ as $s + s' = \mathbf{v}^{\mathsf{T}} \cdot (\mathbf{k} + \mathbf{k}')$.
- $\alpha \cdot s$ for $\alpha \in \mathbb{F}_q$ is shared by $\alpha \cdot \mathbf{s} = M \cdot (\alpha \cdot \mathbf{k})$.

We write $[s]$ to denote s is shared under the LSSS

We write

$$[z] \leftarrow \alpha[x] + \beta[y] + \gamma$$

to denote parties executing the above steps to produce a sharing of z given sharings of x and y.

Shamir is an LSSS

Shamir is a LSSS since we have

$$M = \begin{pmatrix} 1 & 1 & \dots & 1^t \\ 1 & 2 & \dots & 2^t \\ \vdots & & & \vdots \\ 1 & n & \dots & n^t \end{pmatrix} \in \mathbb{F}_q^{n \times (t+1)}$$

$$\mathbf{v} = (1, 0, \dots, 0) \in \mathbb{F}_q^{t+1}$$

and for all i we have $\chi(i) = i$.

Full Threshold Secret Sharing

Another important LSSS is that of full threshold.

A secret is shared among n players as

$$s = s_1 + \cdots + s_n$$

This is also an LSSS with...

$$M = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & & & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix} \in \mathbb{F}_q^{n \times n}$$

$$\mathbf{v} = (1, 1, \dots, 1) \in \mathbb{F}_q^n$$

and for all i we have $\chi(i) = i$.

Replicated Secret Sharing $(t, n) = (1, 3)$

Many LSSS can define a given access structure.

Consider the following replicated scheme for the threshold $(t, n) = (1, 3)$ case

$$M = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \in \mathbb{F}_q^{6\times 3}$$

$$\mathbf{v} = (1, 1, 1) \in \mathbb{F}_q^3$$

and for all i we have $\chi(i) = \lceil i/2 \rceil$.
Multiple parties hold the same share in this scheme (hence the name replicated)

Schur Product

Suppose each party i holds a vector of shares $\mathbf{s}_i$ for each secret s

- In Shamir this a single value.

The Schur product of two such sharings

$$(\mathbf{s}_1, \ldots, \mathbf{s}_n) \text{ and } (\mathbf{s}'_1, \ldots, \mathbf{s}'_n)$$

is the local tensor of each parties

$$\mathbf{s}_i \otimes \mathbf{s}'_i.$$

In the case of Shamir this just means locally multiply the shares together to get one share.

In our replicated example each party will have four elements in $\mathbf{s}_i \otimes \mathbf{s}'_i$.

Multiplicative Secret Sharing

A LSSS is said to be multiplicative if given two sharings

$$(\mathbf{s}_1, \ldots, \mathbf{s}_n) \text{ and } (\mathbf{s}'_1, \ldots, \mathbf{s}'_n)$$

of two secrets s and s' then there is a set of vectors $(\mathbf{v}_1, \ldots, \mathbf{v}_n)$ such that

$$|\mathbf{v}_i| = |\mathbf{s}_i \otimes \mathbf{s}'_i|$$

and

$$s \cdot s' = \sum_{i=1}^{n} \mathbf{v}_i \cdot (\mathbf{s}_i \otimes \mathbf{s}'_i)$$

i.e. the product is a linear sum of the Schur product.

Shamir is Multiplicative

If $t < n/2$ then Shamir is multiplicative:

Given s and s' shared by polynomials f and f' of degree t.

The Schur product held by party i is $f(i) \cdot f'(i)$.

$s \cdot s'$ is shared by the polynomial $g = f \cdot f'$ of degree $2 \cdot t$

The shares of g are $g(i) = f(i) \cdot f'(i)$.

Since $2 \cdot t < n$ the Lagrange coefficients give us how to express $s \cdot s'$ in terms of a linear combination of the $g(i)$.

The Replicated Example is Multiplicative

To share $s = k_1 + k_2 + k_3$

- Party one holds (k_2, k_3).
- Party two holds (k_1, k_3).
- Party three holds (k_1, k_2).

Given two sharings of s and s' the product is given by

$$\begin{aligned} s \cdot s' &= (k_1 + k_2 + k_3) \cdot (k_1' + k_2' + k_3') \\ &= k_1 \cdot k_1' + k_1 \cdot k_2' + k_1 \cdot k_3' \\ &\quad + k_2 \cdot k_1' + k_2 \cdot k_2' + k_2 \cdot k_3' \\ &\quad + k_3 \cdot k_1' + k_3 \cdot k_2' + k_3 \cdot k_3' \\ &= (k_2 \cdot k_2' + k_2 \cdot k_3' + k_3 \cdot k_2') \\ &\quad + (k_3 \cdot k_3' + k_1 \cdot k_3' + k_3 \cdot k_1') \\ &\quad + (k_1 \cdot k_1' + k_1 \cdot k_2' + k_2 \cdot k_1') \end{aligned}$$

Full Threshold is Not Multiplicative

Not all LSSS are multiplicative.

For example the full threshold LSSS is not multiplicative.

Indeed a LSSS being multiplicative is a *very special* property.

For access structures for which there is a multiplicative LSSS we can easily define MPC protocols....

Maurer's Multiplication Protocol

If we have a multiplicative LSSS then we can not only compute linear functions (for free), we can also perform multiplication.

This forms the basis of many MPC protocols.

Step 1: Form the Schur product of the parties shares.
Step 2: Express the product as a sum of the local Schur products.
Step 3: Reshare the resulting full threshold sharing.
Step 4: Recombine the resultings shares locally.

Only step 3 requires interaction.

We explain with a Shamir and a Replicated example...

Multiplication Shamir

We have $s_i = f(i)$ and $s'_i = f'(i)$ sharing s and s'.

Parties form the Schur products locally $t_i = s_i \cdot s'_i$.

We know, as $t < n/2$, that there exists λ_i such that

$$s \cdot s' = \lambda_1 \cdot t_1 + \ldots + \lambda_n \cdot t_n.$$

Parties now compute $u_i = \lambda_i \cdot t_i$, so we actually have a full threshold sharing of the product

$$s \cdot s' = u_1 + \ldots + u_n.$$

Multiplication Shamir

Party i now creates a sharing $[u_i]$ of u_i and sends the shares to each player.

That is

- Party i generates a polynomial $g_i(X)$ of degree t such that $g_i(0) = u_i$.
- Party i sends party j the value $g_i(j)$.

The resulting sharing of u_i we call $[u_i]$.

All parties can then compute a Shamir sharing of degree t of the product $s \cdot s'$ by computing the linear function

$$[s \cdot s'] = [u_1] + \ldots + [u_n]$$

locally.

Multiplication Replicated

We can do the same with our replicated scheme

Assuming we have sharings $s = k_1 + k_2 + k_3$ and $s' = k'_1 + k'_2 + k'_3$

- Party one holds (k_2, k_3), (k'_2, k'_3).
- Party two holds (k_1, k_3), (k'_1, k'_3).
- Party three holds (k_1, k_2), (k'_1, k'_2).

We can now express the product as a full threshold sharing of terms from the Schur products

$$s \cdot s' = u_1 + u_2 + u_3$$

where party i has u_i and

$$\begin{aligned} u_1 &= k_2 \cdot k'_2 + k_2 \cdot k'_3 + k_3 \cdot k'_2 \\ u_2 &= k_3 \cdot k'_3 + k_1 \cdot k'_3 + k_3 \cdot k'_1 \\ u_3 &= k_1 \cdot k'_1 + k_1 \cdot k'_2 + k_2 \cdot k'_1 \end{aligned}$$

Multiplication Replicated

Each party then reshares u_i with respect to the replicated scheme.

So (for example) party one forms $[u_1]$ by writing

$$u_1 = v_1 + v_2 + v_3$$

and then sends

- (v_1, v_3) to party two.
- (v_1, v_2) to party three.

Party one retains the values (v_2, v_3).

The final sum is then achieved as before

$$[s \cdot s'] = [u_1] + [u_2] + [u_3].$$

Multiplicative LSSS Based MPC

We can now define a passively secure MPC protocol for *any* multiplicative LSSS based MPC

The MPC protocol will tolerate any set of adversaries who are not qualified for the LSSS.

The protocol is *information theoretically* secure.

Take the function to be computed as f

- Write f as an arithmetic circuit over $\mathbb{F}_q$
- i.e. As a sequence of $+$ and $\cdot$ operations in $\mathbb{F}_q$.

MPC Protocol

Input: When parties enter a value x they just transmit the sharing $[x]$ to the players.

Add: Parties can locally add shares

$$[x + y] = [x] + [y]$$

as the secret sharing scheme is linear

Mult: Parties can multiply shares with one round of interaction

$$[x \cdot y] = [x] \cdot [y]$$

using Maurer's protocol

Output: For party P_i to obtain output of $[x]$ all parties just open their shares of $[x]$ to P_i.

Other Multiplication Protocols

There are other methods to perform multiplication which we now outline

- Damgård-Nielsen method.
- Beaver Method.

Both methods use a form of "pre-processing"

In both cases we assume our base sharing $[x]$ is given by a LSSS

- $(M, \mathbf{v})$.

Given $[x]$ and $[y]$ we want to compute $[z] = [x \cdot y]$.

Damgård-Nielsen Method

This basically uses a second auxiliary secret sharing scheme

The auxiliary scheme we denote by $\langle x \rangle$ and it denotes a full threshold scheme.

Suppose we have a method to produce two random sharings of the same value

$$[r] \text{ and } \langle r \rangle$$

So

- $r = \mathbf{v}^{\mathsf{T}} \cdot \mathbf{k}$.
- $\mathbf{r} = M \cdot \mathbf{k} = (\mathbf{r}_1, \ldots, \mathbf{r}_n)$.
- $\langle r \rangle = (r_1, \ldots, r_n)$ s.t. $r = r_1 + \ldots + r_n$. This can be done non-interactively using a PRSS (see later).

Damgård-Nielsen Method

To multiply we then do the following for a sharing $[x]$ and $[y]$ given by $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$ and $(\mathbf{y}_1, \ldots, \mathbf{y}_n)$.

- Form the local Schur products $\mathbf{x}_i \otimes \mathbf{y}_i$.
- Think of the product z as a full threshold sum $\langle z \rangle$ via the Schur products

$$z = z_1 + \ldots + z_n$$

- Form $\langle d \rangle = \langle z \rangle + \langle r \rangle$
- Open $\langle d \rangle$ to player one (say)
- Player one now broadcasts d to all players
- All players compute

$$[z] = d - [r]$$

Note this takes two rounds of communication

- But we only transmit $2 \cdot (n-1)$ finite field elements in total.

Beaver Multiplication

A third method of multiplication produces a one round online multiplication protocol, but requires some (potentially) expensive offline pre-processing

Assume we can produce random triples (so called Beaver Triples)

$$[a], \quad [b], \quad [c]$$

where

- a, b are random finite field elements unknown to anyone
- $c = a \cdot b$.

This production can be done in an offline phase using one of the previous multiplication protocols, or via other techniques (see later) when the LSSS is not multiplicative.

Beaver Multiplication

To multiply $[x]$ and $[y]$ we take a new Beaver Triple $([a],[b],[c])$ and execute

- All parties open $[\epsilon] = [x] - [a]$.
- All parties open $[\delta] = [y] - [b]$.
- Compute

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

Notice that ϵ is a one-time pad encryption of x under a

And that δ is a one-time pad encryption of y under b

The final computation is a linear function and so can be computed locally.

Beaver Multiplication

Multiplication is correct because...

$$
\begin{aligned}
[z] &= [c] + \epsilon \cdot [b] + \delta \cdot [a] + \epsilon \cdot \delta \\
&= [c] + ([x] - [a]) \cdot [b] + ([y] - [b]) \cdot [a] + ([x] - [a]) \cdot ([y] - [b]) \\
&= [c] + ([x \cdot b] - [a \cdot b]) + ([y \cdot a] - [b \cdot a]) \\
&\quad + ([x \cdot y] - [x \cdot b] - [a \cdot y] + [a \cdot b]) \\
&= [c] - [b \cdot a] + [x \cdot y] \\
&= [a \cdot b] - [b \cdot a] + [x \cdot y] \\
&= [x \cdot y]
\end{aligned}
$$

Passive to Active

We really want an actively secure protocol.

The traditional way of doing this, in the information theoretic setting, is to use Verifiable Secret Sharing

We restrict to LSSS which are *strongly multiplicative*

- Think Shamir when $t < n/3$

And then use *error correcting* properties to correct for adversarial errors, or work out who the adversary is.

Passive to Active

This technique is relatively slow, but does lead to robust protocols.

Modern method is to be satisfied with active-with-abort security.

So we just want to detect an error, not correct it.

We are also prepared to give up on pure information theoretic protocols, if it gives us extra performance.

First Protocol

The first actively secure protocol we give is actually the most modern

- **Fast Large-Scale Honest-Majority MPC for Malicious Adversaries**
- Koji Chida and Daniel Genkin and Koki Hamada and Dai Ikarashi and Ryo Kikuchi and Yehuda Lindell and Ariel Nof
- `https://eprint.iacr.org/2018/570`

The basic idea is to authenticate each share with a shared MAC value

- This idea is much older

The protocol works when we have *any* method to perform passive multiplication protocol, e.g. Maurer.

Set Up

We assume we have a multiplicative LSSS scheme.

Elements $x \in \mathbb{F}_q$ when in secret shared form are denoted by $[x]$.

We can perform linear operations for free

$$[z] \leftarrow \alpha \cdot [x] + \beta \cdot [y] + \gamma.$$

Multiplication can be done via a passively secure protocol

$$[x \cdot y] \leftarrow [x] \cdot [y].$$

Our aim is an active-with-abort secure protocol.

- We give rough ideas at all stages, some details may be wrong/insecure

Authenticating a Computation Via MACs

A basic idea behind many protocols is the following.

Suppose we have a global key $\alpha \in \mathbb{F}_q$

To authenticate $x \in \mathbb{F}_q$ we use the "MAC" $\gamma = \alpha \cdot x$.

If we keep x, γ and α hidden from the adversary, but make the relationship consistent, then the adversary cannot change x without also changing either γ or α or both.

Initialization Step

We first have an initialization stage:

Init:

- Player P_i generates $\alpha_i \in \mathbb{F}_q$.
- Player P_i shares $[\alpha_i]$ to all players.
- The players compute $[\alpha] = [\alpha_1] + \ldots + [\alpha_n]$.

Note, all players now have a sharing of a value $[\alpha]$ that no one knows.

In our protocol we will store a secret value x as a double sharing $\langle x \rangle = ([x], [\alpha \cdot x])$.

The computation of $[\alpha]$ could be done via a PRSS (see later)

Arithmetic on $\langle x \rangle$ Shares

If $[\cdot]$ is a LSSS then so is the authenticated sharing.

$$\begin{aligned}
\langle x \rangle + \langle y \rangle &= ([x] + [y], [\alpha \cdot x] + [\alpha \cdot y]) \\
\langle x \rangle + c &= ([x] + c, [\alpha \cdot x] + c \cdot [\alpha]) \\
a \cdot \langle x \rangle &= (a \cdot [x], a \cdot [\alpha \cdot x])
\end{aligned}$$

We can also execute a passively secure multiplication protocol as follows, given a passive multiplication protocol for $[\cdot]$.

- On input of $\langle x \rangle$ and $\langle y \rangle$.
- Compute $[z] = [x \cdot y] = [x] \cdot [y]$ using the passive protocol for $[\cdot]$.
- Compute $[\alpha \cdot z] = [y] \cdot [\alpha \cdot x]$, again using the passive protocol for $[\cdot]$.
- Return $\langle z \rangle = ([z], [\alpha \cdot z])$.

Random Values

We need a method to generate random "authenticated" values $\langle r \rangle$

- Player P_i generates $r_i \in \mathbb{F}_q$.
- Player P_i shares $[r_i]$ to all players.
- The players compute $[r] = [r_1] + \ldots + [r_n]$.
- The players execute the passively secure multiplication protocol to generate $[\alpha \cdot r]$.
- The players now have $\langle r \rangle$.

Later we will see how to replace the first three steps using a PRSS.

Input Values

Suppose player P_i wants to enter a value x into the computation

- Take a random authenticated value $\langle r \rangle = ([r], [\alpha \cdot r])$
- The parties "open" the value $[r]$ to player i.
- Player P_i broadcasts $e = x - r$.
- The players set $\langle x \rangle = e + \langle r \rangle$.

Note that the value broadcast by player P_i is a one-time pad encryption of x under the key r.

Output Values

Outputting values is trivial, but before doing so we need to check that the adversary has not cheated.

The only places the adversary can cheat is either in producing a random value (by not entering a valid sharing at some stage), or in the multiplication protocol.

So we need to check all input values have been $\langle \cdot \rangle$-shared correctly, as has the output from all multiplication gates.

Let the input wires be $\{\langle v_m \rangle\}_{m=1}^{N}$.
Let the output wires of all multiplication gates be $\{\langle z_k \rangle\}_{k=1}^{M}$.

MAC Checking

The checking protocol proceeds as follows:

- The parties agree on random values $a_1, \ldots, a_M, b_1, \ldots, b_N \in \mathbb{F}_q$
- Locally compute

$$[u] = \sum_{k=1}^{M} a_k \cdot [z_k] + \sum_{k=1}^{N} b_k \cdot [v_k]$$

$$[w] = \sum_{k=1}^{M} a_k \cdot [\alpha \cdot z_k] + \sum_{k=1}^{N} b_k \cdot [\alpha \cdot v_k]$$

- Open $[\alpha]$ to all parties
- Compute $[T] = [w] - \alpha \cdot [u]$.
- Generate a random shared value $[r]$ as before
- Compute $[Z] = [r] \cdot [T]$.
- Open $[Z]$, if $Z = 0$ the all is OK, otherwise abort.

Note that $T = 0$, but we multiply it by r before opening to ensure no information in T leaks.

Online vs Offline

The previous protocol was essentially twice as slow as the passive variant

- For each active multiplication you had to run two passively secure multiplication protocols.

Sometimes we are prepared to be slower overall, if we can make "online" computation faster.

This can be done within the above framework using the Beaver Triples from earlier.

An **Offline Phase** produces Beaver Triples

An **Online Phase** consumes them.

Online Method

The active protocol is the same as the one above except we now multiply using Beaver Triples

Assume we have sharings $\langle a \rangle, \langle b \rangle, \langle c \rangle$ which are produced in an actively secure manner.

We can then do the Beaver multiplication trick on the $\langle \cdot \rangle$ sharings

- All parties open $\langle \epsilon \rangle = \langle x \rangle - \langle a \rangle$ (Just the ϵ value not the MAC!)
- All parties open $\langle \delta \rangle = \langle y \rangle - \langle b \rangle$ (Just the δ value not the MAC!)
- Compute

$$\langle z \rangle = \langle c \rangle + \epsilon \cdot \langle b \rangle + \delta \cdot \langle a \rangle + \epsilon \cdot \delta.$$

Offline Method

Assume we have a method to produce passively secure multiplication triples wrt $\langle \cdot \rangle$ sharings.

We need to ensure these are (almost) actively secure

We do not at this stage care if the MACs are correct

- Since these are checked in the MAC Checking phase

We need to check that if we are given $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ that $c = a \cdot b$.

We do this by *sacrificing*

- We sacrifice one triple so as to check another

Sacrificing

Suppose we have two passively secure triples

$$(\langle a \rangle, \langle b \rangle, \langle c \rangle) \text{ and } (\langle f \rangle, \langle g \rangle, \langle h \rangle)$$

we want to confirm that $c = a \cdot b$

Pick a random value t known to all players, but controlled by none

Then the parties compute

- Open $\langle \rho \rangle = t \cdot \langle a \rangle - \langle f \rangle$ (Again just the ρ value not the MAC)
- Open $\langle \sigma \rangle = \langle b \rangle - \langle g \rangle$ (Again just the σ value not the MAC)
- Compute

$$\langle e \rangle = t \cdot \langle c \rangle - \langle h \rangle - \sigma \cdot \langle f \rangle - \rho \cdot \langle g \rangle - \sigma \cdot \rho.$$

- Open $\langle e \rangle$ (Again just the e value not the MAC)
- Reject if $e \neq 0$.

MAC Checking

A problem with the previous protocol (without Beaver triples) was that it was not *reactive*

When we executed MAC Checking we opened $\langle \alpha \rangle$.

This meant we could not continue with a computation after we had executed MAC Checking.

With Beaver multiplication this is much easier.

MAC Checking Beaver Multiplication

We simply have to check that every opened value has a valid MAC shared between all parties.

We no longer check multiplication is correct directly

- This is implied by the sacrificing and the MAC checking we are going to do

Note we will apply this MAC check even to the values opened during sacrificing

Suppose we have a set of open values $\{a_i\}_{i=1}^{M}$

We need to check that the parties hold correct shares of $[\gamma_i] = [\alpha \cdot a_i]$

MAC Checking Beaver Multiplication

MAC Checking then proceeds as follows:

- The parties agree on random values $b_1, \ldots, b_M \in \mathbb{F}_q$.
- Locally compute

$$a = \sum_{i=1}^{M} b_i \cdot a_i,$$

$$[\gamma] = \sum_{i=1}^{M} b_i \cdot [\gamma_i],$$

- Compute $[\omega] = [\gamma] - [\alpha] \cdot a$.
- The parties open the value $[\omega]$ (via a commitment protocol first to avoid rushing adversaries)
- If $\omega = 0$ then MAC Check passes

Trading IT Security For Speed

The above protocol made use of information theoretically secure primitives only

In practice we do not really care about this

We want SPEED

A major issue was generating random sharings of values.

- This required expensive interaction

We can however do this based on PRFs (block ciphers) non-interactively

- Have potential high cost of setting up the keys though

Pseudo-Random Secret Sharing: Shamir

A PRSS is a scheme for a bunch of parties to (without interaction) agree on a sharing of a random value.

We use a PRF with keyspace K, message space S and codomain $\mathbb{F}_q$,

$$\psi : \begin{cases} K \times S & \longrightarrow & \mathbb{F}_q \\ (k_A, a) & \longmapsto & \psi(k_A, a). \end{cases}$$

For every set $A \subset \{1, \ldots, n\} = \mathcal{P}$ of size $n - t$ define a polynomial of degree t s.t.

$$f_A(0) = 1 \text{ and } f_A(i) = 0 \text{ for all } i \in \mathcal{P} \setminus A.$$

Also for each set A define a key k_A to the PRF and give k_A to every party in A.

Pseudo-Random Secret Sharing: Shamir

To generate a new Shamir sharing the parties select the "next" value a in S which is unused and compute

$$s_i \leftarrow \sum_{i \in A} \psi(k_A, a) \cdot f_A(i).$$

The underlying polynomial is hence given by

$$f(X) = \sum_{i \in A} \psi(k_A, a) \cdot f_A(X)$$

due to the choice of the $f_A(X)$.

The underlying secret is

$$s = \sum_{A} \psi(k_A, a) \cdot f_A(0) = \sum_{A} \psi(k_A, a).$$

Pseudo-Random Secret Sharing: General

This generalised to any LSSS as follows

For every set A which is a complement of a maximally unqualified set we find a sharing $\mathbf{s}_A$ of the value of **one** such that

- The sharing has zero contribution from all parties in $\mathcal{P} \setminus A$.

This is always possible

$\mathbf{s}_A$ is the analogue of the Shamir sharing $(f_A(1), \ldots, f_A(n))$ above.

For every set A we also distribute a key k_A to every player in A.

- This is the expensive bit, as the number of keys *could* be exponential depending on the access structure of the LSSS.

Pseudo-Random Secret Sharing: General

To form a random sharing of a new value we compute the local shares for player i as

$$\mathbf{s}_i \leftarrow \sum_{i \in A} \psi(k_A, a) \cdot \mathbf{s}_A|_i.$$

where $\mathbf{s}_A|_i$ denotes the components in $\mathbf{s}_A$ which belong to player i.

The underlying secret is

$$s = \sum_A \psi(k_A, a) \cdot 1 = \sum_A \psi(k_A, a),$$

since

$$[s] = \sum_A \psi(k_A, a) \cdot [s_A] = [\sum_A \psi(k_A, a)].$$

Using Any Secret Sharing Scheme

The previous MPC protocol (Beaver+Reactive MACCheck) can be applied to *any* secret sharing scheme

As long as we can produce the Beaver Triples

In particular we can do this for *full threshold* LSSS

To produce the Beaver triples we use Somewhat Homomorphic Encryption (SHE).

This is called the SPDZ protocol

- **Multiparty Computation from Somewhat Homomorphic Encryption**
- I. Damgard and V. Pastro and N.P. Smart and S. Zakarias
- `https://eprint.iacr.org/2011/535`

SPDZ 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

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

We write this additive sharing as $[\cdot]$.

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

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

We write $\langle x \rangle = ([x], [\alpha \cdot x])$ as before

In what follows "partially opening" a share $\langle x \rangle$ means revealing x_i but not the MAC share.

Secret Sharing

Note we can share a public constant v by

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

Addition

Suppose we have two shared values $\langle x \rangle$ and $\langle y \rangle$.

To compute the result $\langle z \rangle$ of an addition gate the parties individually execute

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

Multiplication

To multiply $\langle x \rangle$ and $\langle y \rangle$ to obtain $\langle z \rangle$ we work as follows:

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

$$\langle z \rangle = \langle c \rangle + \epsilon \cdot \langle b \rangle + \rho \cdot \langle a \rangle + \epsilon \cdot \rho.$$

This is the same Beaver trick as before.

We verify correctness as before using MAC Check.

Preprocessing – SPDZ via SHE

We return to the preprocessing, which we do for large finite fields using SHE

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

We assume an SHE scheme with keys $(\mathsf{pk}, \mathsf{sk})$ whose plaintext is $\mathbb{F}_p$

- In practice for efficiency work on vectors of such elements in a SIMD fashion

Given $\mathsf{ct}_1 = \mathsf{Enc}_{\mathsf{pk}}(m_1)$ and $\mathsf{ct}_2 = \mathsf{Enc}_{\mathsf{pk}}(m_2)$ we have

$$\mathsf{Dec}_{\mathsf{sk}}(\mathsf{ct}_1 + \mathsf{ct}_2) = m_1 + m_2$$

and

$$\mathsf{Dec}_{\mathsf{sk}}(\mathsf{ct}_1 \cdot \mathsf{ct}_2) = m_1 \cdot m_2.$$

We only need to evaluate circuits of multiplicative depth one.

Preprocessing – SPDZ via SHE

We require a little more of our SHE scheme though

We assume a shared SHE public key pk for an SHE scheme.

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

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

Preprocessing – SPDZ via SHE: 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 $\text{ct}_{f_i} = \text{Enc}_{\text{pk}}(f_i)$.
- All compute $\text{ct}_{m+f} = \text{ct} + \sum \text{ct}_{f_i}$.
- Execute $\text{Dec}_{\text{sk}_1,\ldots,\text{sk}_n}(\text{ct}_{m+f})$ to obtain $m + f$.
- Party 1 sets $m_1 = (m + f) - f_1$.
- Party $i \neq 1$ sets $m_i = -f_i$.
- Set $\text{ct}' = \text{Enc}_{\text{pk}}(m + f) - \sum \text{ct}_{f_i}$.

Use some "default" randomness for the last encryption.

Preprocessing – SPDZ via SHE: Generating $\langle a \rangle$ and $\langle b \rangle$

We can generate our sharing $\langle a \rangle$ 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 $\langle b \rangle$.

Preprocessing – SPDZ via SHE: Generating $\langle c \rangle$

This is also easy

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

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

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

Course Summary

We have introduced MPC

We have seen how MPC can be done via FHE, but this is slow

We have seen how two party MPC can be done using Garbled Circuits

We have seen how many party MPC can be done via Linear Secret Sharing Schemes