Shared Secrets and Threshold Signatures

Reference Document

Michaella Pettit

Copyright

Contents

1 Introduction

In the following document we describe how to create and verify a shared secret between N participants without a single entity knowing the secret. We show how to calculate the addition and multiplication of two shared secrets, and how to calculate shares of the inverse of a shared secret. We begin by describing these concepts for a general number of participants and then go through a worked example with just three participants to illustrate the ideas in a more digestible way.

We then explore a specific use case: a threshold signature scheme in which participants must cooperate to create a signature on a message. Again, we describe the signature scheme for a general number of participants and then go through a worked example.

The threshold signature scheme described below allows digital signatures to be created without the private key ever coming into existence. It is so-called 'threshold non-optimal' meaning that the number of shares required to recreate the shared secret is less than the number of shares required to create a signature. However, this can be useful if one of the shares is lost as the shared key is still recoverable. It is also computationally fast and practical to implement.

The scheme exploits some useful mathematical properties of polynomials. One key fact that is used is there is a unique polynomial curve of order t that passes through $(t+1)$ points. This means that given any selection of $(t+1)$ points on a polynomial curve of order t it is possible to identify the underlying polynomial, and in turn identify a unique shared secret.

The polynomials we work with are over the finite field $\mathbb{Z}_n$ where n is prime and is related to the order of an elliptic curve group.

Elliptic curve groups

An elliptic curve E satisfies the equation

$$y^2 = x^3 + ax + b \bmod p\ ,$$

where $a, b \in \mathbb{Z}_p$ and a, b are constants satisfying $4a^3 + 27b^3 \neq 0$. We only consider elliptic curves where p is a prime. The group over this elliptic curve is then defined to be the set of elements (x, y) satisfying this equation along with the point at infinity $\mathcal{O}$, which is the identity element. The group operation on the elements in this group is called elliptic curve point addition and denoted by $+$. We denote this group by $E(\mathbb{Z}_p)$ and its order by n.

This group operation can be used to define another operation on the elements called point multiplication denoted by $\cdot$. For a point $G \in E(\mathbb{Z}_p)$ and a scalar $k \in \mathbb{Z}_n^*$, the point $k \cdot G$ is defined to be the point G added to itself k times. The explicit form of point addition is not necessary here, so we refer the reader to one of the many elliptic curve group resources for more information if required.

In Elliptic Curve Cryptography a private key is defined to be a scalar $k \in \mathbb{Z}_n^*$, and the corresponding public key is the point $k \cdot G$ on an elliptic curve. In Bitcoin, this was chosen to be the secp256k1 elliptic curve, and the values $a, b,$ and p are completely specified by this curve. The order n of this group has been calculated given these values, which in the case of this curve is a prime, and the secp256k1 standard specifies a point G which is to be used as the generator of this group in Bitcoin. Given a point $k \cdot G$ on the secp256k1 curve, it is computationally infeasible to calculate k, which is one of the reasons behind why Bitcoin is secure.

2 Secret sharing

In this section we describe how to create a shared secret and verify that all participants are sharing the correct information in the set-up. This is conventionally known as Joint Verifiable Random Secret Sharing. We then show how to calculate the addition and product of two shared, and then how to calculate the shares of the inverse of a shared secret.

2.1 Joint Verifiable Random Secret Sharing

To create a shared secret between N participants a method called Joint Verifiable Random Secret Sharing (JVRSS) is used, introduced by Pedersen [1]. Assume that N participants want to create a joint secret that can only be regenerated by at least $(t+1)$ of the participants in the scheme. This restricts $(t+1) \leq N$ as otherwise it will not be possible to calculate the shared secret. Any attempts to regenerate the secret by less than $(t+1)$ participants will be unsuccessful. To create the shared secret, the following steps are taken.

1. The participants agree on the unique label i for each participant. Each participant i generates $t+1$ random numbers
$$a_{ij} \in_R \mathbb{Z}_n^*, \ \forall j = 0, \dots, t \ ,$$
where $\in_R$ means a randomly generated element of the field $\mathbb{Z}_n$. Then each participant has a secret polynomial of order t
$$f_i(x) = a_{i0} + a_{i1}x + \cdots + a_{it}x^t \bmod n \ ,$$
for $i = 1, \dots, N$. The N participants of the group use these polynomials to generate a shared secret polynomial, where the sum of the zeroth-order of the private polynomials a_{i0} is the shared secret. Note that we omit the $\bmod n$ notation from now on, and it is assumed that all arithmetic operations over integers are done modulo n.
2. Each participant i sends the value $f_i(j)$ to participant j using a secure communication channel with participant j only. It is crucial that $f_i(j)$ is shared only with participant j. If the value $f_i(j)$ is shared with all participants for more than t different values of j, then it is possible for all participants to calculate the private polynomial of participant i. If all private polynomials are able to be calculated, then the shared secret polynomial is all calculable.
3. Each participant i calculates their own private secret share of a shared secret polynomial as
$$a_i := \sum_{j=1}^{N} f_j(i) \ .$$
These secret shares are y-values on the shared polynomial. Essentially, by defining the shares in this way, the shared polynomial is being defined as
$$f(x) := \sum_{i=1}^{N} f_i(x) \ .$$
Of course, no participant knows this polynomial explicitly.
The shared secret a is then the zeroth-order of the shared polynomial
$$a := f(0) = \sum_{i=1}^{N} f_i(0) = \sum_{i=1}^{N} a_{i0} \ ,$$
that is, the sum of the zeroth-order terms of the private polynomials of the participants. Again, no single participant knows this shared polynomial or secret; they only know their own share a_i.

Now that the participants have generated a shared polynomial, they can each verify that the other participants have shared the correct information to all participants, and that all participants have the same shared polynomial. This is done in the following way.

4. Each participant i broadcasts to all participants the obfuscated coefficients

$$a_{ik} \cdot G \ ,$$

for $k = 0, \dots, t$. It is important to broadcast this so that each participant knows that other participants got the same information.

5. Each participant i can check that each participant j has correctly calculated the polynomial point $f_j(i)$. This is done by calculating the public key corresponding to the polynomial point of participant j and compare this to the obfuscated coefficients $a_{jk} \cdot G$ where $k = 0, \dots, t$. That is, participant i calculates $f_j(i) \cdot G$ and checks that

$$f_j(i) \cdot G \stackrel{?}{=} \sum_{k=0}^{t} i^k \left(a_{jk} \cdot G\right) \quad \forall\, j = 1, \dots, N \ .$$

If any of the participants find that this equality doesn't hold, they raise an issue and the adversary can be removed from the group. On the other hand, if all participants find that this equation holds for each polynomial, then these obfuscated coefficients necessarily correspond to the private polynomial point shared by a participant. The group collectively can then be sure that they have all created the same shared polynomial.

This completes the derivation and verification of shared secret shares. Note that the shared secret shares correspond to the y-value of a point on a shared polynomial, and a participant's index i corresponds to the x-value at that point. That is, a shared secret share is a point with the form (i, a_i), where i is the participants label in the scheme. As mentioned, it is important that all participants agree on the label of each participant.

We denote the method described in steps 1-5 by $a_i = JVRSS(i)$ for participant i.

Reconstructing a shared secret: Lagrange Interpolation

In order to reconstruct a shared secret Lagrange interpolation is used. Lagrange interpolation is a method describing how to calculate an order-t polynomial, given $(t+1)$ points on that polynomial.

Assume we want to reconstruct an order-t polynomial $f(x)$. One requires knowledge of at least $(t+1)$ points on this polynomial to reconstruct it. If we label these points by

$$(x_1, y_1), \dots, (x_{t+1}, y_{t+1}) \ ,$$

then to find the polynomial, one calculates

$$f(x) = \left(\sum_{l=1}^{t+1} y_l \prod_{\substack{1 \le j \le (t+1), \\ j \neq l}} (x - x_j)(x_l - x_j)^{-1} \right).$$

Note that with less than $(t+1)$ points it is not possible to find a unique polynomial. If one interpolates over more than $(t+1)$ shares, the result is the same as interpolating over exactly $(t+1)$ shares. However, if one interpolates over more than $(t+1)$ shares, and there is a mistake in any of the shares, the interpolation result will be wrong, even if $(t+1)$ of the shares are correct.

In order to find the zeroth order coefficient of the polynomial, the equation is simplified to

$$f(0) = \left(\sum_{l=1}^{t+1} y_l \prod_{\substack{1 \le j \le (t+1), \\ j \neq l}} (-x_j)(x_l - x_j)^{-1} \right).$$

Recall that the shared secret is defined as the zeroth order coefficient of the polynomial, and so this equation calculates the shared secret.

We denote interpolation at 0 by $interpolate(a_1, \dots, a_{t+1})$, where a_i is the secret share of participant i for $i = 1, \dots, (t+1)$. That is, we define

$$interpolate(a_1, \dots, a_{t+1}) := \left(\sum_{i=1}^{t+1} a_i \prod_{\substack{1 \le j \le (t+1), \\ j \neq i}} (-j)\,(i-j)^{-1} \right).$$

Notice that this is interpolation evaluated at zero, such that the result of the calculation is the shared secret that has been shared using JVRSS.

We now give two methods that allow us to calculate the shared public key corresponding to a shared secret that has been calculated with JVRSS.

JVRSS public key calculation method 1: Elliptic curve interpolation

If one would like to calculate a public key corresponding to a shared secret, it can be calculated using elliptic curve (EC) Lagrange interpolation. In this case, the points are of the form

$$(x_1, y_1 \cdot G), \dots, (x_{t+1}, y_{t+1} \cdot G),$$

where $y_1, \dots, y_{t+1}$ are the shares on the shared secret polynomial, as before, and $y_1 \cdot G, \dots, y_{t+1} \cdot G$ are the corresponding public keys. Then the public key of the shared polynomial is reconstructed using

$$f(x) \cdot G = \sum_{l=1}^{t+1} y_l \cdot G \prod_{\substack{1 \le j \le (t+1), \\ j \neq l}} (x - x_j)(x_l - x_j)^{-1},$$

where the summation is elliptic curve point addition and the product is integer multiplication modulo n. Alternatively, to immediately find the shared secret, the equation simplifies to

$$f(0) \cdot G = \sum_{l=1}^{t+1} y_l \cdot G \prod_{\substack{1 \le j \le (t+1), \\ j \neq l}} (-x_j)(x_l - x_j)^{-1}.$$

This is the usual way to calculate a public key corresponding to a shared secret and can be executed for any shared values in the form of an EC point. However, if JVRSS has been executed, we can use information from this to calculate a public key corresponding to a shared secret in a more efficient way. We describe this alternative method.

JVRSS public key calculation method 2: Simple point addition of the zeroth order coefficients

In step 4 of JVRSS the public keys corresponding to private polynomial coefficients are shared. In order to calculate the public key corresponding to a shared secret, one can simply add the zeroth-order private polynomial coefficient public keys, and this gives the public key corresponding to the shared secret. That is, given the N zeroth-order private polynomial coefficient public keys $a_{i0} \cdot G$ for $i = 1, \dots, N$, the equation for the shared public key is

$$a \cdot G = \sum_{i=1}^{N} a_{i0} \cdot G,$$

where $a \cdot G$ is the public key corresponding to the shared secret a. By calculating the public key corresponding to a shared secret by adding the zeroth-order coefficients explicitly, there are exactly $N - 1$ point additions to calculate.

This summation is equivalent to interpolation over the elliptic curve points $a_i \cdot G$, where the shared secret shares are a_i. We can attempt to compare this summation to the number of additions in EC interpolation given by the equation for $f(0) \cdot G$ above.

We see immediately that there are t point additions that must be done with the overall summation. However, because of the product coefficient for each term, there are more additions that must be done corresponding to scalar point multiplication. For each $y_l \cdot G$, the coefficient is

$$c_l = \prod_{\substack{1 \le j \le (t+1), \\ j \neq l}} (-x_j)(x_l - x_j)^{-1} .$$

It is very difficult to simplify these coefficients without knowing or making assumptions about the explicit values of x_j, x_l. Note that some of these will be negative coefficients, or equivalently $(n - c_l) \bmod n$, which we can assume to be large, as we choose $x_k = k \ll n$.

Additionally, in the case of the shared public key calculation, each $y_l \cdot G$ needs to initially be calculated, since $a_i \cdot G$ is not shared explicitly. This is done by calculating

$$a_i \cdot G = \sum_{j=1}^{N} f_j(i) \cdot G ,$$

for each $i = 1, \dots, N$. This implies that in total at least

$$(N-1)Nt$$

point additions must be calculated in EC interpolation to calculate the public key of a shared secret given JVRSS. In addition to this, the modular inverse is costly, and is calculated $t(t+1)$ times in an interpolation. Since each participant does these calculations, the group as a whole does this number of calculations N times.

With all this in mind, method 2 would generally be the best method of calculating a shared public key. However, there may be some cases, such as when $y_l \cdot G$ is known and $N \gg t$, where elliptic curve interpolation could be preferable.

2.2 Addition of shared secrets

We describe how to calculate the addition of two shared secrets that are shared amongst a group of N participants, where each secret polynomial has order t. This method allows for the calculation of the addition of two shared secrets by the group without a single entity knowing either of the shared secrets. The general process for this is as follows.

1. Generate the first shared secret a, where participant i's share is given by $a_i = JVRSS(i)$ for $i = 1, \dots, N$ where N is the number of participants in the scheme. The shared secret polynomial has order t, meaning $(t+1)$ participants are required to recreate it.
2. Generate the second shared secret b, where participant i's share is given by $b_i = JVRSS(i)$, and the shared secret polynomial again has order t.
3. Each participant calculates their own additive share
$$v_i = a_i + b_i \bmod n .$$
4. All participants broadcast their additive share v_i to all other participants. It is safe for these shares to be shared publicly.
5. Each participant interpolates over at least $(t+1)$ of the shares v_i to calculate
$$v = interpolate(v_1, \dots, v_{t+1}) = a + b .$$
Note that interpolation can be done over more than $(t+1)$ shares, and the result will still be the same provided all shares in the interpolation are correct.

We denote the steps in addition of shared secrets for participant i by $ADDSS(i)$ which results in each participant i knowing $v = (a + b)$.

2.3 Product of shared secrets

We describe how to calculate the product of two shared secrets that are both shared amongst a group of N participants, where each secret polynomial has order t. This method allows for the calculation of the product of two shared secrets by the group without a single entity knowing either of the shared secrets. The general process for this is as follows.

1. Generate the first shared secret a, where participant i's share is given by $a_i = JVRSS(i)$ for $i = 1, \dots, N$ where N is the number of participants in the scheme. The shared secret polynomial has order t, meaning $(t + 1)$ participants are required to recreate it.
2. Generate the second shared secret b, where participant i's share is given by $b_i = JVRSS(i)$, and the shared secret polynomial again has order t.
3. Each participant calculates their own multiplicative share μ_i using
$$\mu_i = a_i b_i \ .$$
These are the y-values on a shared polynomial of order $2t$ at $x = i$, and this polynomial necessarily crosses the y-axis at ab. Since the polynomial has order $2t$, at least $2t + 1$ participants are required to calculate the multiplicative value at the point $x = 0$, which results in ab.
4. All participants broadcast their multiplicative share μ_i to all other participants. It is safe for these shares to be shared publicly.
5. Each participant interpolates over at least $(2t + 1)$ of the shares μ_i at 0 to calculate
$$\mu = interpolate(\mu_1, \dots, \mu_{2t+1}) = ab \ .$$
Again, interpolation can be done over more than $(2t + 1)$ shares, and the result will still be the same provided all shares in the interpolation are correct.

We denote the steps for participant i in the product of shared secrets by $\mu = ab = PROSS(i)$.

2.4 Inverse of a shared secret

If participants would like to generate inverse shares of a shared secret a^{-1} without a single entity knowing the shared secret, the following process must be done. First notice that the interpolation of the inverse shares is not the same as the inverse of the interpolation of the shares, that is,

$$a^{-1} = (interpolate(a_1, \dots, a_{t+1}))^{-1} \neq interpolate(a_1^{-1}, \dots, a_{t+1}^{-1}) \ ,$$

which can be checked with a simple example. This implies that the shared secret a must somehow be calculated in full, without calculating it explicitly, before calculating its inverse. We saw in the previous sections that this can be done without knowing the result a by blinding it with another shared secret b. Note the blinding value must be a shared secret as well, otherwise the secret a can be calculated.

In order to calculate the inverse of a shared secret a, the following steps must be taken.

1. Calculate the product of shared secrets as in the previous section. That is, participant i executes $PROSS(i)$, the result of which is $\mu = ab \bmod n$.
2. Calculate the modular inverse of μ which results in
$$\mu^{-1} = (ab)^{-1} \bmod n \ .$$
3. Each participant i calculates their own inverse secret share by calculating
$$a_i^{-1} = \mu^{-1} b_i \ .$$

This last equation is equivalent to $a^{-1}b^{-1}\ b_i \bmod n$ if one substitutes in for μ. One can then use these inverse secret shares in a calculation with another shared secret, where the inverse of a is required. Interpolating over this will result in a calculation using a^{-1} without ever explicitly knowing a^{-1}.

We denote the steps in the inverse of shared secrets for participant i by $a_i^{-1} = INVSS(i)$.

This completes the section on the general JVRSS. In the next section, we give simple examples of the methods described above.

3 Secret sharing worked example

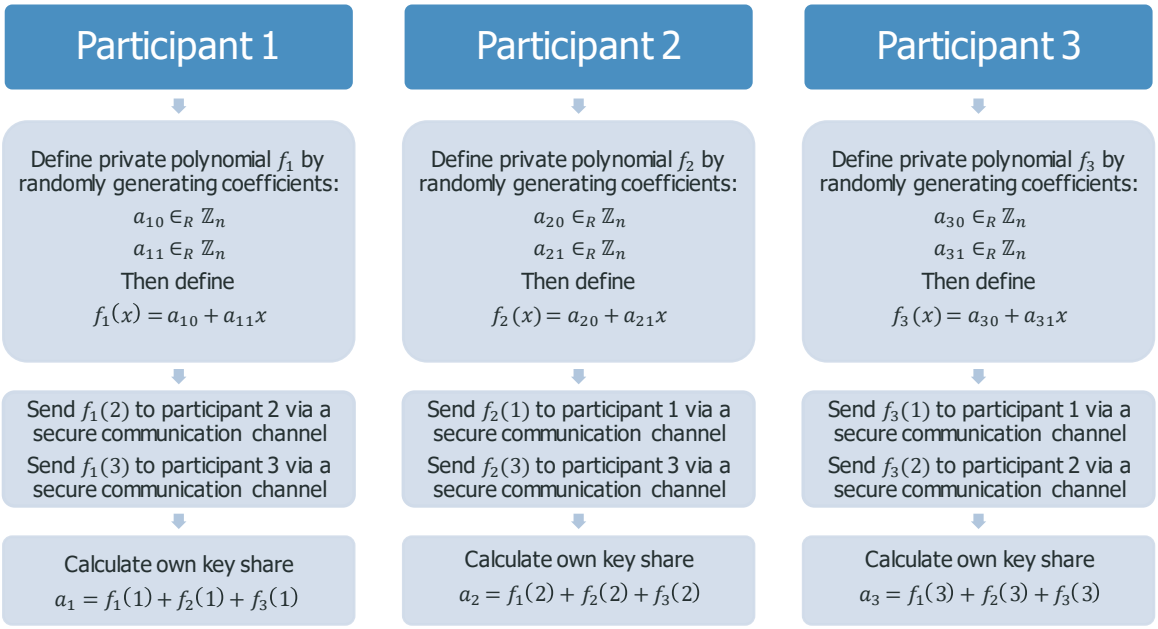
In this section we do worked examples to illustrate the ideas of the previous section. We begin by looking a JVRSS scheme with 3 participants, 2 of which are required to recreate the secret, that is, we take $N = 3$ and $t = 1$ in the previous section.

3.1 Joint Verifiable Random Secret Sharing

We describe a simple example of JVRSS. First, all the participants generate their share and then verify the shared secret. We begin with the generation of the shared secret shares.

Generation of secret shares

In order to generate shared secret shares the participants take the following steps.



These steps correspond exactly to step 1-3 in the general JVRSS scheme described in section 2.1.

The key shares for each participant $i = 1, 2, 3$ are

$$a_i = f_1(i) + f_2(i) + f_3(i) ,$$

which are shares on the shared secret polynomial

$$f_{(a)}(i) = (a_{10} + a_{20} + a_{30}) + i(a_{11} + a_{21} + a_{31}) ,$$

where the subscript (a) denotes that the polynomial corresponds to the shared secret a which is defined to be

$$a := (a_{10} + a_{20} + a_{30}) .$$

In step 2 it is crucial that the point on each private polynomial for participant i is shared only with participant j. If all the polynomial points are shared then every private polynomial can be calculated, and therefore the shared secret. For example, if participant 1 shares the values $f_1(2)$ and $f_2(3)$ with everyone, then all participants know two shares of an order-1 polynomial. Therefore, anyone can calculate the zeroth order of the polynomial using interpolation. This process could

then be repeated for all private polynomials, and therefore everyone can calculate the shared secret $a = \sum_i a_{i0}$.

We can represent this process of generating the secret using a graph. In the graph below, we plot each of the three private polynomials $f_i(x)$ and the shared secret polynomial $f_{(a)}(x)$.

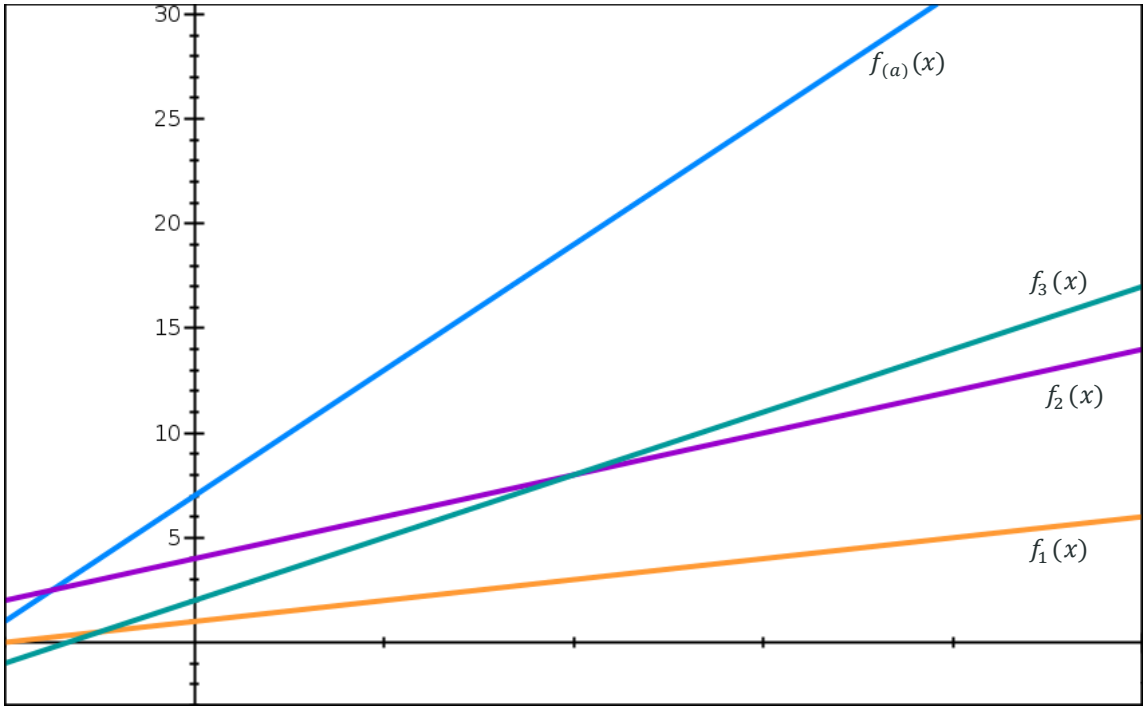


Figure 1: A graphical representation of calculating a shared polynomial $f_{(a)}(x)$ using JVRSS.[1]

The purple, teal, and orange line each represent a private polynomial corresponding each of the participants. The blue polynomial is the summation of the three polynomials, resulting in the shared polynomial, and the shared secret a is where this shared polynomial crosses the y-axis. We introduce the subscript in brackets notation to distinguish between shared polynomials for different secrets. In this figure, the blue polynomial corresponds to the polynomial of the shared secret a.

Each participant i calculates their point on the shared polynomial, by adding their share of each of the private polynomials. Two of these shares can be used to find the shared secret polynomial.

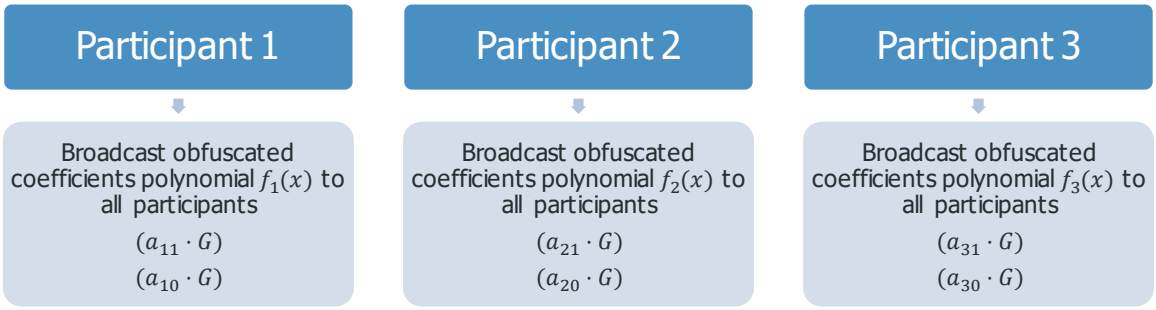
The graph above represents polynomials over the real numbers. However, we are using polynomials are over finite fields, that is, modulo some integer. Polynomials modulo an integer are simply points on a graph rather than lines as shown above. We give the polynomials over the real numbers as above for illustrative purposes, since the principles are the same.

We now describe the steps taken to verify the secret shares.

Verification of key shares

Before any verification can take place, each participant must first share the following obfuscated coefficients of their private polynomials.

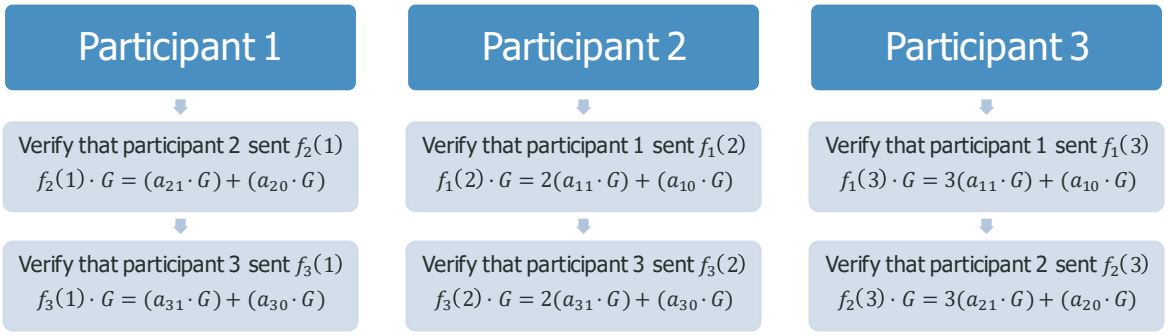
[1] This and all graphs to follow were created using the website graphsketch.com.



This step corresponds to step 4 in the JVRSS described in section 2.1. Once this information is shared, each participant can individually verify the information given to them in step 2 of JVRSS using these obfuscated coefficients.

Verification of polynomial points

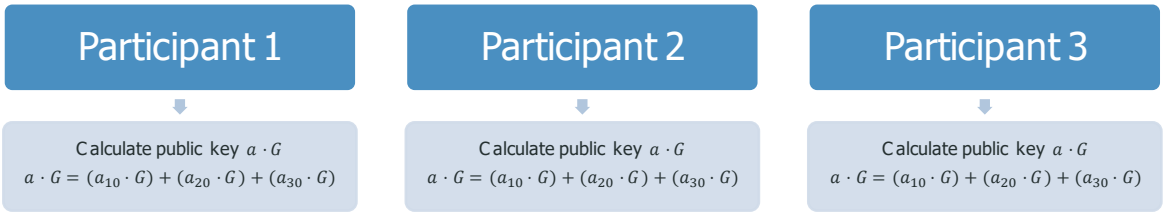
Each participant verifies the shares received from the other participants in the following way.



These steps correspond to step 5 in the general JVRSS scheme described in section 2.1.

Calculate shared public key

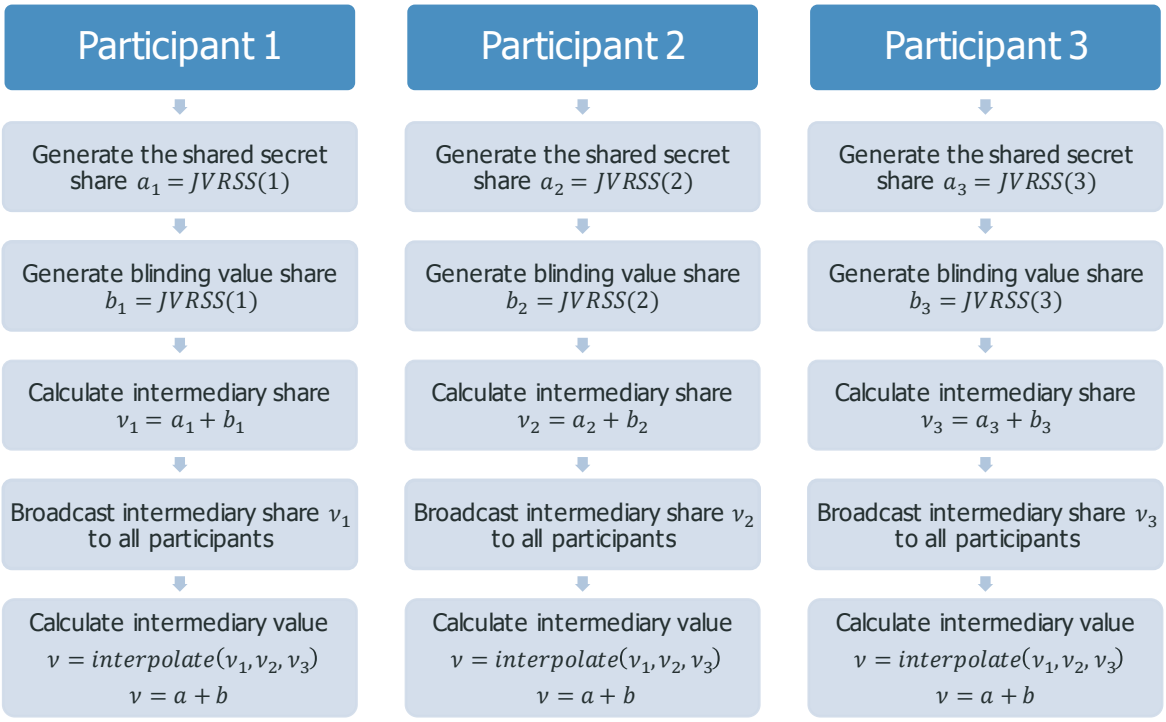
The final step in the generation of the shared secret is to calculate the corresponding public key.



Each participant i can store this public key $a \cdot G$ with their secret share a_i.

3.2 Addition of shared secrets

We describe the steps for calculating the addition of shared secrets.



These steps correspond exactly to steps 1-5 in the protocol describing the addition of shared secrets in section 2.2.

This can be shown graphically.

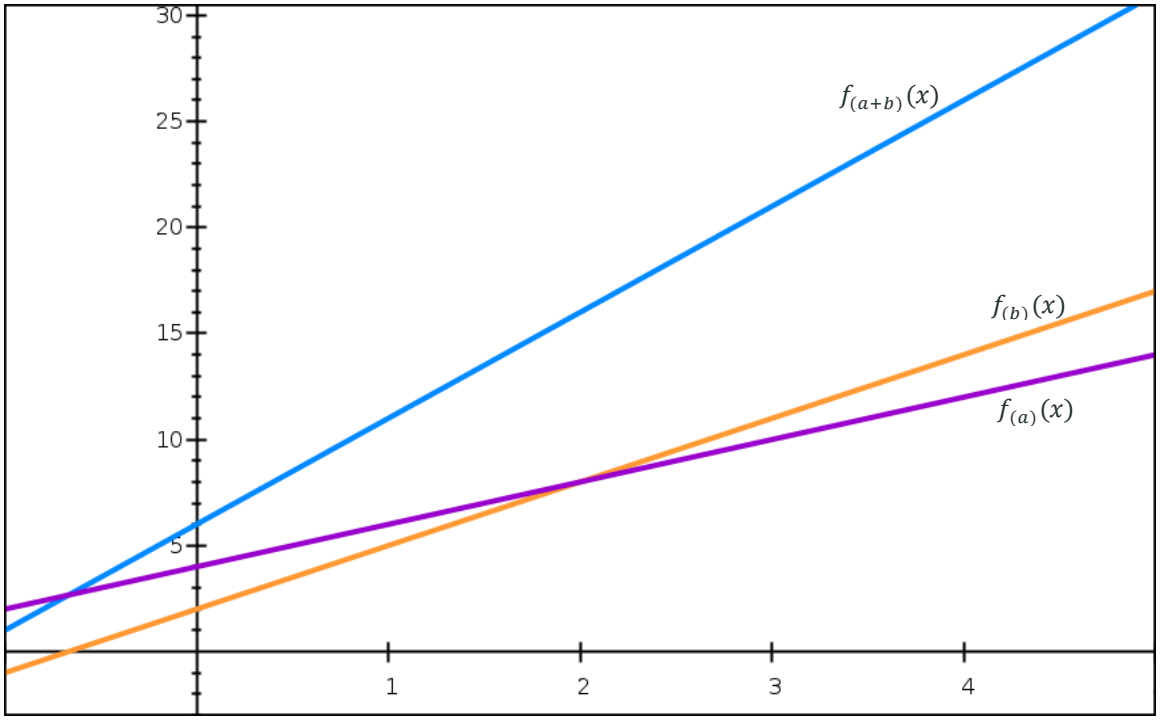
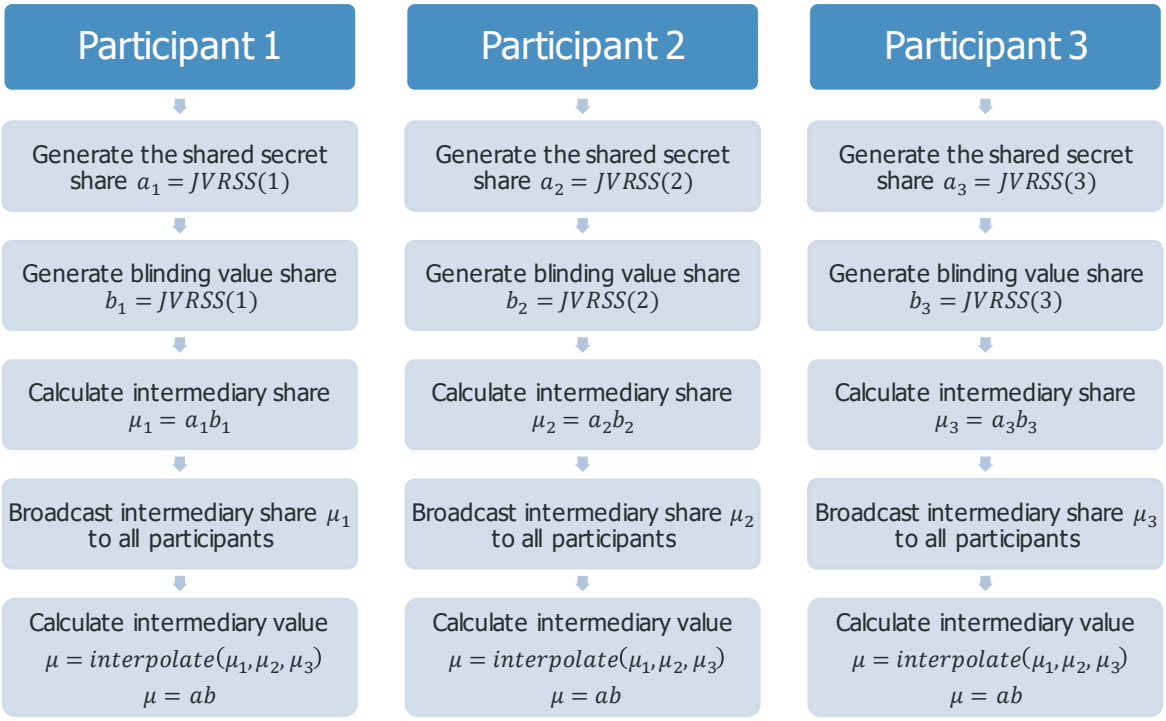


Figure 2: A graphical representation of the addition of two shared secrets.

In this graph there are two shared secrets corresponding to the orange and purple polynomials. The addition of these polynomials results in the blue polynomial. Each participant adds their shares from the orange and the purple polynomial, resulting in an additive share on the blue polynomial. Interpolating over at least two shares on this polynomial results in the addition of the two secrets $(a + b)$.

3.3 Product of shared secrets

We describe the steps for calculating a product of shared secrets.



These steps correspond to steps 1-5 in the protocol describing the product of shared secrets in section 2.3.

We now show why the interpolation of the $\mu_i = a_i b_i$ shares leads to $\mu = ab$. Figure 3 illustrates the result of the multiplication of two polynomials of order-1.

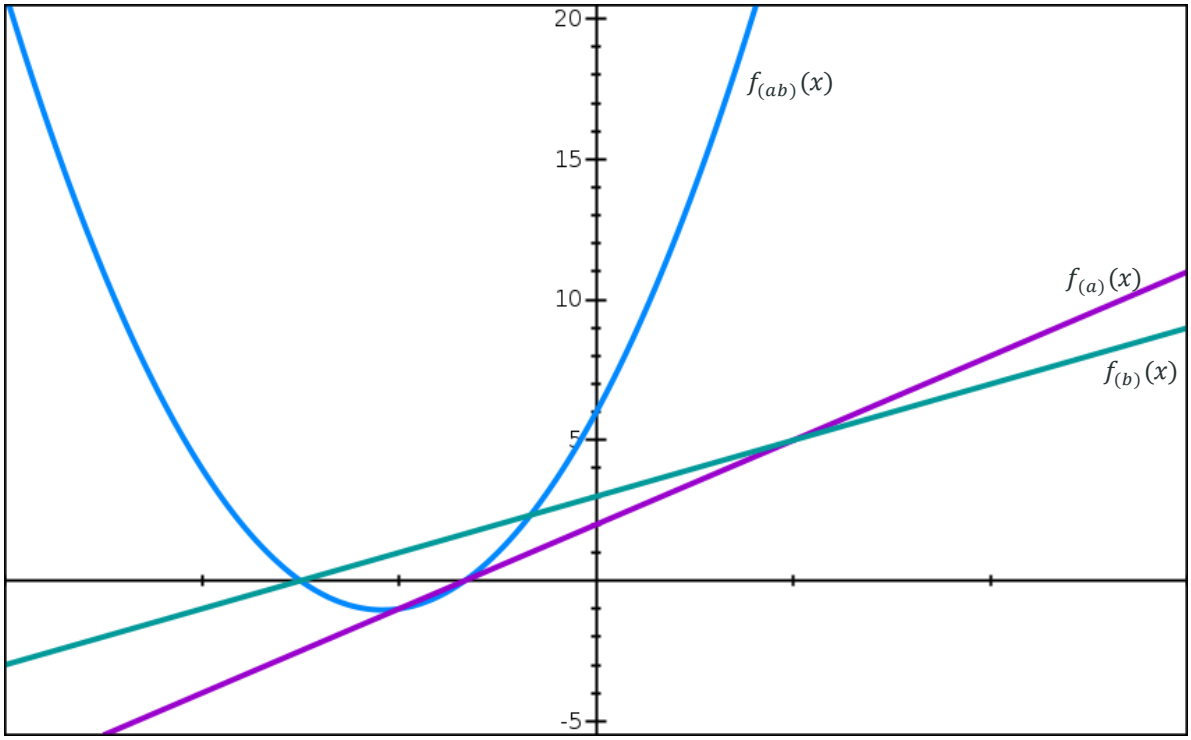


Figure 3: A graphical representation of a product of shared secrets.

The teal and purple lines in this graph are the shared polynomials corresponding to the secrets a and b, respectively. The shared polynomial corresponding to the multiplication of these is given by the order-2 polynomial in blue, and the value ab is exactly where this multiplicative polynomial crosses the y-axis. Since the polynomial corresponding to ab is of order-2, at least 3 participants are required to calculate the multiplicative polynomial secret μ, as described in section 2.3.

3.4 Inverse of a shared secret

We describe how to calculate the modular inverse of a shared secret.

Participant 1	Participant 2	Participant 3
Generate the shared secret product $PROSS(1) = \mu = ab$	Generate the shared secret product $PROSS(2) = \mu = ab$	Generate the shared secret product $PROSS(3) = \mu = ab$
Calculate the modular inverse $\mu^{-1} = (ab)^{-1} \bmod n$	Calculate the modular inverse $\mu^{-1} = (ab)^{-1} \bmod n$	Calculate the modular inverse $\mu^{-1} = (ab)^{-1} \bmod n$
Calculate inverse secret share $a_1^{-1} = \mu^{-1} b_1$	Calculate inverse secret share $a_2^{-1} = \mu^{-1} b_2$	Calculate inverse secret share $a_3^{-1} = \mu^{-1} b_3$

These inverse key shares can be used as a usual shared secret and interpolation will result in the calculation of a^{-1}.

As mentioned, the blinding value is necessary to securely calculate the inverse of the ephemeral key. . Because the shared ephemeral private key is the sum of the individuals shares, the inverse of this cannot simply be calculated as the interpolation of the inverse shares, that is,

$$a^{-1} = (interpolate(a_1, a_2, a_3))^{-1} \neq interpolate(a_1^{-1}, a_2^{-1}, a_3^{-1}) .$$

The inverse is instead calculated using the product of the shared secret with a blinding value. The interpolation over the product of the shares of the secrets calculates

$$\mu = ab = (a_{10} + a_{20} + a_{30})(b_{10} + b_{20} + b_{30}) ,$$

and the inverse of this is the blinded inverse of the ephemeral key

$$\mu^{-1} = a^{-1}b^{-1} = (a_{10} + a_{20} + a_{30})^{-1}(b_{10} + b_{20} + b_{30})^{-1}$$

As mentioned, the blinding value must also be kept secret, otherwise one can calculate the shared secret.

To use this blinded inverse $\mu^{-1} = (ab)^{-1}$, the participants multiply the value by their blinding value share to find an inverse shared secret share

$$a_i^{-1} = \mu^{-1} b_i \ .$$

Interpolating over these shares would result in

$$\mu^{-1} b = a^{-1} b^{-1} b = a^{-1} .$$

Since this is the secret that was being hidden in the first place, this calculation is not done, and is mentioned for illustrative purposes. However, the inverse shared secret shares a_i^{-1} can be used in calculations with other shared secrets, such as the product of shared secrets again. For example, if a shared secret share c_i is multiplied with the inverse shared secret shares and then interpolated, one finds the product of the shared secret c multiplied with the inverse of a shared secret a^{-1}

$$a^{-1}c = interpolate\,(a_1^{-1}c_1, \dots, a_l^{-1}c_l) ,$$

where l is at least the threshold of the product of the shared secret polynomials.

This completes the worked example of JVRSS. We now describe a use case for this method, which is threshold signatures.

4 Threshold signatures

In this section we give an example of how to use the multiparty computations given in the previous sections. This is to create a signature using a shared secret key between multiple parties, without revealing the shared secret key.

A signature on a message is proof that the owner of a private key acknowledges the message and that the message has not been tampered with since signing. Only the person with knowledge of the private key can sign a message, and the signature depends on the message to ensure the message isn't modified. In Elliptic Curve cryptography, the integrity of a signature depends on the difficultly of solving the Elliptic Curve Discrete Logarithm Problem. That is, it is difficult to calculate a private key, given the corresponding public key, and so it can be assumed that only the owner of a private key can create a signature corresponding to that private key.

In Bitcoin, outputs of transactions can be spent by creating a signature using the private key corresponding to the public key given in that output. These signatures are constructed and validated using the Elliptic Curve Digital Signature Algorithm (ECDSA). This algorithm is easy to do with a single signer, and transactions which are to be signed by multiple parties can simply repeat this algorithm for each individual signer.

In the following section we describe an alternative way to create transactions requiring multiple signers. Signatures created using this scheme below are the same as signatures created by single signers. That is, given a threshold signature, it looks like an ECDSA signature created with a single private key. Therefore, the locking and unlocking script in a transaction using this scheme is the same as the usual P2PKH script, and this provides privacy to users of the scheme.

The concepts in this section are based on a paper by Gennaro et al [2]. We finish this introduction with the necessary mathematical background to understand the Threshold Signature algorithm. We begin by describing ECDSA for a given private key using the secp256k1 elliptic curve.

Elliptic Curve Digital Signature Algorithm (ECDSA)

In order to create a signature on a message, the message digest is first calculated

$$e = SHA\text{-}256\big(SHA\text{-}256(message)\big)\ ,$$

by definition. Then the signature is generated in the following way.

1. Choose a random integer $k \in \{1, \dots, n-1\}$, where n is the order of the secp256k1 curve and call k the ephemeral key.
2. Calculate the ephemeral public key corresponding to this ephemeral private key
$$k \cdot G = (x, y)\ .$$
3. Calculate $r = x \bmod n$. If $r = 0$, return to step 1.
4. Calculate the multiplicative inverse of the ephemeral key k^{-1} in the finite field corresponding the secp256k1 curve.
5. Calculate $s = k^{-1}(e + ar) \bmod n$. If $s = 0$, return to step 1.
6. The signature on the message is (r, s).

Note that each time a signature is generated, a different ephemeral key must be used. If this doesn't happen, it is possible to calculate the private key.

Given a message, a public key $(a \cdot G)$, and corresponding signature (r, s), then one can verify the signature by completing the following steps.

1. Calculate the message digest $e = SHA\text{-}256\big(SHA\text{-}256(message)\big)$.
2. Calculate the multiplicative inverse s^{-1} of s modulo n.
3. Calculate $j_1 = es^{-1} \bmod n$ and $j_2 = rs^{-1} \bmod n$.

4. Calculate the point $Q = j_1 \cdot G + j_2(a \cdot G)$.
5. If $Q = \mathcal{O}$, the point at infinity, the signature is invalid.
6. If $Q \neq \mathcal{O}$, then let $Q \coloneqq (x, y)$, and calculate $u = x \bmod n$. If $u = r$, the signature is valid.

This completes the description of ECDSA signature calculation and verification. We now describe how to generate a shared private key a, that is later used to create an ECDSA signature.

4.1 Shared private key generation and verification

To calculate a shared private key a between N participants, $M = 2t + 1$ of which are required to create a signature, they execute JVRSS described in section 2.1. The result is that every participant $i = 1, \dots, N$ has a private key share a_i with threshold $t + 1$. The participants calculate the shared public key using the method described at the end of section 2.1. This results in a shared public key $(a \cdot G)$, and each participant safely stores this along with their private key share a_i. This shared private key shall be used to sign messages using the ECDSA scheme.

Note that signatures with the shared secret require $2t + 1$ participants. This is because the ephemeral key must be a shared secret as well as the private key. If the ephemeral key were not secret, then anyone with knowledge of the signature, message, and ephemeral private key would be able to calculate the shared private key a. Therefore, a signature is calculated with the multiplication of two shared secret polynomials of order t, and so the multiplicative shared polynomial will have order $2t$, with $2t + 1$ shares required to recreate it. Finally, this implies that the signing threshold of a scheme with N participants is restricted by the inequality $2t + 1 \leq N$. Clearly, if the threshold is larger than the number of participants, then the group will not be able to sign any messages.

We describe the process for generating shared ephemeral keys that will be used in the signature.

4.2 Ephemeral key shares generation

The next step is to generate ephemeral key shares and the corresponding r, as required in a signature. The general process for this is as follows.

1. Generate the inverse share of a shared secret $k_i^{-1} = INVSS(i)$, where $(t + 1)$ shares are required to recreate it.
2. Each participant calculates
$$(x, y) = \sum_{i=1}^{N} (k_{i0} \cdot G),$$
using the obfuscated coefficients shared in the verification of k_i, then they calculate
$$r = x \bmod n .$$
3. Each participant i stores (r, k_i^{-1}).

Note that ephemeral keys should never be used more than once to calculate a signature. If ephemeral keys are used for different messages, the private key is calculable.

4.3 Signature generation

We describe the process for generating a signature with the shared private key a. Assume that at least $M = 2t + 1$ participants would like to create a signature on a message, and one of the participants chooses to coordinate this. The following steps are taken.

1. The coordinator requests a signature on the message from at least $M = 2t + 1$ participants.

2. Each participant i recovers the next ephemeral key (r, k_i^{-1}). All users must use a share corresponding to the same ephemeral key.
3. Each participant calculates the message digest $e = SHA\text{-}256\,(SHA\text{-}256\,(message))$.
4. Each participant i calculates their own signature share s_i
$$s_i = k_i^{-1}(e + a_i r) \bmod n\ ,$$
where a_i is their private key share.
5. Each participant sends their signature share (r, s_i) to the coordinator.
6. When the coordinator has received M signature shares, they calculate
$$s = interpolate\,(s_1, \dots, s_{2t+1})\ ,$$
and output the signature as (r, s).
7. The coordinator verifies the signature using the ECDSA described in section 4.1. If this fails, at least one of the shares must be incorrect, and the signature generation algorithm should be run again.

It is possible to find a dishonest actor if the signature fails the verification, with the information obtained in JVRSS. Each coordinator wanting to find a malicious actor would need to store some additional information from the JVRSS protocol. Recall that the public keys corresponding to each participant's point on each participant's private polynomial is shared

$$f_i(j) \cdot G, \qquad \forall i, j = 1, \dots, N.$$

All participants can calculate the public key corresponding to each participant j's private key a_j share by simply adding the corresponding values

$$a_j \cdot G = \sum_{i=1}^{n} f_i(j) \cdot G\ .$$

Given participant j's signature share (r, s_j), this public key $a_j \cdot G$, and the message, one can verify is this is the correct signature share. If the verification does not hold, the coordinator knows that this is an incorrect share.

We now do a worked example of a group of size 3, all of which are required to calculate a signature. This is called a 3-of-3 signature scheme.

5 Threshold signatures worked example

We give a simple example of a threshold signature scheme, a 3-of-3 scheme, meaning that all three participants in a scheme are required to generate a signature.

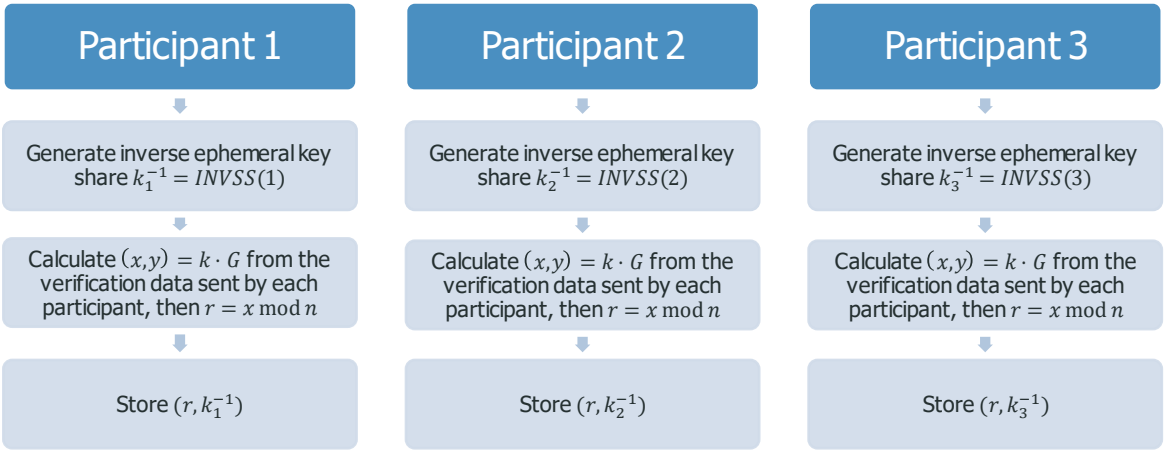
5.1 Shared private key generation and verification

All the participants generate and verify their share of the signing key and calculate the shared public key. This is done using JVRSS and method 2 of calculating the shared public key as described in section 2.1. The result is that each participant has a share a_i of a shared private key a, with corresponding shared public key $(a \cdot G)$.

5.2 Ephemeral key shares generation

Multiple of these ephemeral keys and the corresponding r value are pre-calculated and stored at the initial set-up, so that they do not need to be calculated for each signing.

We describe the process for generating these ephemeral keys. We focus on the generation of one ephemeral key and its corresponding r, and this process is repeated for as many precalculated ephemeral keys as required.

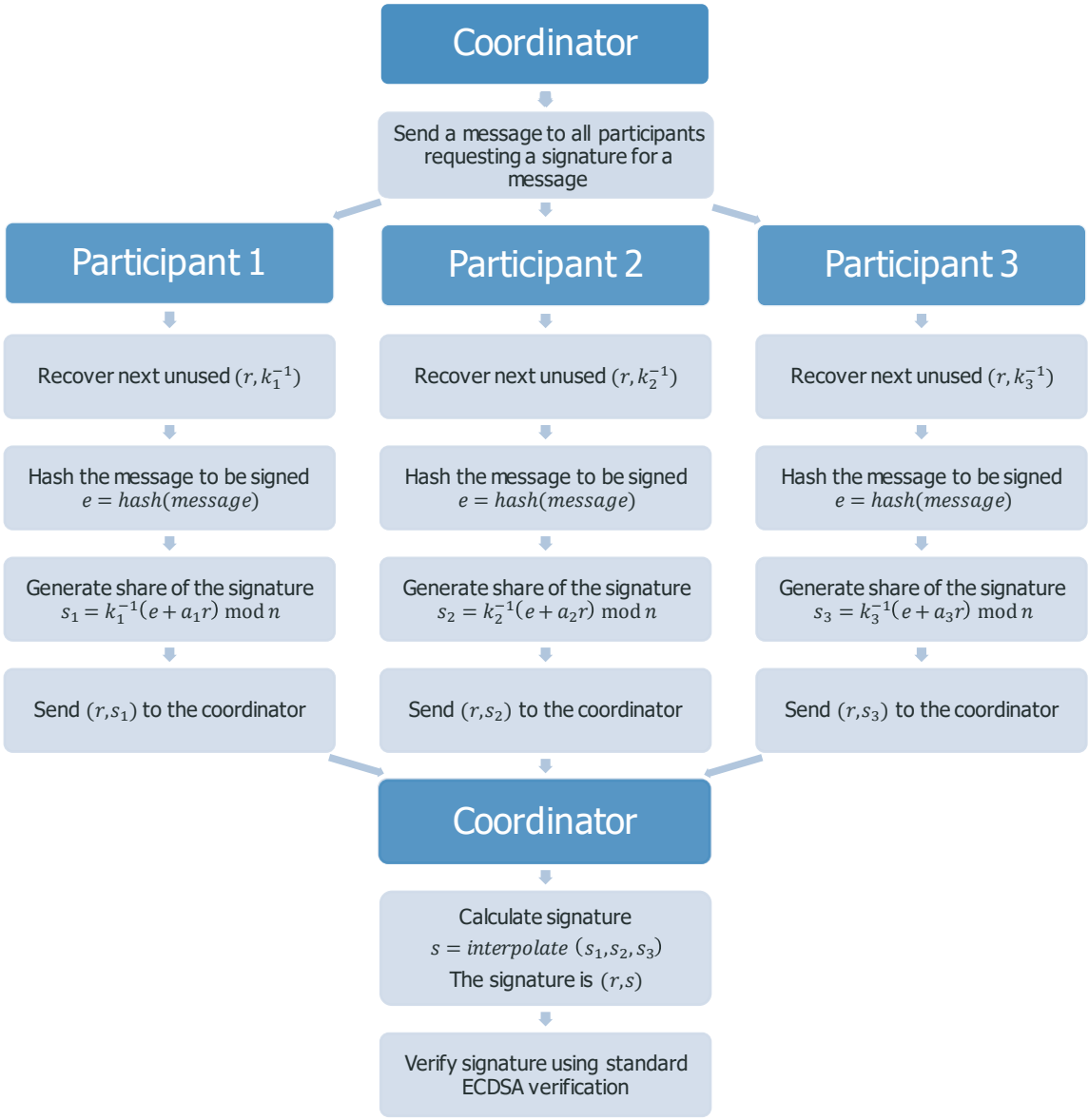


In the second step, $k \cdot G$ is calculated using $k_{i0} \cdot G$ that each participant shares in the JVRSS step. Then the x-coordinate of the result modulo n is the definition of r.

This completes the description of the ephemeral key shares. We now describe how to use these and the shared private key to calculate a shared signature.

5.3 Signature generation

To calculate a shared signature in a 3-of-3 scheme, the following steps are taken.



Lagrange interpolation over the signature shares effectively calculates the polynomial

$$f_{(s)}(x) = (k^{-1}\alpha^{-1}) \times f_{(\alpha)}(x)\left(e + f_{(a)}(x)\, r\right) ,$$

where k is the ephemeral key, α is the blinding secret, $f_{(\alpha)}(x)$ is the order-1 polynomial corresponding to the blinding secret, and $f_{(a)}(x)$ is the order-1 polynomial corresponding to the shared private key. The polynomial corresponding to the signature is order-2 and so 3 participants are required to create it. The value of this at $x = 0$ is

$$f_{(s)}(0) = k^{-1}(e + ar) ,$$

which is the signature on the message by the private key a as required. Finally, the signature is (r, s). This completes the worked example of a 3-of-3 threshold signature scheme, and our description of secret sharing and threshold signatures.

6 References

Reference	Author, date, name & location
[1]	Pedersen, T. P. "A threshold cryptosystem without a trusted party." *Workshop on the Theory and Application of Cryptographic Techniques*. Springer, Berlin, Heidelberg, 1991.
[2]	Gennaro, R, et al. "Robust threshold DSS signatures." *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, Berlin, Heidelberg, 1996.