Decentralized Mnemonic

Backup System

PriFi Labs Inc.

If you do not own your private keys . . .

. . . you do not own your crypto assets!

Non custodial wallets

➡ The private keys are stored on your device

◉ Problem
What if your device gets lost or stolen?

✓ Solution
During setup, you were given a "secret recovery phrase"
a 12-word mnemonic phrase (BIP39 standard) that can be used as a backup or to import the wallet into another device

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

How do I keep my wallet safe?

Backup your Secret Recovery Phrase

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 can't 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.

source: *Metamask FAQ*

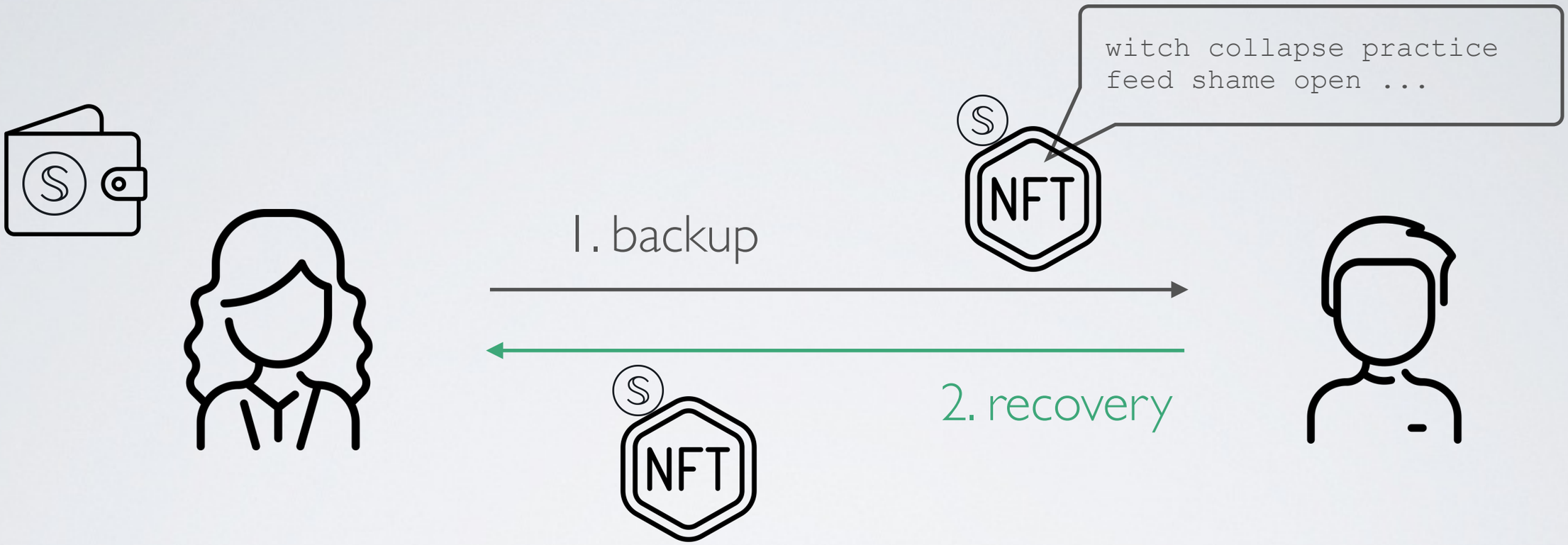
Storing a piece of paper in a safe place is inconvenient

- Problem - Where do you store your paper?
  In your cash and credit card wallet? It could get lost or stolen
  In the house? It could burn down
  In a deposit box at the bank? Good but cumbersome

✓ Solution
How about storing it on the blockchain directly?

➡ Decentralized Mnemonic Backup System
to save a backup of any blockchain mnemonic to the Secret Network

By leveraging Secret NFTs (Secret Network)

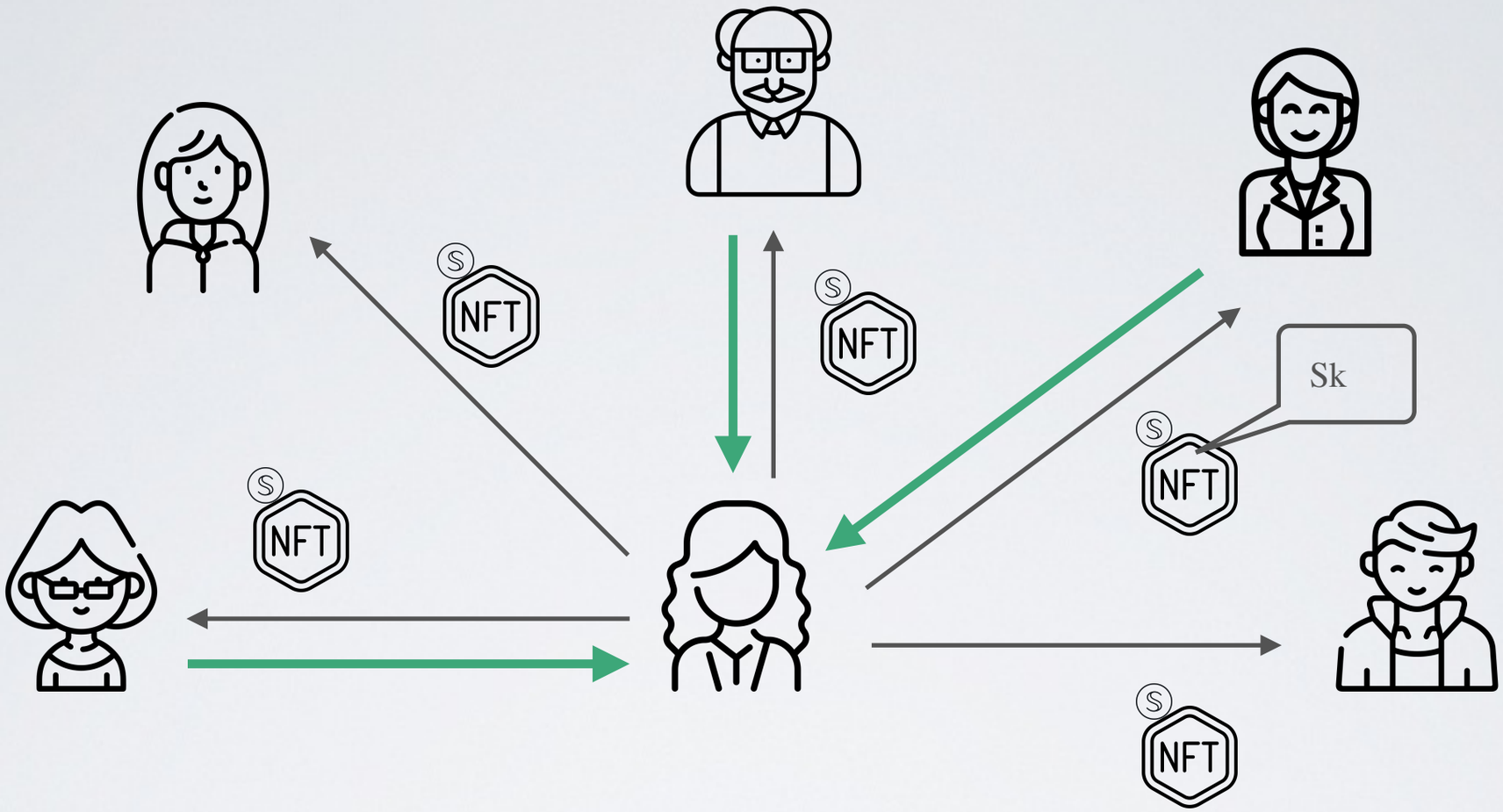


1. **Backup** - Alice creates a throwaway Secret wallet to mint an NFT (with the passphrase as private metadata) and send it to Bob
2. **Recovery** - Alice creates a new wallet and gets the NFT back to recover the passphrase

Not the best solution yet

- Bob can actually get Alice passphrase when owning the NFT
- What if Bob cannot return the NFT back to Alice when needed

By using Shamir's Secret Sharing scheme (SSS)



1. **Backup** - Alice splits the passphrase m into i secret shares with j threshold

$$(s_1, \ldots, s_n) = E_{SSS}(m, i, j)$$

2. **Recovery** - Alice needs only k shares back (threshold) to regenerate the passphrase

$$m = D_{SSS}(s_1, \ldots, s_j)$$

Better solution but still imperfect

- Problem
  Alice's friends can collude to gather j shares and uncover her passphrase

✓ Solution
  Encrypt the passphrase using an AES symmetric key k

$$\text{Backup} : (s_1, \ldots, s_n) = E_{SSS}(E_{AES}(k, m), i, j)$$

$$\text{Recovery} : m = D_{AES}(k, D_{SSS}(s_1, \ldots, s_k))$$

- .... but yet another problem
  Where does Alice store the encryption key when she needs it back?

✓ Solution : Key recovery system

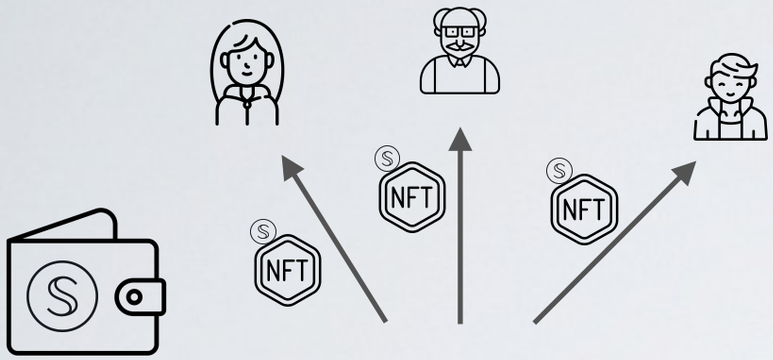
Key Recovery System

System requirements

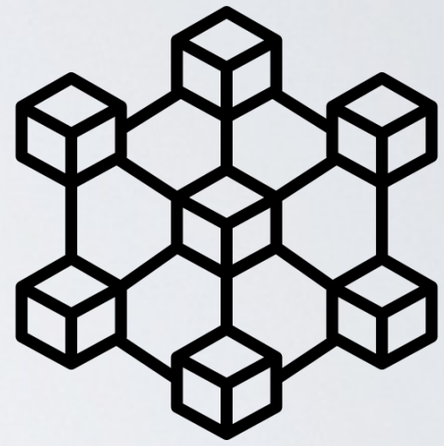
1. **Backup** - Alice pushes the secret key to the blockchain using her email and a throwaway wallet

2. **Recovery** - Alice recovers her secret key using her email and yet another throwaway secret wallet

Key registration



Recovery Secret Contract



1. register(email, id, key_info)

4. confirm(email, id, checksum)

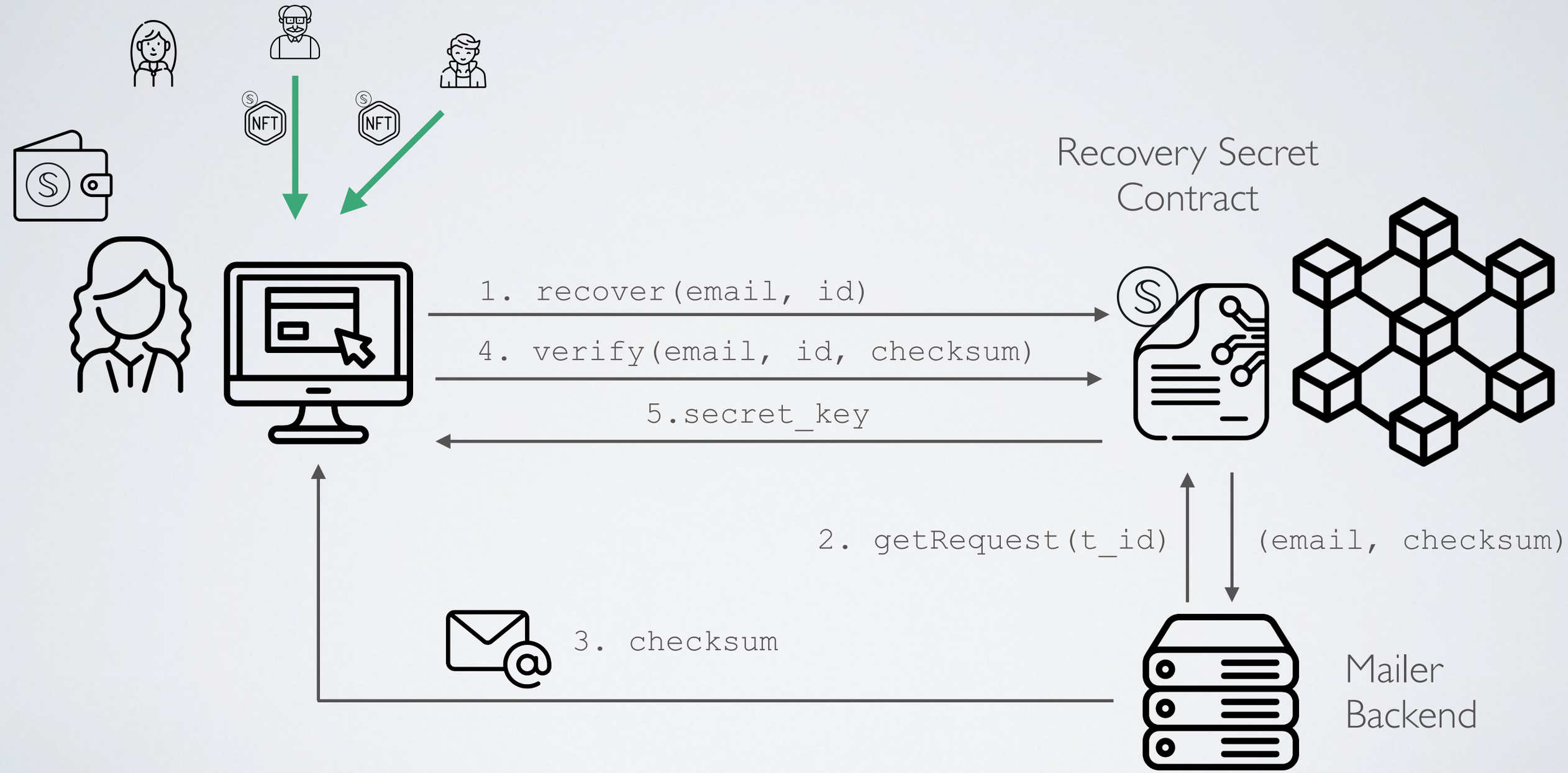
2. getRequest(t_id)

(email, checksum)

3. checksum

Mailer Backend

Key recovering



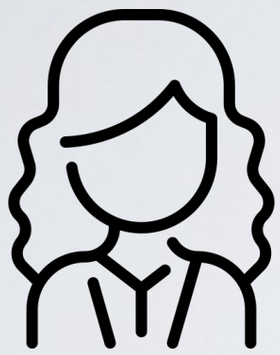
The cryptographic protocol

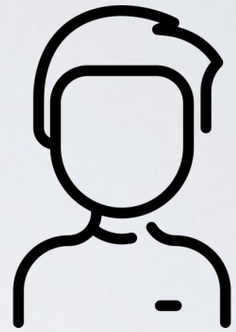
The problem of sending the key to the server

✓ Instead of sending the key directly, let's use a key agreement protocol between Alice and the Recovery Secret Contract

➡ ECDH (Elliptic-curve Diffie–Hellman)

ECDH (Elliptic-curve Diffie–Hellman)





1. generates (Sec_A, Pub_A)

n, Pub_A →

← Pub_B

2. calculates s = ECDH(Sec_A, Pub_A, Pub_B)
3. calculates k = PBKDF2(s, n)

1. generates (Sec_B, Pub_B)
2. calculates s = ECDH(Sec_B, Pub_B, Pub_A)
3. calculates k = PBKDF2(s, n)

Key registration

1. email, id, m, i, j (inputs)
2. generates (sec_A, pub_A) and n_A

3. register $_{tid, bid}$(email, id, pub_A, n_A) →

4. generates (sec_B, pub_B) and n_B
5. calculates s = ECDH(sec_B, pub_B, pub_A)
6. calculates k = PBKDF2(s, n_{1A}+ n_B)
7. store register(email, id, k, t_{id}, b_{id}, pub_B, n_B)

8. getTx $_{qry}$ (t_{id}) →

← 10. (email, pub_B, n_B)

9. assert mailer query and b_{id} + t < current$_{id}$

← 11. (pub_B, n_B)

12. calculates s = ECDH(sec_A, pub_A, pub_B)
13. calculates k = PBKDF2(s, n_A + n_B)
14. calculates h = HMAC(k, email + id)

15. confirm $_{tid', bid'}$(email, id, h) →

16. assert b_{id} + t < current$_{id}$
17. calculates h' = HMAC(k, email + id)
18. assert h == h'
19. store confirm(email, id, k)

20. calculates (s_1, ... , s_n) = E_{SSS}(id + E_{AES}(k, m), i, j)
21. for each s_k, generates secret NFT

Key recovery

1. email, (s_1, ... , s_j) (inputs)
2. calculates id + c = D_{SSS}(s_1, ... , s_j)
3. generates n_A

4. recover $_{tid, bid}$(email, id, n_A) →

5. assert confirm(email, id, k)
6. generates n_B
7. store recover(t_{id}, b_{id}, email, id, k, n_A, n_B)

8. getTx $_{qry}$ (t_{id}) →

9. assert mailer query and b_{id} + t < current$_{id}$

10. (email, n_B) ←

11. n_B ←

12. calculates h = HASH(n_A+ n_B)

13. verify $_{qry}$(email, id, h) →

14. assert b_{id} + t < current$_{id}$
15. calculates h' = HASH(n_A + n_B)
16. assert h == h'

17. k ←

18. calculates m = D_{AES}(k, c)

Threat model

What if the attacker hacks Alice's contract after registration?

✓ He/she will have access to Alice's ECDH public key but not the key

What if the attacker hacks the mailer backend?

✓ He/she will have access to contract's ECDH public key and the email but not the key

What if the attacker hacks the secret contract?

✓ He/she will have access to all cryptographic keys but since he/she does not know who owns the NFTS, he/she will not be able to recover the passphrase