FROST Workshop

by Christopher Scott

https://github.com/cmdruid/workshops

What is FROST?

- Fast Round-Optimized Schnorr Signatures with Threshold.
- Published in December, 2020 by Chelsea Komlo and Iam Goldberg.
- Builds on top of Shamir Secret Sharing (SSS) and Pederson Distributed Key Generation (DKG) protocols.
- Allows a **k** sub-group of **n** participants to sign a message using secret shares, without revealing their share to others.

What can we do with FROST?

- We can publish a signed message by collecting **k** of **n** signatures.
- Control magic internet money using a quorum of **k** of **n** participants.
- Keep the on-chain identity of all signing participants private.

Hidden Benefits of FROST

- Should work with any schnorr-based digital signature protocol.
- The shamir secret key can be provably unknowable (via DKG).
- The **k** of **n** terms of a shamir pubkey can be updated by **k** particpants.
- The terms can be updated without changing the pubkey or secret.

Crash Course Topics

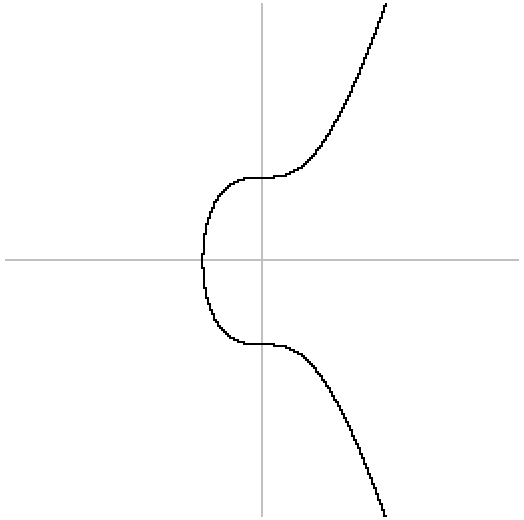
- Polynomials and Interpolation.
- Shamir Secret Sharing.
- Distributed Key Generation (DKG).
- Attacks and vulnerabilities.
- FROST signature protocol.

What is a Polynomial?

- Expresses a sum of terms for one or more variables and coefficients:

  P(x) = $3x^2$ + 2x - 1

- The relationship between **P(x)** and **x** can be plotted on a graph:

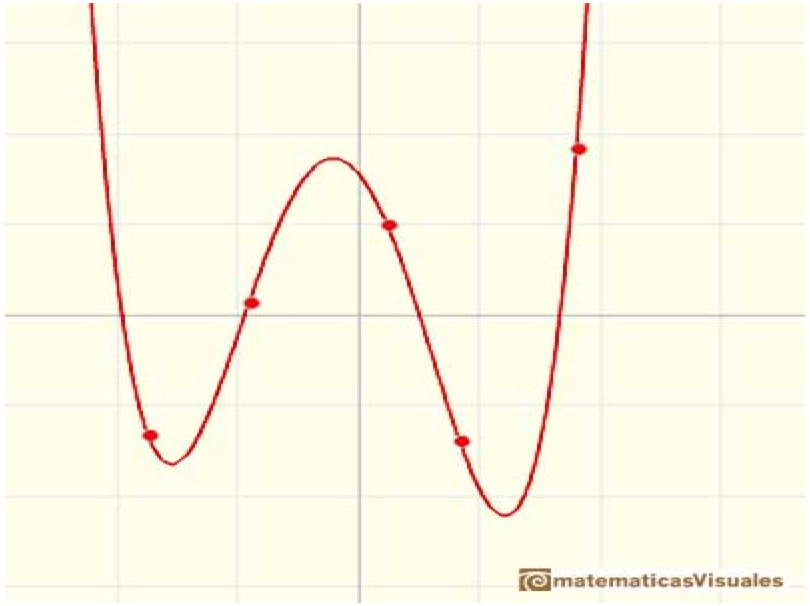


$P(x)^2 = x^3 + 7$

Polynomial Interpolation

- Creates a polynomial based on a set of data points.
- Evaluating **P(x)** will create new points in relation to the existing set.
- The preferred method we use is called Lagrange Interpolation.

  P(x) = y_0L_0(x) + y_1L_1(x) + ... + y_iL_i(x)



Shamir Secret Shares

- Create a set of numbers from **a1** to **a**$_{t-1}$.
- Create the following polynomial of degree (**t - 1**) with constant term **S**:

  P1(x) = S + a1x + a2x^2 **+ ... + a**$_{t-1}$**x**$^{t-1}$
- For each **P1(x)** we evaluate, we receive a data point (called a "share").
- Using interpolation, we create a second polynomial using **t** number of shares.

  P2(x) = y$_1$**L**$_1$**(x) + y**$_2$**L**$_2$**(x) + ... + y**$_t$**L**$_t$**(x)**
- If we evaluate at **P2(0)**, we will return the constant **S**.

Why use Shamir Secret Shares?

- We can imbue a secret **S** to have a desired threshold of **t**.
- We can then split **S** into *any* number of individual shares.
- We can recover **S** using any subset of shares, provided we have **t** shares.

Limitations of Shamir Secret Sharing

- Because **t** shares will reveal **S**, each share must be kept secret.
- The protocol assumes one person will generate all shares.
- We need a polynomial that is shared by a group of adversaries.
- How do we create a group polynomial where nobody knows **S**?

Distributed Key Generation

For each participant :

- Create your own shamir secret with random values and threshold **t**.
- Create a share for each participant **x** in the group (including you).
- Deliver each share to participant **x**, and collect your own shares.
- Sum the collected shares (including your private share) into share **gx**.
- Share **gx** is a part of group polynomial **gP(x)**, which is unknown.

Participants must collect **t** shares to construct **gP**, and recover **gP(0) = gS**.

Attacks on Schnorr Multi-Signatures

- Subset sum attack.
- Wagner's Algorithm.
- ROS Attack.

Development of FROST

- Published by researchers from University of Waterloo.
- Developed by engineers from Zcash and Cloudflare.
- Draft specification available on IETF.
- Rust implementation on ZCash Github, backed by multiple security audits.
- Many test implementations available in the wild.

How FROST Works

FROST defines a safe, secure and efficient protocol for signing a message using schnorr and secret shares.

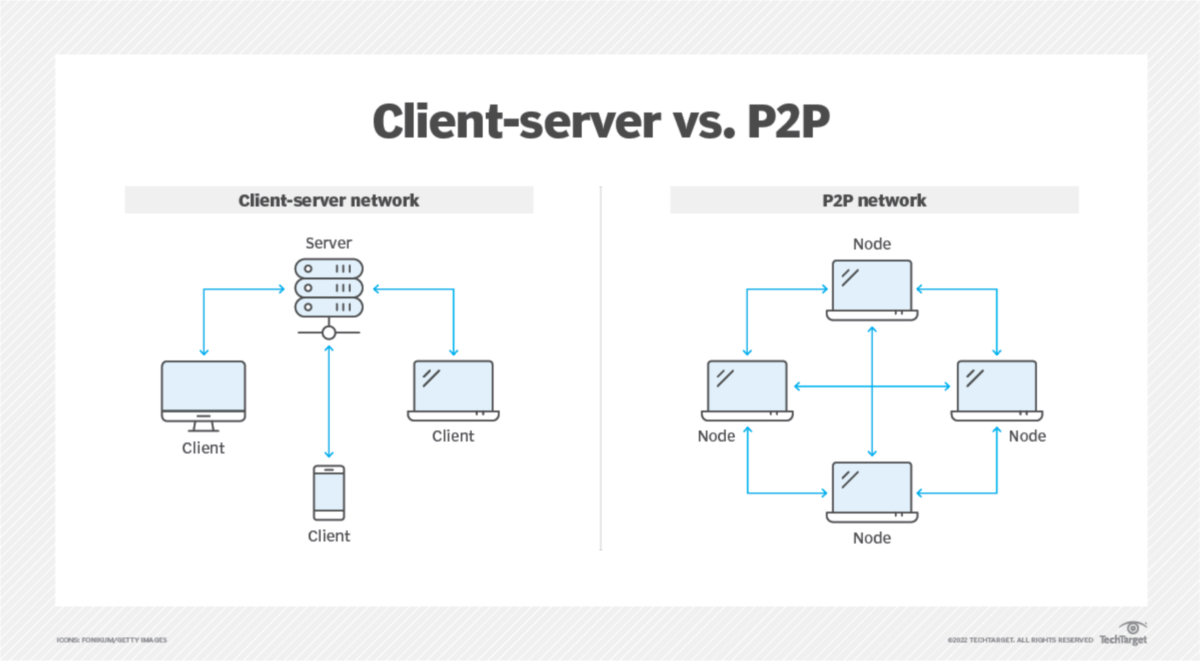
For each participant:

- Step 1: Create and distribute two public nonce values (Round 1).
- Step 2: Collect other nonces, compute the group nonce and challenge.
- Step 3: Sign challenge with your secret share + secret nonce values.
- Step 4: Distribute, collect and verify partial signatures (Round 2).

Finally, combine **t** partial signatures into a complete schnorr signature.

Round 1: Enrollment

- Generate two nonce values: "hidden" nonce and "binding" nonce.
- All participants exchange public nonces with each other.
- Also a good time to exchange shares and commits for DKG.



Computing the Group Nonce for a Signature

- Generate a list of ids (**x**) and public nonces from each participant.
- Concatenate all participant data and hash it (using sha256).
- Concatenate this hash with the group pubkey and message being signed, then hash again. This is the group "prefix".
- For each participant, concatenate their id (**x**) with the prefix, then hash it. This is the participant's "binding factor".
- Tweak the participant's binding nonce value with the binding factor. Then add the tweaked nonce to the hidden nonce.
- Combine all particiant nonces into the group nonce **R**.

Computing the Group Nonce for a Signature

```
# Generate a list of concatenated data from each participant.
commits = [ concat(p.x, p.h_pnonce, p.b_pnonce) for p in participants ]

# Reduce all commitments into a single hash.
c_hash = sha256(...commits)

# Compute the group prefix hash.
prefix  = sha256(group_pk, message, c_hash)

# Compute the binders for each participant.
binders = [ sha256(prefix, p.x) for p in participants ]

# Compute the group nonce value.
group_R = 0
for p in participants:
  group_R += p.h_pnonce
  group_R += (p.b_pnonce + binders[p.x])
```

Why are we doing this?

- The nonce value is the most vulnerable part of a digital signature. It sits between your private key and the world.
- Allowing influence over this nonce is very dangerous. Nonce manipulation is the basis of many different types attacks.
- The group nonce value must *not* be gameable by any participant in the group, in any sort of way.

Creating a Partial Signature

- Taproot: We have to negate our nonce values if the group nonce has an odd y-value.
- Taproot: We also have to calculate the parity and accumulative parity of the group public key when tweaks are applied (see BIP327).
- Taproot: Compute the challenge using BIP340.
- Compute **x** coefficient for the participant's secret share.
- Compute the participant's secret nonce.
- Compute the final signature.

Creating a Partial Signature

```
# Taproot: Negate nonces if needed.
if group_R.y % 2 !== 0:
  h_snonce = N - h_snonce
  b_snonce = N - b_snonce

# Taproot: Set the correct parity for the secret key (BIP327).
sk = group_Q.parity * group_Q.state * share_secret

# Taproot: Format challenge using BIP340.
e = hash340('BIP340/challenge', message, share_pubkey, group_R)

# Compute participant coefficient from x values.
c = interpolate(identifiers, share_x)

# Compute the participant's secret nonce.
k = (b_snonce * bind_factor) + h_snonce

# Compute the participant's partial signature.
psig = (sk * e * c) + k
```

Combining Partial Signatures

- If **t** partial signatures are combined, the result is a valid schnorr signature.
- Any tweaks to the group pubkey must also be applied to the signature.

```
# Taproot: Compute the proper tweak value.
T = challenge * group_Q.parity * tweak

# Sum partial signature values, plus tweak.
s = ps_1 + ps_2 + ... + ps_t + T

# Return the final schnorr signature.
signature = concat(R, s)
```

Protocol Misbehavior

- Participant share contributions must be verified (using VSS).
- Partial signatures must be verified.
- FROST trades robustness for efficiency.
- ROAST offers a more robust version of FROST.

BONUS: Updating shares of a Group Secret

- Set of **t + 1** participants create a new polynomial where **P(0) = 0**.
- Each participant delivers shares of their polynomial using DKG.
- Each participant computes group share **gy**, then sums **gx + gy = gz**
- Because **gPY(0) = 0**, **gPZ(0) = gS**. The group constant does not change.

Based on how **gPY** is constructed, you can add / remove participants, replace shares, change the threshold, and more!

The End

https://github.com/cmdruid/workshops