# A Decentralized Mnemonic Backup System

PriFi Labs Inc.

## 1 Introduction

*"Not Your Keys, not Your Coins."*

This crypto mantra, popularized by *Andreas Antonopoulos*, encourages us, crypto enthusiasts, to use non-custodial crypto wallets in which the private keys are directly stored on their device. However, using non-custodial wallets comes with the risk of losing all of our crypto assets if we lose our private keys. If our device gets lost, stolen or irremediably break down, our crypto assets might be locked forever [1].

Fortunately, most non-custodial wallets such as Exodus (Bitcoin), Metamask (Ethereum) and Keplr (Secret Network) offer a way to recover our private keys by the mean of a "recovery phrase" also known as "mnemonic seed phrase". It is the 12-word phrase that we are being given the first time we setup our wallet. Here is an example of such a phrase:

```
witch fox practice feed shame open
despair creek road again ice least
```

That phrase is important as it is used to generate the same private keys on demand (BIP39 standard [2]). We can use them as a backup or to import our wallet into a new device. Indeed, this mnemonic phrase is a really sensitive piece of information. Anyone who has access to this phrase would have full control over our crypto assets as explained in the [Metamask FAQ page][3]:

*"MetaMask requires that you store your Secret Recovery Phrase in a safe place. It is the only way to recover your funds should your device*

> *crash or your browser reset. We recommend you to write it down. The most common method is to write your 12-word phrase on a piece of paper and store it safely in a place where only you have access. Note: if you lose your Secret Recovery Phrase, MetaMask cant help you recover your wallet. Never give you Secret Recovery Phrase or your private key(s) to anyone or any site, unless you want them to have full control over your funds."*

As written here, we are not supposed to remember that phrase like a password. Instead, it is recommended to write it down on a piece of paper and store it in a "safe place". But where is that "safe place" exactly? In our physical wallet with our cash and credit cards? Not ideal as it could get lost or stolen. Inside our home? No ideal again as it could burn down. A safe deposit box in the vault of a bank? it might be the best solution after all but the logistics that comes with it makes that solution very cumbersome.

In the end, it is quite inconvenient to store physical objects safely. So, could we design a simple application that would take custody of my passphrase and would allow me to recover by simply using my email? Yes we could build such an application, it would be something similar to a password manager such as *1password* [4] for instance. However, such a solution requires 1) that the service provider is trustworthy an 2) that the whole application is secured. Indeed, such a centralized solution is not a good solution as it goes against the philosophy of decentralized application.

In this paper, we propose a Decentralized Mnemonic Backup system that anyone can use to give custody to any blockchain passphrase to the Secret Network [5] and recover it using a simple email. To better explain our idea, we will go through 3 iterations each more secure than the previous one. The first iteration (section 2) is meant to be simple to capture the user experience however that first design is rather naive and not very secured. In the second iteration (section 3), we harden the security by encrypting the passphrase and separating the key management from the passphrase storage. That second iteration makes use of the well know DiffieHellman key exchange protocol to securely generate the encryption key on the blockchain. Finally in the last iteration (section 4), we will split and distributed the encrypted passphrase across the network by using another cryptographic protocol called the Shamir's Secret Sharing scheme.

# 2 Iteration 1: The User Experience

Alice is a crypto enthusiast that holds all kind of crypto assets: BTC, ETH, SCRT and others. She would like to have an online backup of all of her wallets' passphrases in case her device gets lost, stolen or irremediably breaks down. When that doomsday comes, she should be able to recover her passphrases easily. The overall user experience is rather simple:

- **When Alice wants to backup a passphrase**, she visits our Mnemonic Backup website and enters a passphrase and her email. After submitting her information, our application sends her an email with a confirmation code that she should enter to finalize the backup process.
- **When Alice wants to recover a passphrase**, she visits our Mnemonic Backup website and enters her email. Our application sends her an email with a verification code that she should enter before getting her passphrase back.

This user experience is similar to existing security mechanisms used in traditional web applications where we must make sure that the user is the legitimate owner of the email address used for signing up, signing in (with two factor authentication) or reseting a password.

## 2.1 Architecture

Our Mnemonic Backup System is a decentralized application on the Secret Network. One of the greatest feature of the Secret Smart Contracts is that they enable storing private data on the blockchain. So for this first iteration, we implement our Mnemonic Backup System as a Secret contract that records the passphrase when backing up and restores that passphrase when recovering. However, our secret contract cannot send emails by itself, so we are pairing it with an off-chain mailer that will send emails to the users.

The figure 1 shows the three entities of our system:

- **The Secret Recovery Contract** is the Secret contract that stores the users email and passphrases (first iteration only, it will change in our second iteration)

- **The Frontend** is the webpage that allows users to backup and recover passphrases. The frontend will interact with the Recovery Secret Contract exclusively.
- **The Mailer Backend** is the off-chain mailer that sends confirmation/verification code the the users. In a nutshell, it will be an event handler listening for transactions sent to the Recovery Secret Contract. For security reasons, the mailer backend will never handle any passphrase. Only the Secret Recovery Contract will have access to passphrases.

## 2.2 The Protocol

**Backup** During backup (see figure 1), the goal is to verify Alice's email address to eventually store her passphrase.

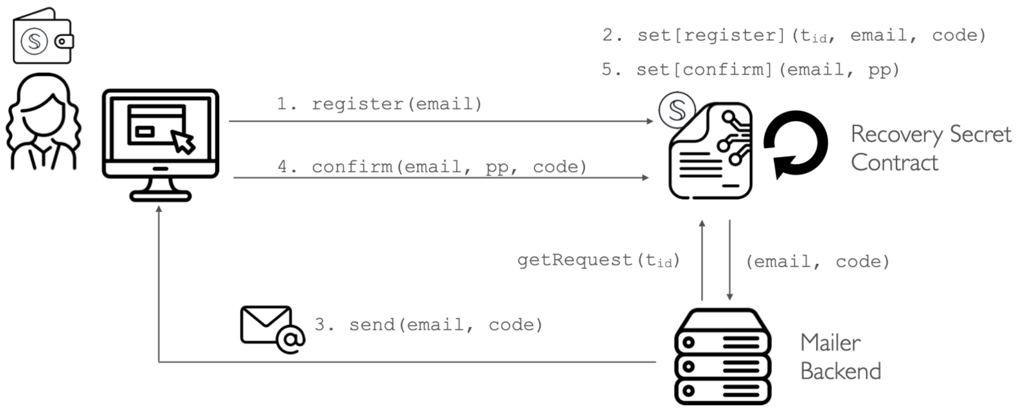

Figure 1: Iteration 1 - Backup

1. **Request** - Alice enters her *email* and, after pressing the submit button, a script running inside the webpage sends a transaction register($email$) to the Secret Network using a throwaway Secret account that Alice has just provisioned.
2. **Registration** - The Secret Recovery Contract generates a random confirmation code and stores the record $(t_{id}, email, code)$ in the `register` dataset.

3. **Emailing** - Once the request transaction has been validated, an event handler is triggered on the Mailer Backend that queries the Recovery Secret Contract for the email and confirmation code associated with the transaction id. Thus, the Mailer Backend sends the confirmation code to Alice by email.

4. **Confirmation** - Now Alice sees a webpage that asks her to enter the confirmation code sent by email and her passphrase *pp*. She opens her email and copies'n paste the confirmation code into her browser. After pressing the submit button, a script running inside the webpage sends a transaction confirm$(email, pp, code)$ to the Recovery Secret Contract.

5. **Recording** - The Secret Recovery Contract verifies the verification and stores the record $(email, pp)$ in the `confirm` dataset.

**Recovery** During recovery (see figure 2), the goal is to verify Alice's email address to eventually send her passphrase back.

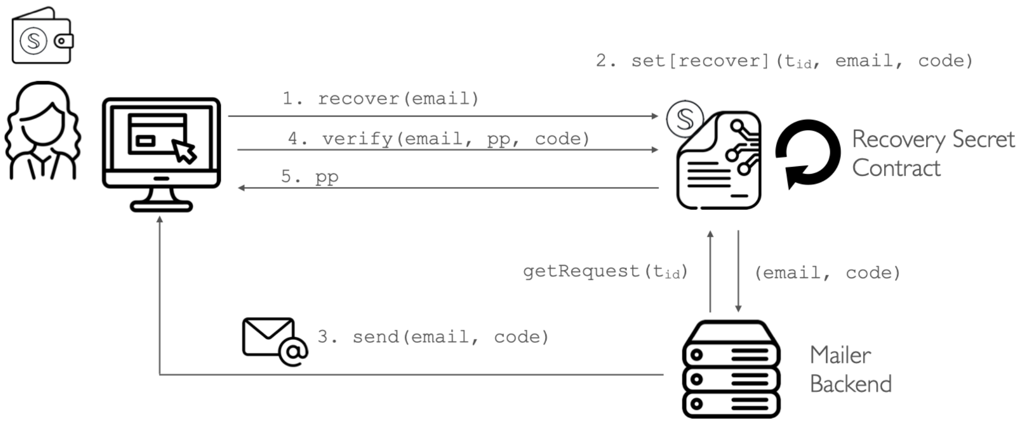

Figure 2: Iteration 1 - Recovery

1. **Request** - Alice enters her *email* and, after pressing the submit button, a script running inside the webpage sends a transaction recover$(email)$ to the Recovery Secret Contract to the Secret Network using a throw-away Secret account that Alice has just provisioned.

2. **Registration** - The Secret Recovery Contract generates a random verification code and stores the record $(t_{id}, email, code)$ in the `recover` dataset.

3. **Emailing** - Once the request transaction has been validated, an event handler is triggered on the Mailer Backend that queries the Recovery Secret Contract for the email and verification code associated with the transaction id. Thus, the Mailer Backend sends the verification code to Alice by email.

4. **Verification** - Now, Alice sees a webpage that asks her to enter the verification code sent by email. She opens her email and copies'n paste that verification code into her browser. After pressing the submit button, a script running inside the page sends a query $\mathsf{verify}(email, code)$ to the Recovery Secret Contract.

5. **Response** - The Secret Recovery Contract verifies the verification code, retrieves the corresponding record from the `recover` dataset and returns the passphrase.

## 2.3 Security Analysis

Although, this first iteration captures the user experience that we want, it is unfortunately not satisfactory in terms of security. There are three single points of failure: 1) Alice's account, 2) the Mailer Backend and 3) the Secret Recovery Contract.

What if Alice's account is hacked? As explained earlier, it is recommended that Alice uses a throwaway Secret account when registering a passphrase. As soon as the passphrase has been recorded, she can forget about this account since she will not need it to recover her passphrase. However, what if the account's private key is leaked somehow? It would be bad since the attacker with this key could look into the transaction history and get the passphrase.

What if the Mailer Backend is hacked? Let us consider the two phases: backup and recovery. During the backup phase, the attacker could use the confirmation code to upload an arbitrary passphrase for Alice. This is a problem if Alice recovers what she believes is her original passphrase but is in fact the passphrase to an account that the attacker can access as well. Then, any new asset Alice puts in this account from now on can be stolen by the attacker. During recovery phase it is even worst since the attacker could use the verification code to query the Recovery Secret Contract directly and get the passphrase back.

What if the Recovery Secret Contract is hacked? This is by far the biggest threat here since the attacker will have access to all users' passphrases.

Indeed, all of these security threats are not acceptable and we are going to address all of these issues by hardening our protocol in the next iteration.

## 3 Iteration 2: Security Hardening

As discussed earlier, we should not store all users' passphrases in the same Secret contract in case that contract gets breached. Actually, it would be safer to store each passphrase in its own contract and have those contracts be owned by different accounts. So, for this second iteration, let us try a different approach. Let us have Alice creating a throwaway Secret account to mint a Secret NFT embedding the passphrase as a secret metadata. Then, she sends that NFT to her friend Charlie as a way to backup her passphrase. When the doomsday comes, she could recover her passphrase by asking Charlie to send her NFT back. Indeed, this approach is not secure since Charlie could access the passphrase as soon as he owns the NFT. A solution to that is to have Alice encrypting her passphrase when minting the NFT so that Charlie cannot decrypt without the key. However, we have another problem now, where to store the key when Alice needs it to recover her passphrase? We could actually adopt the same approach as describe in iteration 1. Instead of storing all passphrases in the Recovery Smart Contract, we could store all users' encryption keys. But, did not we say that this was insecure? It is more secure actually. While it is true that all the keys would be exposed if the contract is breached, however, those keys are now worthless if the attacker is not able to locate and regain those NFTs one by one to extract each passphrase. Our system is more secure since breaking those passphrases is now lot harder for the attacker.

There were two other problems we identified in iteration 1 that do not go away with our new NFT approach. First, if an attacker could hacks into Alice's backup account, he could recover the key sent to the Recovery Secret Contract. Secondly, if an attacker could hacks into the mailer backup, he could also recover the key by getting the verification code. In this approach we fix those two problems by implementing the Diffie-Hellman Key Exchange key protocol on the Secret blockchain. The goal is to have Alice's account and the Recover Secret Contract negotiate a key without exchanging it explicitly during the backup phase.

## 3.1 The Diffie-Hellman Key Exchange Protocol

The *DiffieHellman Key Exchange Protocol* is a cryptography protocol that allows two parties, usually named Alice and Bob, that have no prior knowledge of each other, to securely agree on a shared key over an insecure channel. That channel is considered as insecure because we assume that an attacker, usually named Eve, can eavesdrop the communication and read all messages send back and forth between Alice and Bob.

The algorithm relies on modular arithmetics but let us abstract the mathematical details by simply saying that Alice generates two numbers: one "secret" $sec_A$ that Alice will keep and another one "public" $pub_A$ that Alice can send to Bob over the insecure channel. When Bob receives Alice's public number, he will also generates its own pair $(sec_B, pub_b)$ and send his public number back to Alice. Once the public numbers have been exchanged, Alice and Bob can generate the same shared secret value $s$. Alice computes $s = \mathsf{ECDH}(sec_A, pub_A, pub_B)$ and Bob computes the same shared secret value $s = \mathsf{ECDH}(sec_B, pub_B, pub_A)$. The security of the protocol reside in the fact that it is impossible for Eve to compute that secret value $s$ even if she knows $pub_A$ and $pub_B$ because she does not know either $sec_A$ or $sec_B$.

In practice, this shared secret value is usually not used as a cryptographic key directly. Instead, that shared value is given as input of a key derivation function such as PBFK2 that will generate the same cryptographic key based based on the secret value.

## 3.2 The Protocol

Moreover, we are adding two additional features to our NFT-based Mnemonic Backup System. First, Alice can now backup several passphrases and still recover each of them with the same email. To do so, she will attach a unique id with each NFT during backup and ask the Recovery Smart Contract for that specific id when she needs to recover the associated key. Secondly, Alice has now a limited time to enter her confirmation code during backup or validation code during recovery. This time limit is calculated based on the number of blocks that have been validated since Alice has made her original request to the Secret Recovery Contract.

All of those new features are detailed in the new version of the protocol shown in figures 3 and 4.

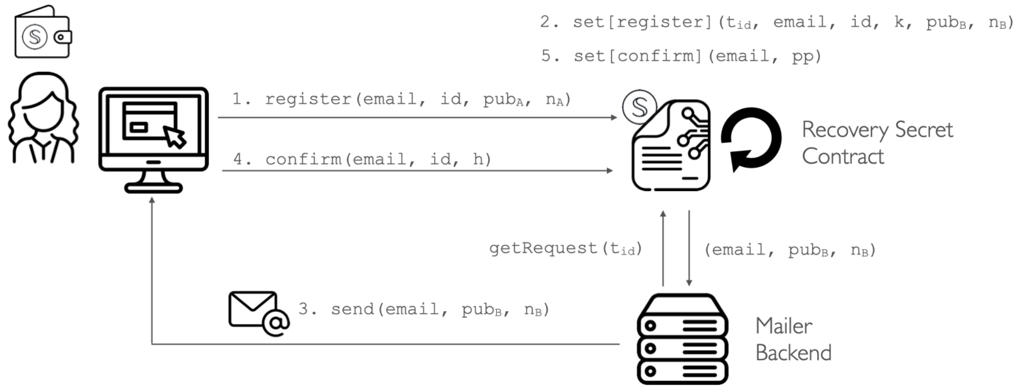
2. set[register](t_id, email, id, k, pub_B, n_B)
5. set[confirm](email, pp)
1. register(email, id, pub_A, n_A)
4. confirm(email, id, h)
Recovery Secret Contract
getRequest(t_id)
(email, pub_B, n_B)
3. send(email, pub_B, n_B)
Mailer Backend

Figure 3: Iteration 2 - Backup

### Backup

1. **Request** - Alice enters her *email*, her passphrase $m$, an *id* for her passphrase. After pressing the submit button, a script running inside the webpage generates an ECDH private and public key pair $(sec_A, pub_A)$, a nonce $n_A$ and sends the transaction $\mathsf{request}(email, id, pub_A, n_A)$ to the Secret Network using the throwaway Secret account that Alice has just provisioned.

2. **Registration** - The Secret Recovery Contract generates a new ECDH private and public key pair $(sec_B, pub_B)$ and a nonce $n_B$. It calculates the ECDH secret $s = \mathsf{ECDH}(sec_B, pub_B, pub_A)$ and derives the 128-bit AES symmetric key $k$ using the standard password-based key derivation function $\mathsf{PBFK2}$ and the concatenation of $n_A$ and $n_B$ as a salt $k = \mathsf{PBKDF2}(s, n_A || n_B)$. Finally, the contract stores the record $(email, id, t_{id}, b_{id}, pub_B, n_B)$ in the `register` dataset.

3. **Emailing** - Once the request transaction has been validated, an event handler is triggered on the Mailer Backend that queries the Recovery Secret Contract $\mathsf{info}()$. The Secret Recovery Contract checks that the query comes from the Mailer Backend address, retrieves the record from the `register` dataset and checks that the query has not expire based on the initial block id $b_{id}$ and the current block id on the Secret Network. If not, it returns the *email*, the public key pair $pub_B$ and the nonce $n_B$. Finally, the mailer daemon sends an email to Alice with the confirmation code $(pub_B, n_B)$.

4. **Confirmation** - Now Alice sees a webpage that asks her to enter the confirmation code sent by email and her passphrase $pp$. She opens her email and copies'n paste the confirmation code into her browser. After pressing the submit button, a script running inside the webpage calculates the ECDH secret $s = \mathsf{ECDH}(sec_A, pub_A, pub_B)$ and derives the 128-bit AES symmetric key $k$ using $\mathsf{PBFK2}$ and the concatenation of $n_A$ and $n_B$ as a salt $k = \mathsf{PBKDF2}(s, n_A || n_B)$. Then, it calculates a hash-based message authentication code (abbreviated HMAC) using the key $k$ and the concatenation of the email and id $h = \mathsf{HMAC}(k, email || id)$. Finally, the script sends the transaction $\mathsf{confirm}(email, id, h)$ to the Secret Network.

5. **Recording** - The Secret Recovery Contract checks that the query has not expire based on the initial block id $b_{id}$ and the current block id on the Secret Network. Then, it calculates the HMAC of the concatenation of the email and id using the key $h = \mathsf{HMAC}(k, email || id)$ and checks that this hash is strictly equal to the hash $h$ in the request. Finally, the contract stores the tuple $(email, id, k)$ in the `confirm` dataset.

6. **NFT minting** - The client-side script can now encrypt (using AES in GCM mode) the passphrase $m$ with the key $k$ to obtain the ciphertext

$c = E_{\mathsf{AES}}(k, m)$. It mints a Secret NFT with the concatenation of the id and the ciphertext as private metada.

Once the protocol is completed, Alice can transfer the NFT to Charlie.

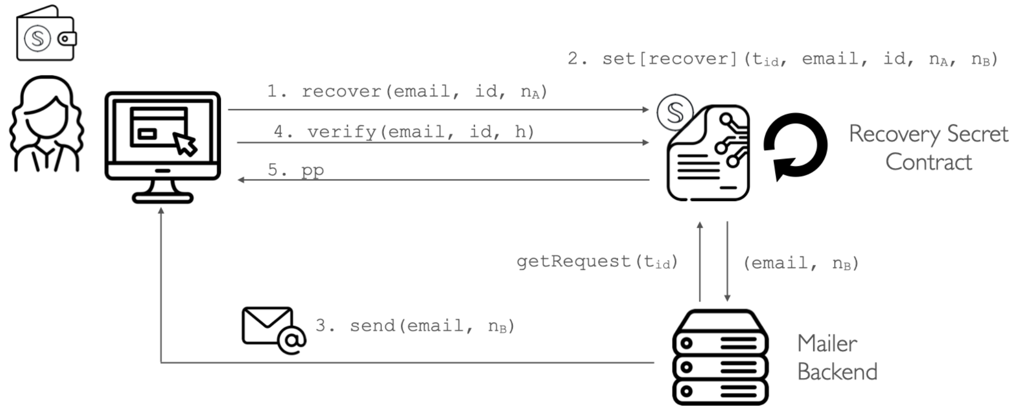

Figure 4: Iteration 2 - Recovery

**Recovery**

1. **Request** - Alice creates a new Secret Wallet and asks her friend to send the NFT back. Then, she goes to the recovery page that asks her to select the NFT. A script running inside the webpage extracts the *id* and the ciphertext from the NFT. Finally, it generates a nonce $n_A$ and sends the transaction $\mathsf{request}(email, id, pub_A, n_A)$ to the Secret Network.

2. **Registration** - The Secret Recovery Contract retrieves the record from the `confirm` dataset using the email and id from the transaction. Then, it generates a nonce $n_B$ and stores the record $(email, id, t_{id}, b_{id}, n_B)$ in the `register` dataset.

3. **Emailing** - Once the request transaction has been validated, an event handler is triggered on the Mailer Backend that queries the Recovery Secret Contract for the email and verification code associated with the transaction id. The Secret Recovery Contract checks that the query comes the Mailer Backend and retrieves the record from the `register`

dataset and checks that the query has not expire based on the initial block id $b_{id}$ and the current block id on the Secret Network. Then, it returns the *email* and the nonce $n_B$. Finally, the mailer daemon sends an email to Alice with the verification code $n_B$.

4. **Verification** - Now, Alice sees a webpage that asks her to enter the verification code sent by email. She opens her email and copies'n paste that verification code into her browser. After pressing the submit button, a script running inside the page calculates the hash of the concatenation of the two nonces $h = \mathsf{H}_{\mathsf{SHA512}}(n_A||n_B)$. The script sends a query $\mathsf{verify}(email, id, h)$ to the Secret Network.

5. **Response** - The Secret Recovery Contract checks that the query has not expire based on the initial block id $b_{id}$ and the current block id on the Secret Network. Then, it calculates the hash of the concatenation of the two nonces $h = \mathsf{H}_{\mathsf{SHA512}}(n_A||n_B)$ and checks that this hash is strictly equal to the hash $h$ in the request. Finally, it returns the key $k$ back to the client

6. **Passphrase Decryption** - After receiving the key back, the client-side script can now decrypt the ciphertext $c$ to obtain the passphrase $m = D_{\mathsf{AES}}(k, c)$.

Once the protocol is completed, Alice sees her passphrase on the webpage.

## 3.3 Security Analysis

This second iteration of our Mnemonic Backup System is more secured. Let us discuss each entity in our architecture.

What if Alice's account is hacked? As explained earlier, it is recommended that Alice uses a throwaway Secret account when registering a passphrase. However, if the account's private key was leaked, an attacker could know to which address the NFT has been transfered but could not retrieve the encryption key with the information recorded in the different transactions.

What if the Mailer Backend is hacked? Let us consider the two phases: backup and recovery. During the backup phase, the attacker could not use the confirmation to upload an arbitrary key since the attacker does not know the client's id and ECDH secret. During the recovery phase, the attacker could retrieve the key but without locating and getting each Secret NFT back, the attacker will not be able to get the passphrase.

What if the Recovery Secret Contract is hacked? The attacker would be able to get all of the encryption keys but, again, without locating and getting each Secret NFT back, the attacker will not be able to get the passphrase. The fact that the encrypted passphrase is embedded in a Secret NFT separated from the Recovery Secret Contract is the most important security features that makes our system resilient to attacks.

However, having a friend holding the Secret NFT introduces another problem: what if that friend does not or cannot return the NFT back? The passphrase would be locked forever. We are improving this availability issue in our next and final iteration.

# 4 Iteration 3: Improving Availability

Having a unique friend holding Alice's NFT can be a problem if that friend does not or cannot return it to her. A naive solution would be to duplicate the same NFT and send it to multiple friends. This solution is not ok but not ideal in terms of security since we are extending the attack surface. The attacker can now target multiple people to regain one of these NFTs. In this third and last iteration, we adress the availability problem while preserving the security of our system by using the Shamir's Secret Sharing scheme.

## 4.1 The Shamir's Secret Sharing Scheme

Shamir's Secret Sharing scheme is a cryptography protocol usually used for splitting a secret into into multiple parts, called shares, which individually do not give any information away about that secret. To recover that secret, not all shares are needed but a minimun of shares called *"the threshold"*. On one hand, this is ideal from the usability perspective since the user does not have to collect all the shares back but only the minimum threshold required. On the other hand, this is perfect from the security perpective since any attacker who discovers any number of shares less than the threshold will not be able to break the secret.

## 4.2 The Protocol

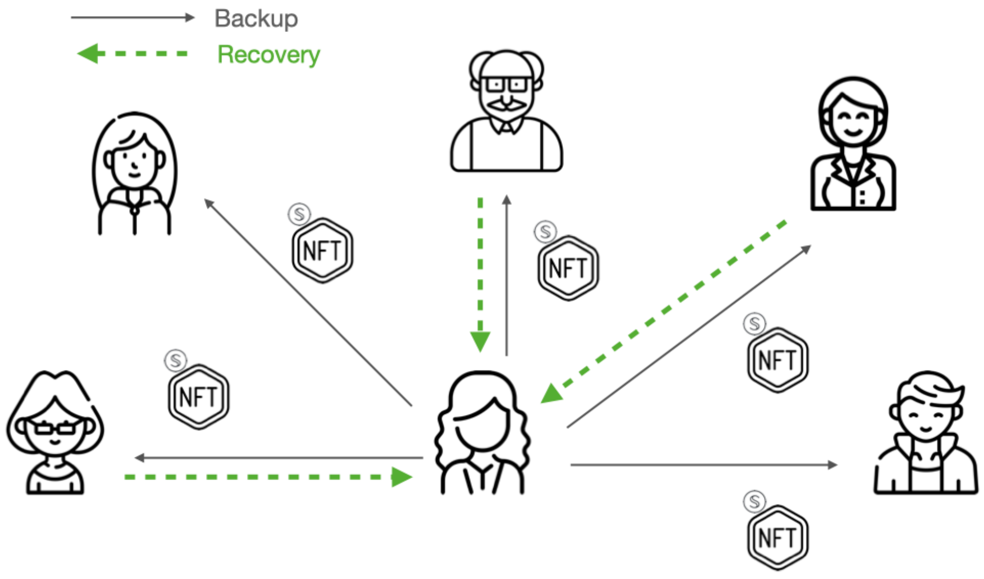

Figure 5: Iteration 3 - Shamir's Secret Sharing NFTs

**For the backup**, Alice needs to specify the number of shares $i$ she wants to generates and the minimum number of shares $j$ needed to retrieve the passphrase (threshold value). Once, the encryption key has been fully registered in the contract, the script encrypts the passphrase, concatenates the id and the ciphertext to generate the $i$ shares using the SSS algorithm: $(s_1, \cdots, s_i) = E_{\mathsf{SSS}}(id||c, i, j)$. Finally, the script mints the $i$ Secret NFTs on the Secret Network. Once the protocol is completed, Alice can now transfer the NFTs to her accointances.

**For the recovery**, Alice creates a new Secret Wallet and asks all of her friends to send their NFTs back. Once she has at least $j$ of those NFTs on her wallet, she goes to the recovery page that asks her to select the NFTs. Then, the script in the webpage collects the shares and extracts the $id$ and the ciphertext from the NFTs $id||c = D_{\mathsf{SSS}}(s_1, \cdots, s_j)$. Then, the protocol continues like in the previous iteration.

## 5 Value Capture for Secret Network Ecosystem

This backup system will demonstrate that the Secret Network privacy model is adequate to keep sensitive information confidential relying on two important features of Secret NFTs: secret metadata and private ownership.

Why implementing our backup system on Secret Network rather than Ethereum? Is it because Ethereum-based NFTs do not have secret metadata? Not really since we could have created a password protected version of the share and hid it inside a public NFT image using Steganography [6]. The real problem is that if an attacker knows that an NFT contains a share to recover a mnemonic phrase, he or she will be able to locate all others NFTs that contains the other shares easily since ownership is public on Ethereum. The fact that Secret NFTs protects the ownership is the key feature that makes our backup system secured.

## 6 Conclusion

In this paper, we propose a Decentralized Mnemonic Backup system that anyone can use to give custody to any blockchain passphrase to the Secret Network [5] and recover it using a simple email. The idea is to encrypt the passphrase and split it into multiple Secret NFTs using the Shamir's Secret Sharing cryptographic protocol. The encryption key is saved in a Secret smart contract. This key is never recorded in any transaction. Instead, it is generated using the Diffie-Hellman Key Exchange Protocol that is similarly used in well known protocols such as TLS and Signal.

The key recovering system can be used outside of our Mnemonic Backup system. It can be used for more advanced cryptographic protocols that involve storing and managing secret keys on chain with the option of recovering it using an email. For instance, this can be used to encrypt files on IPFS and manage the access using a Secret Contract that would hold custody of the encryption key. Such an approach has been proposed by *DataVault* in [7]. However, the *DataVault* white paper does not provide any details regarding how the encryption key handle on the Secret Network. Our system could be used to safely generate and manage the encryption key on the Secret Network and possibly have an email backup solution if such feature is desired.

## References

[1] Lost Passwords Lock Millionaires Out of Their Bitcoin Fortunes, https://www.nytimes.com/2021/01/12/technology/bitcoin-passwords-wallets-fortunes.html

[2] Mnemonic Generation (BIP39) Simply Explained, https://medium.com/coinmonks/mnemonic-generation-bip39-simply-explained-e9ac18db9477

[3] Metamask FAQ page, https://metamask.io/faqs/

[4] 1password, https://support.1password.com/forgot-account-password/

[5] Secret Network: A Privacy-Preserving Secret Contract & Decentralized Application Platform, https://scrt.network/graypaper

[6] Steganography, https://en.wikipedia.org/wiki/Steganography

[7] DataVault A Modular Encrypted Data Paywall and Storage Access Protocol https://data-vault.medium.com/datavault-a-modular-encrypted-data-paywall-and-storage-access-protocol-773a60c