Post Quantum Key Recovery Strategies

Classical key recovery

When talking about classical key recovery we are talking about a situation where we have a private key *k* which is a number and we want to implement a cryptographic protocol that would allow us to reconstruct or obtain in some other way our private key in the case it's lost.
Some of the approaches we can take include:

- redundant backup locations - save the key in multiple secure locations like encrypted devices or cloud
- hardware recovery - using encrypted devices like USB to hold key accessible by devices built-in mechanisms like biometrics
- encrypt the key with password-based key and store it locally or in the cloud
- use multisig and guardians like some wallets already use (Argent wallet, Loopring wallet)

We will be using social recovery which is a technique in which our private key will be split into multiple shares and distributed to guardians and for the recovery we would need to gather some of the shares from said guardians. Our implementation will be based on Shamir's secret sharing (SSS), so let's take a closer look at SSS.

Shamir's secret sharing

SSS is a (*n*, *t*)-threshold scheme which means that using SSS we can split our secret *S* into *n* shares in such a way that only with combining *t* or more shares can we get the secret *S* back. Any individual or any combination of less than *t* shares isn't sufficient to reconstruct the secret.

Generating the shares

SSS uses polynomials to generate shares. If we want *t* to be a threshold, then we would need to construct a polynomial of degree *t-1*. Let that polynomial be *P(x)*. We will construct *P* in such a way that *P(0) = S*, or in other words *P* will be in the form:

$$P(x) = S + r_1 x + r_2 x^2 + \cdots + r_{t-1} x^{t-1}$$

where r_i is a random number.

Now we can simply calculate shares $S_i, i \in \{1, 2, ..., n\}$ as points $(i, P(i))$.

Example:

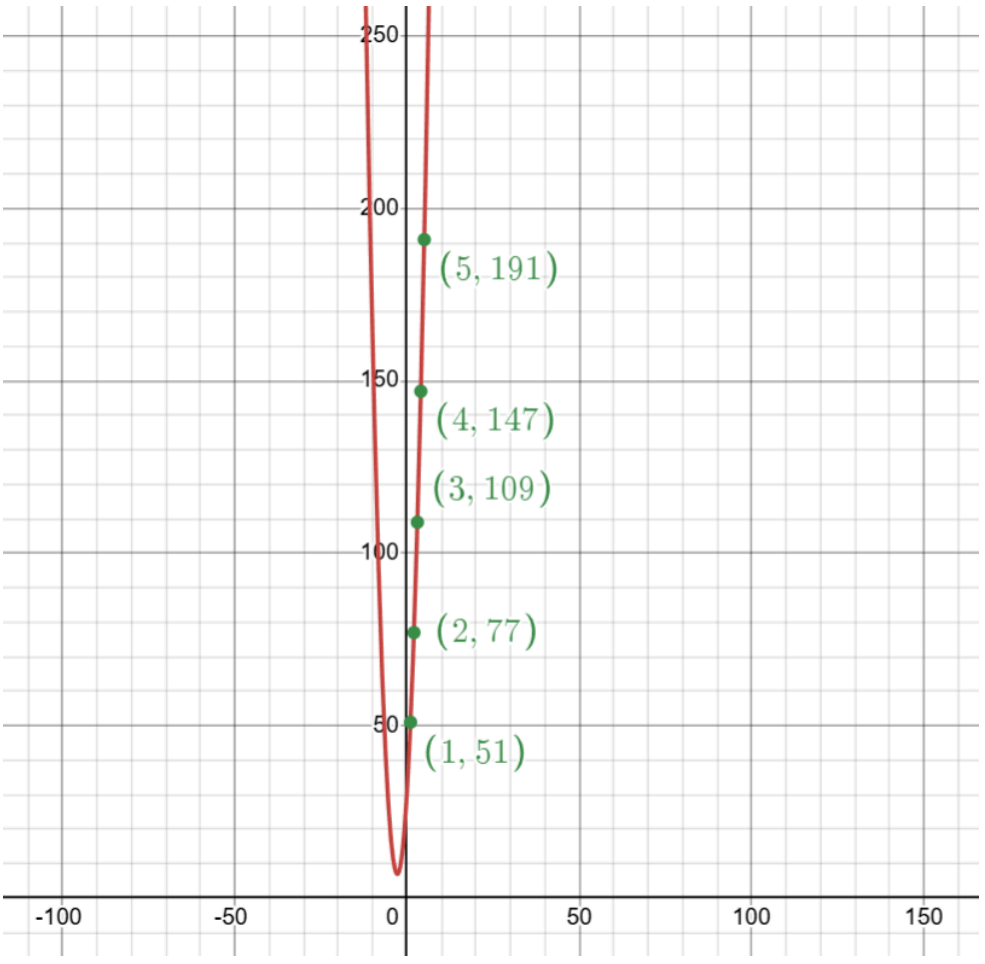
$$t = 3, n = 5, S = 31$$

We randomly form *P:*

$$P(x) = 31 + 17x + 3x^2$$

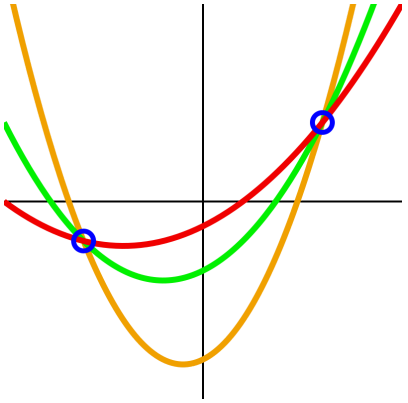
And now the 5 shares we need are:

$$S_1 = (1, P(1)) = (1, 51)$$
$$S_2 = (2, P(2)) = (2, 77)$$
$$S_3 = (3, P(3)) = (3, 109)$$
$$S_4 = (4, P(4)) = (4, 147)$$
$$S_5 = (5, P(5)) = (5, 191)$$



Secret reconstruction

SSS uses polynomial interpolation to reconst the secret. SSS relies on the fact that a polynomial of degree *t* isn't uniquely identified with *t* or less points; only with *t+1* points can we construct one polynomial that passes through all the points (see a picture below; we can fit many parabolas through 2 points, but with 3 point there is only one parabola that passes through all 3 of them)



source: wikipedia

We can use Lagrange interpolation to reconstruct a polynomial *P(x)* used to generate the shares. Let *k* be the number of shares at our disposal. In order to correctly interpolate a polynomial $t \leq k \leq n$ inequality needs to hold. *P(x)* can be calculated using the formula:

$$P(x) = \sum_{j=1}^{k} y_j \prod_{m=1 \wedge m \neq j}^{k} \frac{x - x_m}{x_j - x_m}$$

Now the secret will simply be:

$$S = P(0)$$

Computing the polynomial *P* can be expensive because it involves many polynomial multiplications, so there is an optimized, direct version of the formula to calculate just *S*. It's simply derived by replacing the value of $x = 0$ into the interpolation formula:

$$S = \sum_{j=1}^{k} y_j \prod_{m=1 \wedge m \neq j}^{k} \frac{x_m}{x_m - x_j}$$

Example:

Given shares S_1, S_2 and S_5 from the previous example, reconstruct the secret *S*.

We can use the formula to easily calculate the secret:

$$S = 51\frac{2}{2-1}\frac{5}{5-1} + 77\frac{1}{1-2}\frac{5}{5-2} + 191\frac{1}{1-5}\frac{2}{2-5} = 31$$

And we got the correct value of the secret.

Lets what happens we have less than 3 shares.
Given shares S_1 and S_2 from the previous example, reconstruct the secret S.

$$S = 51\frac{2}{2-1} + 77\frac{1}{1-2} = 25 \neq 31$$

We got the incorrect value of the secret.
As you can see SSS is an extremely simple scheme for sharing a secret.

Security analysis

SSS relies on polynomials and polynomial interpolation so it's resistant to both classical and quantum attacks, but using it over a field of real numbers can give some information about S even if the threshold isn't being satisfied (see an example). In order to solve that problem it's better to use an SSS polynomial over finite fields like Z_p where p is a large prime number.
SSS, just like all other threshold schemas, can be vulnerable to a collusion attack. In this attack shareholders (guardians) can form an alliance and combine their shares to find out the secret. SSS has no mechanisms to prevent or detect this. Some of precautions to take in order to try to prevent this attack are:

- Choosing $t \geq ceil(\frac{n}{2})$
- Choosing large enough n and in terms choosing enough guarding
- Giving shares (role of a guardian) to people and institutions you trust

If some guardians are malicious they could substitute share with some random number from the field effectively disrupting the reconstruction process which would yield incorrect value for the secret. SSS has no way of detecting fake shares and there is no way to prevent or detect them without saving the shares which would defeat the whole point of key recovery protocol.

Designing a Secure and Quantum-Resistant Protocol

Before delving into the details of our protocol, it is essential to outline the foundational principles and requirements that guided its design. These principles ensure the protocol's robustness, security, and practicality, addressing the limitations of traditional secret sharing methods.

1. **Statelessness:**
   The protocol must not require the secret owner to store or remember any additional data (e.g., the secret or auxiliary values) apart from eventually their chosen password.

2. **Post-Quantum Security:**
   The protocol should use cryptographic primitives resistant to both classical and quantum attacks, ensuring long-term security against emerging threats.
3. **Share Verifiability:**
   The protocol must allow verification of the integrity of shares. This prevents tampering and ensures successful reconstruction.
4. **Collusion Resistance (optional):**
   The design must mitigate the risks of collusion among shareholders by incorporating mechanisms to make unauthorized reconstruction infeasible.
5. **Efficiency and Practicality:**
   The protocol should be computationally efficient, making it feasible for real-world applications without requiring extensive resources or specialized hardware.
6. **Interoperability:**
   The protocol should integrate seamlessly with existing cryptographic systems and applications, such as cryptocurrency wallets, key management solutions, and secure storage platforms.

Our secret sharing protocol - high level

With these design goals in mind, we propose a novel protocol that enhances traditional secret sharing schemes while addressing their key limitations. Our protocol will have a few steps which we will first explain on a high level and then our approach to every step. Here are the steps that we created in out protocol:

1. Initialization - this step is responsible for initialization of all parameters using in the scheme like t and n for generating the shares and password P (more on those lates)
2. Commitment to the secret - in this step we need to create a public commitment to the secret S which we will denote as $C(S)$. This commitment should bind the secret in such a way that only someone who knows the secret can compute this commitment and commitment should not reveal any information about the secret (it should be impossible to get the secret using the commitment)
3. Generating the shares - in this step we are generating n shares that are going to be given to the guardians using a secret sharing scheme like SSS

4. Generating the verification tags - this step is responsible for generating information that is needed in order to verify that the shares haven't been tempered with
5. Verifying the shares - before the reconstruction of the secret we verify that all the shares used in reconstruction are valid and haven't been tempered
6. Reconstring the secret - in the final step we reconstruct the secret using given shares and verify the secret by recalculating the commitment and checking if it matches the original commitment

Now we are going to see examples of implementation for every step. Note that the mechanisms we suggest and implement are not the only viable solutions. Concrete implementation of every step we are just presented can and should be switched to fulfill the needs of the concrete problem they are being used on with only limitation that they follow the requirements stated above.

Initialization

This is a really simple and straightforward step. We just need to agree and set the parameters used in the next phases of the protocol. In our implementation we are going to need t and n for SSS. They are used just as we explained in the SSS section. n is the number of shares and t is the threshold - the number of shares needed to reconstruct the secret. Another parameter we will need is a password P. This will be the only secret piece of information that the secret owner should remember and save and it will be used to sign and verify tags.

Commitment

The commitment step plays a crucial role in ensuring the security and integrity of the protocol. The commitment is a cryptographic value generated from the secret and is shared publicly. It serves as a binding and hiding mechanism that guarantees the integrity of the secret and its corresponding shares.

Why is the commitment needed?

1. Detecting Tampering:
   Without a commitment, there is no way to verify whether a shareholder has provided a valid share during the reconstruction process. Malicious shareholders could supply invalid or fake shares, causing the reconstruction to fail or yield an incorrect secret. The commitment allows the system to verify the validity of shares before reconstruction.

2. Preserving Statelessness:
   The commitment enables the secret owner to generate verifiable shares without needing to remember the secret or store auxiliary data
3. Preventing Tempering:
   The commitment alongside password *P* acts as a safeguard against tampering with the shares by ensuring that only shares consistent with the original secret are accepted during reconstruction. Shareholders cannot manipulate their shares to form a new secret without breaking the cryptographic security of the commitment.

How does commitment work in this protocol?

The secret S is combined with itself using a quantum-resistant hash function to create the commitment:

$$C(S) = H(S \oplus H(S))$$

Symbol $\oplus$ denotes the XOR operation which is used to introduce additional confusion and diffusion. *H* is a quantum resistant hash function (SHA-3 (Keccak), BLAKE2, SHA-512/256), and *H(S)* is a hash of the secret.
Before the XOR we must ensure that *S* and *H(S)* are of the same length either by padding the shorter one, truncating the longer one of the two or usings some expanding mechanism to avoid lots of zeros.

This ensures that the commitment is both cryptographically secure and unique to S.

The commitment is public information so it can be safely stored and distributed. Some approaches can be publishing to a blockchain registry, saving it locally or in the cloud and distributing it to guardians alongside shares.

Generating the shares

We are using original SSS to generates shares S_i for $i \in N_n$.
In order to prevent the collusion attack we could try encrypting the secret with some private key or password but that would require us to store additional information which would break the stateless principle of this protocol. Additionally if that key is lost we could never recover the secret.

Generating verification tags

This is a crucial part of the antitempering mechanism. For each share S_i we generate corresponding tag T_i. When a secret owner requests a share from the guardian, guardian is sending both the share S_i and tag T_i that secret owner will use before reconstruction the secret *S* to ensure validity and integrity of the share.
Tag is computed by following formula:

$$T_i = H(C(S),\ S_i,\ KDF(P,\ C(S))$$

H is a quantum resistant hash function as previously explained.
C(S) is commitment calculated in previous phase
S_i is the share of the secret *S* that is given to the *i*-th guardian
KDF(P) is a key derivation function.
A key derivation function is a cryptographic algorithm that derives one or more secret values from a common master key (usually a password or passphrase as in our case). We are more interested in its second use, password hashing. We are going to use Argon2 as the current standard. The main idea behind this part of the formula is to make brute force attack harder. Password *P* isn't meant to be secure, it's meant to be a private piece of information used for signing and verifying tags that is easy to remember so we are using KDF to strengthen it and make it harder to brute force.
Using this private password *P* allows only the secret owner to generate the tags. Since both commitment (it's public) and share are known to the guardian, nothing would prevent him to fake share S'_i and generate tag T'_i which would pass the validation.
The KDF uses public information to prevent the preimage attack using rainbow tables. Generating random salt would require us to store more information, but there is a workaround. We could use the commitment *C(S)* or its hash *H(C(S))* as the salt. Since commitment is already public and generated for every secret S it fits all criteria to be used as salt.

Verifing the shares

When secret needs to be reconstructed, secret owner will collect shares and their corresponding tags from the guardians, but before the secret is reconstructed the owner will validate all tags by recomputing them from the share provided by the

guardian S_i , public commitment *C(S)* and private password *P*. Only if the tag passes the validation share will be used in the reconstruction process, otherwise the share can't be trusted and is automatically dropped.
If the password *P* is lost this step could be omitted knowing that the reconstructed secret could be invalid.

Reconstruction

Secret *S* is reconstructed from the shares that pass the validation phase only if there are enough of them (at least *t*) using SSS reconstruction algorithm. After calculating the secret the final step is to recalculate the commitment using newly reconstructed secret and checking if it matches with the publicly available commitment generated with true original secret. This matching could be skipped if somehow the commitment is lost which would mean that constructed sacred couldn't be verified.

Conclusions

The proposed commitment scheme using $C(S) = H(S \oplus H(S))$ and a *KDF* for tag generation provides a secure, efficient, and verifiable way to commit to a secret without needing to store the secret or additional information. The use of a cryptographic hash function ensures that the scheme is both classically secure and quantum-resistant, while the KDF provides a flexible method for tag generation.
The key benefits of this scheme are:

- Quantum Resistance: The scheme relies on quantum-resistant hash functions for its security.
- Minimal Data Storage: The secret owner does not need to store additional data besides the secret and the password.
- Verifiability: The secret owner can verify that shares haven't been tampered with and avoid using invalid shares in reconstruction process in order not to get incorrect secret

Vulnerabilities of this scheme are:

- The collusion attack hasn't been dealt with and still represents a risk for the user.
- Using a weak password can be a vector to attack by brute forcing the password. If a guardian or group of guardians find out the password *P*

they will be able to modify shares and generate valid tags that would not be detected by the secret owner. This way, if a sufficient number of malicious guardians collude together, they would be able to completely change the secret. That is why it's important to use commitment *C(S)* to check the validity of a reconstructed secret.

- Once the shares are distributed to the guardians there is no way to deactivate the share and remove the guardian which could be a risk in the long run

This protocol is very flexible because it does not specify any relation between secret owner and guardians allowing applications using this protocol to decide what fits them best within in turn allows this scheme to be used in various scenarios requiring secret sharing with verifiable, quantum-resistant commitments, such as in distributed key management or cryptographic protocols.

PQSAP Key Recovery

When dealing with Post Quantum Stealth Address Protocol, the private key that needs to be reconstructed isn't a number from the finite field but it is a vector of polynomials. We could use the previously described protocol with small changes to implementations.
In the case of Kyber512 our secret *s* will be a vector of 2 polynomials. For Kyber768 there are 3 polynomials, and for Kyber1024 4 polynomials. Since those numbers are relatively small, we can just assume that if we can use a polynomial as a secret we can just repeat that operation a few more times and as a resulting share get a vector of shares of polynomial split.
Now we have reduced our problem to using a polynomial as a secret, but since a polynomial can be represented as a vector of its coefficients we can just use SSS on each coefficient individually and combine it back to a vector of shares.
We can define this operation as a matrix multiplication in the following way:
Let our polynomial secret be

$$s = c_0 + c_1 x + c_2 x^2 + \ \dots \ + c_n x^n = (c_0, c_1, \dots, c_n)$$

Now, just like we defined a random polynomial in classical key recovery from previous section we will define a random matrix *M* with *n+1* rows and *t* columns like this:

$$\begin{bmatrix} c_0 & r_{0,1} & \cdots & r_{0,t} \\ c_1 & r_{1,1} & & r_{1,t} \\ \vdots & & \ddots & \vdots \\ c_n & r_{n,1} & \cdots & r_{n,t} \end{bmatrix}_{(n+1)\times t}$$

Where $r_{i,j}$, $i \in \{0, \ldots, n\} \wedge j \in \{1, \ldots, t\}$ is a random number from the field.

Now we can simply define our share as:

$$S(x) = \begin{bmatrix} c_0 & r_{0,1} & \cdots & r_{0,t} \\ c_1 & r_{1,1} & & r_{1,t} \\ \vdots & & \ddots & \vdots \\ c_n & r_{n,1} & \cdots & r_{n,t} \end{bmatrix} \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^t \end{bmatrix} = \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ \vdots \\ s_n \end{bmatrix}$$

Other way

Previously defined method is more mathematical, but another approach we could take is to take the advantage of the existing Kyber implementations. Kyber KeyGen returns a byte array as a private key, so we can just interpret those bytes as integers and use a key recovery strategy defined for scalars.
One problem that may arise in this approach is the size of the integer we would get as well as choosing a prime number for the field. One possible solution is to combine two discussed methods. Since Kyber uses *q=3329* as its prime, we could use it as well and then split the byte array into chunks and use it as a vector of integers. Since we want our secret to be inside the field Z_q we can split the array in chunks of up to 11 bits (since $2^{11} = 2048$). If we want to save space we could use 10 or 11 bits, but for the simplicity of implementation we suggest one byte.

Commitment

We also need to change the commitment, since we are not using large numbers anymore.
One way could be to use the same commitment for each component of the vector and then combine them into a single commitment using something like Merkle tree, but since the coefficients are not very large numbers XOR operation wouldn't be the best choice.

Another approach would be to combine all coefficients binary form into a single large byte array (reverse of the process defined previously) and then using that byte array as secret since both hash and XOR are valid operations on bytes.