We imagine n untrusting entities.
We'll choose a number t, where $1 \leq t \leq n$, as the "threshold".
Each participant from 1 to n has their own polynomial; they keep the coefficients a_{ij} secret:

$$f_1(x) = a_{1,0} + a_{1,1}x + a_{1,2}x^2 + \ldots + a_{1,t-1}x^{t-1}$$
$$\ldots$$
$$f_n(x) = a_{n,0} + a_{n,1}x + a_{n,2}x^2 + \ldots + a_{n,t-1}x^{t-1}$$

Notice that t points on the curve of each polynomial $f_k(x)$ will always be enough to recover the polynomial in total.
(As an example that's easy to keep in your head: if the curve is a parabola, two points are not enough to define it, but 3 are (here, $t = 3$ and $t - 1 = 2$ so x^2 is the highest power of the polynomial).)

The key insight of the Feldman "Verifiable Secret Sharing Design" is that you can exploit the linearity of polynomials such that an *overall* secret can be recovered using Lagrange interpolation, by combining the process for each participant. Thus, consider the sum of the above:

$$F(x) = \sum_{k=1}^{n} f_k(x)$$

The shared secret will be the evaluation of F at 0, i.e. $\sum_k a_{k,0}$. To recover it, you would need t evaluations of F, which would be deducible from t evaluations of *each* f_k (the linearity; just add them).

This is a good time to note that **that never happens** in these schemes. Unlike a system like Shamir's secret sharing, this represents a "distributed key generation" in which the overall "shared secret" is never reconstructed in one place.

This extremely powerful property extends further, in schemes like FROST: it's now possible to construct a proof of knowledge, and therefore a signature, using that "shared secret", by a distributed process between the untrusted parties, where they do *not* reveal their "secret shares".

So, the "private key" in this construct is $F(0)$, hence the public key is $Y = F(0)G$.

In a moment, we'll talk about the "public key sharing" process, but first, let's see concretely how the individual participants' secret shares relate to the overall secret (which as we said - nobody ever sees!):

Each participant has a number assigned, from 1 to t.
This is the tricky part to understand, so I'll use a concrete example.
Let's suppose that $n = 8$; each participant in setup is assigned a number $(1 \ldots 8)$, let's suppose we are participant 3. Suppose $t = 4$, and it's participants 2,4,7 and us (3) that want to do the collaborative signing.
We will have our own polynomial, $f_3(x)$, and as earlier noted, we will have

chosen the $a_{3,j}$ coefficients at random. So we'll know $f_3(3)$. Now, the other participants will give us $f_i(3)$ for $i = 2, 4, 7$ (we'll talk about how this part can be done safely below; for now, let's just trust that we are given correct values). The sum, $f_2(3) + f_3(3) + f_4(3) + f_7(3)$, is our "secret share". The other 3 participants will find their share analogously. Notice that this secret share is actually $F(3)$.
With 4 of those evaluations of F, you would be able to reconstruct F, and more specifically, we could calculate $F(0)$ using Lagrange interpolation:

$$\begin{aligned} F(0) = F(2) \times \left(\frac{(0-3) \times (0-4) \times (0-7)}{(2-3)(2-4)(2-7)} \right) \\ + F(3) \times \left(\frac{(0-2) \times (0-4) \times (0-7)}{(3-2)(3-4)(3-7)} \right) \\ + F(4) \times \left(\frac{(0-2) \times (0-3) \times (0-7)}{(4-2)(4-3)(4-7)} \right) \\ + F(7) \times \left(\frac{(0-2) \times (0-3) \times (0-4)}{(7-2)(7-3)(7-4)} \right) \end{aligned}$$

While the Lagrange formula is complicated, or at least, big, notice that it is a very trivial arithmetic calculation.
Notice that it would work just as well with *any* four participants, because each participant essentially "owns" an evaluation of F at a particular point, and we only need 4 of them. If we have 5 or more, it still works.

But how do we sign without reconstructing the secret?

Recall the formula for a Schnorr signature on a private key x:
$s = k + ex$
... where k is a one time (secret) nonce, and e is the hash of the message. Recall also that, because the equation is linear, then a *naive* aggregation is possible (if, usually, unsafe):

$s_1 + s_2 = (k_1 + k_2) + e(x_1 + x_2)$

Now, as we have just established:

$$F(0) = \lambda_2 F(2) + \lambda_3 F(3) + \lambda_4 F(4) + \lambda_7 F(7)$$

where λ_i is the parenthetical term used in the above concrete calculation; it's a function of *only* the participants' indices (e.g. for participant 2, λ_2 is just $(-3 \times -4 \times -7)/((2-3)(2-4)(2-7))$).

... and so, if that is the signing secret, then what we need overall is:

$s = k + eF(0) = k + e\left(\lambda_2 F(2) + \lambda_3 F(3) + \lambda_4 F(4) + \lambda_7 F(7)\right)$

... which naturally leads to the idea that each participant should provide this *partial signature share*:
$s_i = k_i + e\lambda_i F(i)$

Adding these together will give a valid overall signature on the secret $F(0)$, without any participant ever seeing $F(0)$, as long as $k = \sum_i k_i$, **and** we have addressed the cryptographic security of combining the values of k in a way which doesn't allow an adversarial participant to forge a signature or steal our secret. This point is addressed in detail in both the FROST and MuSig(2) papers, in detail (the solutions are basically the same, in each case. It would be a sidetrack to go further into that, here).

Backing up: how do we share polynomial evaluations?

Remember that one of the first steps, is each participant giving evaluations of *their* polynomials, to the others. So in our concrete example, as participant 3, we need to be given $f_2(3)$ and $f_4(3)$ and $f_7(3)$. These will just be (32 byte in typical cases) random numbers. Since the other guys' polynomials are secret, how do we know these values are correct? Here, we use the homomorphism of the elliptic curve. We do something that's sometimes described as "lifting the polynomial into the exponent" (it used to make more sense when we used finite fields instead of elliptic curves, but, whatever).

Consider that if you have

$$f_1(x) = a_{1,0} + a_{1,1}x + a_{1,2}x^2 + \ldots + a_{1,t-1}x^{t-1}$$

then

$$f_1(x)G = a_{1,0}G + a_{1,1}xG + a_{1,2}x^2G + \ldots + a_{1,t-1}x^{t-1}G$$

where we're using all the scalar values as multipliers for the generator point G of the elliptic curve. Because polynomials have linearity, all calculations on the scalars also work with curve points, with the crucial difference that you don't actually *reveal* the scalars. This allows the untrusting participants to prove to each other that they are evaluating the polynomials honestly without screwing up the security of the scheme by revealing their own polynomial coefficients.
To make this work, each participant gives all other participants, **commitments** to every one of their coefficients, like this:
$C_{4,0} = a_{4,0}G, \quad C_{4,1} = a_{4,1}G \ldots C_{4,t-1} = a_{4,t-1}G$

You'll notice there'll be a total of $n \times t$ such commitments; they are all points on the curve. Once everyone has all of these, and once we've shared the evaluations (like, for example participant 4 gives us, participant 3, $f_4(3)$), we can now use these commitments to check correctness:

$$f_4(3)G =? C_{4,0} + 3C_{4,1} + 3^2C_{4,2} + \ldots 3^3C_{4,3}$$

Notice what's going on here is: we have a polynomial f_4 which, because t is 4, is a cubic polynomial. We're evaluating it at $x = 3$, but *we're doing the evaluation of the polynomial entirely hidden behind the points on the curve.* A good mental model is that this is an **encrypted polynomial evaluation**. If it matches, we know that we have been given a valid $f_4(3)$ and the same for all the other indices.