Weighted Schnorr Threshold Signatures

Joseph Wiley Yandle*

2023-01-19

Abstract. We present Weighted Schnorr Threshold Signatures (WSTS aka WileyProofs), which optimizes the base FROST implementation to reduce bandwidth when FROST parties control multiple keys.

Keywords: Weighted Aggregate Threshold Signatures; Zero-Knowledge Proofs; FROST

1 Introduction

FROST *(Flexible Round-Optimized Schnorr Threshold)* [1] is a system for making aggregate threshold signatures. This allows a set of *parties* to construct a group key, and then sign messages using this key as long as a *threshold* subset cooperates to form the signature. This signature is aggregated, so that its size does not depend on the number of signing parties.

The design of FROST assumes that each *party* controls exactly one key, i.e. the *threshold* is not weighted. The naive approach to turning FROST into a weighted threshold scheme involves allowing each *signer* to control a subset of the *parties* proportional to its weight. But this results in inefficiencies in the number of messages required for the protocol, total bandwidth, and computational complexity.

Here we present optimizations on top of vanilla FROST which reduce these inefficiencies when used as a weighted threshold scheme. We call this scheme WSTS *(Weighted Schnorr Threshold Signatures)*, aka WileyProofs, in honor not only of the author's grandfather, but also the greatest engineer of all time: Wiley E. Coyote.

1.1 Notation

While WSTS can be implemented in any group, we will use curve point terminology exclusively. This will hopefully make it more accessible to engineers wishing to implement it.

Let G be a generator in an elliptic curve group $\mathbb{G}$. Let f be a polynomial of a scalar variable with scalar coefficients, and P be a scalar polynomial with curve point coefficients. Both f and P can be evaluated at a scalar, though the latter will use scalar/point multiplication rather than scalar/scalar.

Let H be a hash function, which maps a series of binary inputs to a fixed size binary digest. For a hash function to be effective, this mapping must be $one - way$; i.e. it should be impossible to go from a hash digest back to the binary inputs. The most common hash functions include SHA2, SHA3, and Blake, each of which can create a variety of digest sizes, frequently 256 or 512 bits.

*xoloki@gmail.com

2 Background

2.1 Threshold Signatures

Threshold signatures allow a subset of a group of N signers to collaborate to sign messages. There are many such protocols, but they all specify a threshold T, which is the number of signers necessary to create a valid signature.

2.2 Aggregate Signatures

Traditional approaches to the multisignature problem require that each signer signs separately, which means that the size of the signature grows linearly with the number of required signers. An aggregate scheme, conversely, condenses these individual signatures into a single group signature. This allows a large set of signers without increasing the data size necessary to transmit and store the signature.

In a blockchain context, where transaction fees are proportional to the size of the data, this leads to significant savings. It also prevents the case where a large group signature would exceed the blocksize, thereby artificially limiting the number of signers.

2.3 Aggregate Threshold Signatures

By combining the concepts of aggregate and threshold signatures, we arrive at a construction well suited to a blockchain context. We can have any number of possible signers, and any threshold required to make a valid signature. This allows signatures which can be used in many more contexts than a traditional multisig construct, while keeping transaction fees low and taking no more blockchain space than a standard single signature.

2.4 Weighted Aggregate Threshold Signatures

Aggregate threshold signatures function well in a blockchain context, but the number of messages and associated protocol bandwidth required grow linearly with the total number of parties. In many use cases, though, not all signers should be given equal weight. This is particularly relevant to Proof of Stake blockchain systems, where the size of a user's stake gives their vote more weight.

It would be ideal to construct a native weighted threshold scheme, where this size would instead grow proportionally to the number of actual signers, rather than the total number of votes.

2.5 Polynomial Interpolation

Consider a polynomial f of degree k:

$$f(x) = a_0 + a_1 x + a_2 x^2 + ... + a_k x^k$$

There are $k+1$ coefficients needed to form a polynomial of degree k. Thus in order to uniquely determine a polynomial of degree k, we will need to evaluate it at $k+1$ points, giving us a system of $k+1$ equations in $k+1$ unknowns. Typically, we evaluate at the points $[1, k+1]$. This is called polynomial interpolation.

While we can use linear algebra to solve these equations, there is no closed form which can be used algebraically. Lagrange interpolation [2] is an alternative approach which does provide a closed form. To interpolate $f(x)$ it uses the Lagrange polynomial $L(x)$ and the Lagrange bases $l_i(x)$:

$$L(x) = \sum_{i=1}^{k+1} f(x_i) \cdot l_i(x) \tag{1}$$

$$l_i(x) = \prod_{j=1, j\neq i}^{k+1} \frac{x - x_j}{x_i - x_j} \tag{2}$$

We can see that $l_i(x_i)$ must be 1, since $\frac{x_i - x_j}{x_i - x_j} = 1$. For all $m \in [1, k+1] \ni m \neq i$, $l_i(x_m)$ must be 0, since one of the top terms will be $x_m - x_m$, hence the product must be 0. So $\forall i \in [1, k+1];\ L(x_i) = f(x_i)$.

If we only need to evaluate the interpolated polynomial at 0, we can condense $l_i(0)$ to:

$$l_i(0) = \prod_{j=1; j\neq i}^{k+1} \frac{x_j}{x_j - x_i} \tag{3}$$

We call this λ_i, and use it to quickly evaluate $L(0)$ when given a set $(x_i, f(x_i))$.

2.6 Schnorr Proofs

A Schnorr proof [3] is a zero knowledge proof of ownership of a public key. Let scalar x be a private key, with point $X = x \cdot G$ the corresponding public key. To prove knowledge of x given X, Schnorr proofs use a 3-move commitment-challenge-response protocol, i.e. a Sigma protocol.

The prover first commits to a random scalar v with the corresponding point V, then sends V to the verifier. The verifier then returns a random challenge scalar c. The prover then constructs the response $r = v + cx$. The proof consists of the tuple (r, c, V).

To verify the proof, check that:

$$V = r \cdot G - c \cdot X \tag{4}$$

This must be true for a valid proof, since:

$$V = v \cdot G = (r - cx) \cdot G = r \cdot G - cx \cdot G = r \cdot G - c \cdot X$$

To make this protocol non-interactive, use the Fiat-Shamir transform [4] to construct c by hashing the initial proof elements (G, X, V):

$$c = H(G, X, V) \tag{5}$$

2.7 FROST

FROST is an aggregate threshold signing scheme. It allows for a group of N parties to create a distributed group signing key, then some threshold T of them can cooperate to sign a message. This group signature is an aggregate of the individual party signatures, and functions as a Schnorr proof.

Thus there are three discrete elements required: distributed key generation, gathering party signatures, and forming an aggregate group signature using the party signatures.

2.7.1 Distributed Key Generation (DKG)

Like many DKG schemes, FROST employs the technique of polynomial interpolation. But whereas traditional DKG protocols use a trusted dealer, FROST is a trustless protocol. To accomplish this, each party acts as a trusted dealer to every other party, and runs a traditional DKG protocol. The parties then combine their keys trustlessly, and the result is a fully trustless distributed key, of which each party controls an equal share.

Remember that in polynomial interpolation, you need $k + 1$ points to interpolate a degree k polynomial. FROST specifies a threshold T, which is the number of signers necessary to sign a message. Thus we will need a group polynomial of degree $T - 1$ so that any group of T signers will be able to interpolate the key, and thereby sign. This group polynomial f will be the sum of the party polynomials f_i; the group private key x will be $f(0)$, and the group public key Y will be $P(0)$, where $P = f \cdot G$.

First, each party is given a sequential ID $i \in [1, N]$, which will be used both to identify it and as an element in the underlying math. Next, each party i constructs a random private scalar polynomial f_i of degree $T - 1$, where the coefficients are also scalars. A public DKG share is constructed from f_i by multiplying each coefficient by a generator G. The ID i and public polynomial P_i are then shared to all other parties.

Each party i evaluates its own f_i at each $j \in [1, N]$, and sends a private DKG share to each consisting of $(j, f_i(j))$. Each receiving party checks each private share for consistency with the public polynomial of the sending party. This is done for party i by checking that:

$$\forall j \in [1, N] \ni j \neq i;\ f_j(i) \cdot G = P_j(i) \tag{6}$$

Once the private shares are verified, each party i computes its share s_i of the group secret key by summing the private polynomial evaluations at i from all parties, then uses it to construct its public key Y_i:

$$s_i = \sum_{j=1}^{N} f_j(i);\ Y_i = s_i \cdot G \tag{7}$$

Each party computes the group public key Y by summing the first coefficient of each public polynomial:

$$Y = \sum_{j=1}^{N} P_j(0) \tag{8}$$

At this point DKG is complete, and each party i knows (s_i, Y_i) and the group key Y.

2.7.2 Gathering Party Signatures

To sign a message m, it is necessary to choose the signing set S before any party can sign, with $|S| >= T$; this allows polynomial interpolation to work. Each party $i \in S$ then constructs a private nonce (d_i, e_i) and corresponding public nonce (D_i, E_i), then sends (i, D_i, E_i) to all other parties in S; this forms the set B.

Once nonces have been received, each party $i \in S$ computes the binding values ρ_j using B and m:

$$\forall j \in S;\ \rho_j = H(j, m, B) \tag{9}$$

Next they compute the interpolation value λ_i:

$$\lambda_i = \prod_{j \in S; j \neq i} \frac{j}{j - i} \tag{10}$$

Each party also computes the sum R:

$$R = \sum_{j \in S} D_j + \rho_j \cdot E_j \tag{11}$$

Once they have R and message m they can construct the challenge c:

$$c = H(Y, R, m) \tag{12}$$

Finally, each party i can construct its signature share z_i:

$$z_i = d_i + \rho_i \cdot e_i + s_i \cdot c \cdot \lambda_i \tag{13}$$

2.7.3 Aggregating the Group Signature

Once the aggregator has gathered the signature shares z_i, it must first verify that they are valid by checking that:

$$z_i \cdot G = D_i + \rho_i \cdot E_i + Y_i \cdot c \cdot \lambda_i \tag{14}$$

Finally the aggregator can construct the group signature z:

$$z = \sum_{i \in S} z_i \tag{15}$$

The full signature consists of (R, z) using R from (11).

2.7.4 Verifying the Group Signature

To verify the group signature, construct the value R' from the group public key Y, challenge c, and group signature z:

$$R' = z \cdot G - c \cdot Y \tag{16}$$

If $R = R'$, the proof is valid.

2.7.5 FROST as a Schnorr Proof

How does the FROST algorithm produce a Schnorr proof? To do so, it should first commit to some randomness v via V, produce a challenge that depends on V, and then give a response that combines the commitment v, challenge c, and underlying secret x: $r = v + cx$.

This is exactly what FROST does. The committed values are the nonces (d_i, e_i), shared as (D_i, E_i), and aggregated into the sum R, which is used to construct the challenge. And if we expand the aggregation of the party signatures z_i, we can see that the response, the signature z, is indeed $v + cx$:

$$z = \sum_{i \in S} z_i = \sum_{i \in S} d_i + \rho_i \cdot e_i + s_i \cdot c \cdot \lambda_i = \sum_{i \in S} (d_i + \rho_i \cdot e_i) + c \cdot \sum_{i \in S} (s_i \cdot \lambda_i)$$

As we know from Lagrange interpolation, $\sum_{i \in S} s_i \cdot \lambda_i$ is the group polynomial evaluated at 0, which is the group private key x.

3 Related Work

There are many aggregate threshold signature schemes, some quite similar to FROST. While FROST is optimized for a reduced number of rounds in the optimistic case of no byzantine actors, there are others which are called *robust*; i.e. they take more rounds in the common case, but can complete even if some signers are byzantine actors.

An excellent example of a robust protocol is Provably Secure Distributed Schnorr Signatures (PSDSS) [5]. The construction features robustness in both the DKG and signing rounds. At both stages, the protocol will succeed as long as at least T parties complete it honestly.

We did not choose PSDSS as the basis for WSTS for two reasons. First, rather than gather nonces from signing parties in each signing round, it runs an additional DKG to create an ephemeral round key. Since DKG is an expensive protocol in the case of a large number of voting slots, it was suboptimal for our use case. Also, WSTS-style bandwidth reductions did not appear to be viable for PSDSS.

There are also many native weighted threshold signing schemes, but ultimately all were rejected. Most of them relied on RSA with a trusted setup, and so were considered prima facie unsuitable.

There was one compelling candidate though: An Efficient and Secure Weighted Threshold Signcryption Scheme [6]. It uses standard elliptic curves combined with a dynamic knapsack sytem, and the Chinese Remainder theorem to set the voting weights. We rejected it due to its complexity and lack of track record in production systems. But it remains an ongoing object of research.

4 WSTS: Weighted Schnorr Threshold Signatures

Our contribution is a set of optimizations on top of vanilla FROST to streamline the protocol in a heavily weighted threshold scenario (e.g. 150 signers controlling a total of 4000 keys). In FROST, each party controls a single key, which acts as a single vote. A naive implementation of a weighted threshold scheme on top of FROST involves giving each signer control of multiple parties, proportional to that signer's weight. This is functional, but suboptimal.

WSTS optimizes this by having each signer control a single party, but that party now controls multiple shares of the group key. Crucially, each party still uses only a single polynomial per DKG round and a single nonce per signing round. This also allows for a single signature per party, rather than per key. This leads to substantial savings during both DKG and signing.

Whereas FROST contains two primary parameters (T and N), WSTS splits the N parameter into two: the number of parties N_p, and the number of keys N_k. T is now a threshold of the keys, not the parties. Each party must still send a DKG private share for each key, but DKG public and signature shares are now per party.

Each party is given not only a party ID $i_p \in [1, N_p]$ as in FROST, but also a set of key IDs $i_k \in [1, N_k]$. We denote the latter set as K_{i_p}.

4.1 Distributed Key Generation (DKG)

Each party $i_p \in [1, N_p]$ begins DKG by generating a random scalar polynomial f_{i_p} with scalar coefficients. They then construct DKG public shares by multiplying the coefficients of f_{i_p} by the generator, to obtain P_{i_p}.

Once the DKG public shares have been distributed, the parties create DKG private shares by evaluating f_{i_p} at all points $j_k \in [1, N_k]$. These shares consist of a set of tuples $(i_p, (i_k, f_{i_p}(i_k)))$.

Once a party receives its DKG public and private shares, it first must verify that each share is valid using a similar formula as before:

$$\forall j_p \in [1, N_p], i_k \in K_{i_p}; f_{j_p}(i_k) \cdot G = P_{j_p}(i_k) \tag{17}$$

Once the private shares are verified, each party i_p computes its shares s_{i_k} of the group secret key by summing the private polynomial evaluations from all parties for each key i_k, giving the group polynomial f evaluated at i_k, i.e. one interpolation point. The parties then use each s_{i_k} to construct the corresponding public key Y_{i_k}:

$$s_{i_k} = \sum_{j_p=1}^{N_p} f_{j_p}(i_k);\ Y_{i_k} = s_{i_k} \cdot G \tag{18}$$

Each party computes the group public key Y by summing the first coefficient of each public polynomial:

$$Y = \sum_{j_p=1}^{N_p} P_{j_p}(0) \tag{19}$$

At this point DKG is complete, and each party i_p knows all of its (s_{i_k}, Y_{i_k}) and the group key Y.

4.2 Gathering Party Signatures

As before, in order for polynomial interpolation to work, it is necessary to choose the signing parties $S_p \ni \forall s_p \in S_p, s_p \in [1, N_p]$, and the corresponding S_k which consists of the union of all $i_k \in K_{i_p}\ \forall i_p \in S_p$, before any party can sign, with $|S_k| >= T$. Each party $i_p \in S_p$ then constructs a private nonce (d_{i_p}, e_{i_p}) and corresponding public nonce (D_{i_p}, E_{i_p}), then sends (i_p, D_{i_p}, E_{i_p}) to all other parties with keys in S_k, forming set B as before.

Once nonces have been received, each party $i_p \in S_p$ computes the binding values ρ_{j_p} using B and m:

$$\forall j_p \in S_p;\ \rho_{j_p} = H(j_p, m, B) \tag{20}$$

Next, $\forall i_k \in K_{i_p}$, they compute the interpolation values λ_{i_k}:

$$\lambda_{i_k} = \prod_{j_k \in S_k; j_k \neq i_k} \frac{j_k}{j_k - i_k} \tag{21}$$

Each party also computes the sum R:

$$R = \sum_{j_p \in S_p} D_{j_p} + \rho_{j_p} \cdot E_{j_p} \tag{22}$$

Once they have R and message m they can construct the challenge c:

$$c = H(Y, R, m) \tag{23}$$

Finally, each party i_p can construct its signature share z_{i_p}:

$$z_{i_p} = d_{i_p} + \rho_{i_p} \cdot e_{i_p} + \sum_{j_k \in K_{i_p}} s_{j_k} \cdot c \cdot \lambda_{j_k} \tag{24}$$

4.3 Aggregating the Group Signature

Once the aggregator has gathered the signature shares z_{i_p}, it must first verify that they are valid by checking that:

$$z_{i_p} \cdot G = D_{i_p} + \rho_{i_p} \cdot E_{i_p} + \sum_{j_k \in K_{i_p}} Y_{j_k} \cdot c \cdot \lambda_{j_k} \tag{25}$$

Finally the aggregator can construct the group signature z:

$$z = \sum_{i_p \in S_p} z_{i_p} \tag{26}$$

The full signature consists of (R, z) using R from (22).

4.4 Verifying the Group Signature

To verify the group signature, construct the value R' from the group public key Y, challenge c, and group signature z:

$$R' = z \cdot G - c \cdot Y \tag{27}$$

If $R = R'$, the proof is valid.

4.4.1 WSTS as a Schnorr Proof

WSTS constructs a Schnorr proof in the same way as FROST, with the same commitment value and challenge. The response, the signature z, likewise expands to $v + cx$:

$$z = \sum_{i_p \in S_p} z_{i_p} = \sum_{i_p \in S_p} d_{i_p} + \rho_{i_p} \cdot e_{i_p} + \sum_{j_k \in K_{i_p}} s_{j_k} \cdot c \cdot \lambda_{j_k} = \sum_{i_p \in S_p} (d_{i_p} + \rho_{i_p} \cdot e_{i_p}) + c \cdot \sum_{j_k \in S_k} (s_{j_k} \cdot \lambda_{j_k})$$

As we know from Lagrange interpolation, $\sum_{j_k \in S_k} s_{j_k} \cdot \lambda_{j_k}$ is the group polynomial evaluated at 0, which is the group private key x.

5 Performance

The WSTS Rust reference implementation [7] contains vanilla FROST in the v1 module, and WSTS in the v2 module. This crate contains criterion benchmarks for both v1 and v2.

The results are quite staggering. When testing DKG with 4 signers and 20 keys, v2 outperforms v1 by an order of magnitude: $53ms$ per DKG round for v1 compared to $8.3ms$ for v2. Similar speedups were noted for party signing and group signing.

In the table, N_s will refer to the number of signers, which is N in FROST and N_p in WSTS. N_k is the number of keys, and T is the threshold.

Table 1: FROST vs WSTS (runtime)

			DKG		Party Sign		Group Sign	
N_s	N_k	T	v1	v2	v1	v2	v1	v2
4	20	13	53ms	8.3ms	18ms	0.80ms	2.1ms	0.85ms
4	40	13	200ms	15ms	120ms	2.0ms	6.5ms	2.2ms
4	40	26	420ms	36ms	120ms	2.0ms	6.6ms	2.3ms
4	60	26	1000ms	59ms	380ms	4.0ms	13ms	4.6ms
4	60	40	1900ms	120ms	380ms	4.0ms	13ms	4.6ms
4	80	40	3400ms	160ms	880ms	6.7ms	23ms	7.6ms
4	80	53	4900ms	240ms	880ms	6.7ms	23ms	7.6ms
4	100	53	7900ms	300ms	1700ms	10ms	35ms	12ms
4	100	66	10000ms	400ms	1700ms	10ms	35ms	12ms

6 Robustness

As discussed in Related Work, there are some aggregated threshold schemes which are robust, in that they can complete even in the presence of byzantine actors, as long as there exists a threshold set which completes the protocol honestly. These were rejected for a variety of reasons, as discussed.

However, it is possible to develop a meta protocol on top of a non-robust scheme which provides robustness. It simply requires running a non-robust scheme repeatedly, removing the bad actors as they are exposed. Eventually the meta protocol will converge to success, if a threshold set of honest actors exists, or to failure if it does not. Here we will explore two such meta protocols: FIRE and ROAST.

6.1 FIRE

In addition to WSTS, here we also present FROST Interactive Robustness Extension (FIRE). This is a straightforward extension of FROST into a meta protocol which provides robustness. It works with both FROST and WSTS, with minor differences in the protocol.

A FIRE signing round takes place after a DKG round has established a group key. Each round is labeled with an integer ID i, and consists of a series of sessions j, where j starts at 0 for every round.

Let A be the set of active parties. When a signing round begins, A will contain all parties who successfully completed DKG. We will remove parties from A as the protocol runs, which will allow us to excise byzantine actors.

So when round i begins, we request nonces for session $j = 0$ from A. Once we get $k = T$ nonces for i, j, the corresponding parties will be selected for the signing set S_j. We then request signature shares from all parties in S_j. If all sign correctly, and we obtain a valid signature, round i is complete. We must set a timeout for the signing portion of the session, or else byzantine actors can slow the protocol indefinitely.

Parties who do not return signatures, or return invalid signatures, are removed from A. Parties who did not return a nonce for round j are also removed. We then we begin session $j + 1$. All members of A are then requested to give nonces for session $j + 1$. The same process happens as before; when we get $k = T$ nonces we form S_{j+1} and again request signature shares.

As before, if the aggregate signature is valid, round i is complete. Otherwise, we begin a new session $j + 2$, and continue in the same vein. Since each session will either complete or remove some parties from A, this algorithm will eventually terminate. When $|A| < T$, signing round i has failed.

When running a FIRE round with WSTS vs FROST, the only difference is that we must count the number of keys controlled by each party who responds with a nonce in every session. Only when the sum of keys controlled by the given nonces equals or exceeds T do we form S_j and begin the signing portion of the session.

6.2 ROAST

ROAST[8] is a wrapper around FROST and other threshold signature schemes which provides robustness and asynchronicity. It operates in a very similar manner to FIRE, but the asynchonicity allows for faster completion if there exists a set of honest and responsive signers.

ROAST differs from FIRE by allowing session $j+1$ to run in parallel with session j. To do so, it is necessary to keep track of two sets: R is the set of responsive signers, and M of known malicious signers. Once a party becomes a member of set M, all messages from that party will be ignored, and if the governance model includes sanctions on bad actors, those in M will be nominated for such.

The ROAST paper describes the protocol in terms of events and responses, which makes an initial read somewhat challenging. We will describe it here more linearly. So as with FIRE, signing round i begins session $j = 0$ with the coordinator asking all signers for nonces. R and M are initially the null set. Signers who receive a request to begin $i, 0$ respond with a nonce n_0 only; this will be different $\forall j \neq 0$.

As the coordinator receives responses, the responding signers are placed into R, and their nonces are placed in ρ_j. Once $|R| = T$, the coordinator sends (ρ_j, R) to the signers to request signature shares. Signers respond with signature z_j, and also a nonce n_{j+1}. Piggybacking the next session's nonce with the current session's signature share improves the responsiveness of the protocol.

Crucially, when the coordinator sends (ρ_j, R) to signers, those signers in R are removed. As they respond, the signers are placed back into R if their signature shares are valid; if they are not valid, those signers are placed into M and declared malicious. Signers who respond with nonces for the current round while the signing session continues are placed into R with those who provided valid signatures.

As signature shares and nonces are received, the coordinator does two things. It first checks to see if it has received a full set of valid signature shares; if this happens, and the group signature is valid, then round i is complete. At the same time, the size of R is examined. When $|R| = T$, session $j + 1$ begins. Crucially, this happens in parallel to session j.

As the sessions continue for each signing round, it is clear that if there exists a threshold set of honest and responsive signers, the protocol will complete succesfully. Every session will either succeed or add malicious actors to M. If $|M| > (N - T)$, then the signing round has failed. Likewise, we will never have more than $N - T$ sessions for any round. Though the paper does not describe the use of timeouts, without them we will never be able to declare that a round is over, since a byzantine set of actors can simply stall any session where one of the set is a participant. But since the session timeouts run in parallel, the speedup even in the byzantine failure case is significant.

As with FIRE, we can use either FROST or WSTS as the signature aggregation protocol, with the only difference being how we count the nonces when determining if we have a threshold set of possible signers.

7 Acknowledgements

The author would like to thank Alie Slade for initial work on the WSTS algorithm.

References

[1] Chelsea Komlo, Ian Goldberg *FROST: Flexible Round-Optimized Schnorr Threshold Signatures* 2020.12.22. https://eprint.iacr.org/2020/852.pdf

[2] *Lagrange polynomial*. https://en.wikipedia.org/wiki/Lagrange%5Fpolynomial

[3] *Schnorr signature*. https://en.wikipedia.org/wiki/Schnorr%5Fsignature

[4] *Fiat Shamir heuristic*. https://en.wikipedia.org/wiki/Fiat%2DShamir%5Fheuristic

[5] D.R. Stinson, R. Strobl *Provably Secure Distributed Schnorr Signatures and a (t,n) Threshold Scheme for Implicit Certificates* ACISP 2001. Lecture Notes in Computer Science, vol 2119 2001.01.23. https://doi.org/10.1007/3-540-47719-5_33

[6] Chien-Hua Tsai *An Efficient and Secure Weighted Threshold Signcryption Scheme* Journal of Internet Technology, vol. 20, no. 5 , pp. 1523-1534, Sep. 2019 https://jit.ndhu.edu.tw/article/view/2135

[7] Joey Yandle *WSTS Reference Implementation* https://github.com/Trust-Machines/wsts

[8] Tim Ruffing, Viktoria Ronge, Elliott Jin, Jonas Schneider-Bensch, Dominique Schroder *ROAST: Robust Asynchronous Schnorr Threshold Signatures* https://eprint.iacr.org/2022/550.pdf

A Security

A.1 Correctness

In Sections 4.4.1 and 2.7.5, we show how WSTS and FROST reduce to a Schnorr proof. This is sufficient to show correctness.