A Decentralized Mnemonic Backup System for Non-Custodial Cryptocurrency Wallets

Thierry Sans[1,3], (David) Ziming Liu[2,3], and Kevin Oh[3]

[1] University of Toronto Scarborough, Toronto, Canada
[2] dApp Technology Inc. Toronto Canada
[3] PriFi Labs Inc. Toronto, Canada

thierry.sans@utoronto.ca david.liu@dapp-inc.com
kevin.oh@prifilabs.com

Abstract. When using non-custodial cryptocurrency wallets such as Exodus (Bitcoin), Metamask (Ethereum), and Keplr (Osmosis), the private keys are directly stored on the user's device. Using such a wallet comes with the risk of losing all crypto assets when the device gets lost, stolen, or breaks down irremediably. Fortunately, most non-custodial wallets offer a way to recover the private keys by the mean of a "recovery phrase" also known as a "mnemonic seed phrase". It is the 12-word phrase that is given when the wallet is created. Indeed, this mnemonic phrase is a really sensitive piece of information since anyone knowing that phrase can get full control of the crypto assets held by the wallet. Usually, it is recommended to write this passphrase down on a piece of paper and store it in a "safe place". However, storing a physical object is still not ideal since it can get stolen, lost, or destroyed as well. In this paper, we propose a decentralized application that can be used to back up mnemonic phrases and recover them eventually using a simple email. This application is built on a privacy-preserving blockchain to store the confidential passphrase and protect the identity of its owner.

Keywords: Blockchain · Smart Contracts · Decentralized Applications · Privacy · Wallets · Cryptography Protocol

1 Introduction

In the world of cryptocurrency, a *wallet* provides users with an interface to manage their crypto assets. Under the hood, those wallets store a set of private keys and use them to sign transactions. There are two types of wallets: "custodial wallets" in which the private keys are in the custody of a trusted third party and "non-custodial wallets" in which the private keys are directly stored on the user's device whether it is a computer, a mobile phone or a dedicated USB dongle. Using either of these types of wallets comes with inherent risks [1]. Using custodial wallets comes with the risk of losing all crypto assets when the trusted entity goes bankrupt[4] and non-custodial wallets come with the risk of

[4] as popularized by the mantra from *Andreas Antonopoulos*: *"Not Your Keys, not Your Coins"*

losing all crypto assets if the device gets lost, stolen or irremediably breaks down.

Yet, non-custodial wallets (our focus here) such as Exodus (Bitcoin), Metamask (Ethereum), and Keplr (Secret Network) offer a way to recover the private keys through the mean of a “recovery phrase” also known as “mnemonic seed phrase”. It is the 12-word phrase given the first time a wallet is set up. 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 [14]). It can be used as a backup or to import the 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 the crypto assets as explained in the *Metamask* Wallet FAQ page:

> *“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 your 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 above, users are not supposed to remember that phrase like a password but instead write it down on a piece of paper and store it in a “safe place”. However, storing physical objects is still not ideal since they can get stolen, lost, or destroyed as well. As a consequence, users do not always follow such a recommendation and put themselves at risk as studied in [17]. So, as an alternative, could we design a simple application that would take custody of that passphrase and would allow users to recover it based on some sort of authentication? Intuitively, that application could be something similar to a password manager but such a solution requires 1) that the service provider is trustworthy and 2) that the whole application is secured [9]. Moreover, a password manager is a centralized solution that goes against the idea of decentralized applications [4,5].

In this paper, we propose a decentralized application that can be used to back up private keys and mnemonic phrases and recover them eventually using a simple email. This application is built on a privacy-preserving blockchain [19] to store the confidential passphrase and protect the identity of its owner. To better explain our idea, we will go through 3 iterations each more secure than the previous one. In the first iteration (section 2), we aim at capturing the user experience but its overall design is rather naive and not secure. In the second iteration (section 3), we harden the security by encrypting the passphrase and separating the key storage from the passphrase storage. That second iteration makes use of Non-Fungible Tokens (NFT) [18] and the well know *Diffie–Hellman key exchange protocol* [6] to securely generate the encryption key directly on the blockchain. Finally, in the third iteration (section 4), we improve the reliability by splitting and distributing the encrypted passphrase across the network using another cryptographic protocol called *Shamir's Secret Sharing scheme* [16].

2 Iteration 1: The User Experience

Alice is a blockchain user that holds all kinds of crypto assets using different wallets. She would like to have an online backup of all of her wallet's passphrases in case her device gets lost, stolen, or breaks down irremediably. When that doomsday comes, she would like to recover her passphrases easily. The overall user experience is rather simple:

– **When Alice wants to back up 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 must copy onto the webpage 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 must copy onto the webpage 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 enabled), or reseting a password.

2.1 Architecture

Our Mnemonic Backup System is a decentralized application developed and deployed on a privacy-preserving blockchain called *Secret Network* [19]. Secret Network Smart Contracts enable storing and processing of private data directly on the blockchain. It relies on the Intel SGX (Software Guard Extension) Trusted Execution Environment (a.k.a Confidential Computing) [10] to prevent validator nodes from reading private data directly from memory during execution.

For this first iteration, we implement our Mnemonic Backup System as a smart contract on the Secret Network 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 Backend that will send emails to users. So, there are three entities in our system:

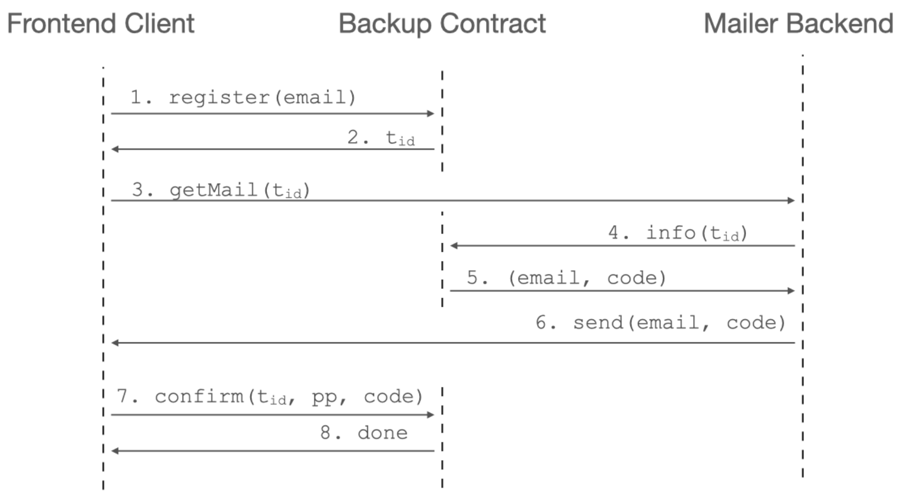

Fig. 1. Iteration 1 - Backup

- **The Backup Contract** is the smart contract that stores users' email and passphrases (the first iteration only, it will change in our second iteration)
- **The Frontend Client** is the webpage that allows users to back up and recover passphrases.
- **The Mailer Backend** is the off-chain mailer that sends confirmation/verification codes the users. For security reasons, the mailer backend never handles any passphrase. Only the Backup Contract has access to passphrases.

2.2 The Protocol

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

1. 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 wallet that Alice has just provisioned.
2. The Backup Contract generates a transaction id t_{id}, a random confirmation code, stores the record $(t_{id}, email, code)$ in the `register` dataset, and returns the transaction id to Alice.
3. Alice forwards the transaction id to the Mailer Backend.
4. The Mailer Backend queries the Backup Contract using that transaction id.
5. The Backup Contract returns the email and confirmation code associated to the transaction id.
6. The Mailer Backend sends the confirmation code to Alice by email.
7. Alice opens her email and copies and pastes the confirmation code into her browser and her passphrase *pp*. After pressing the submit button, a script,

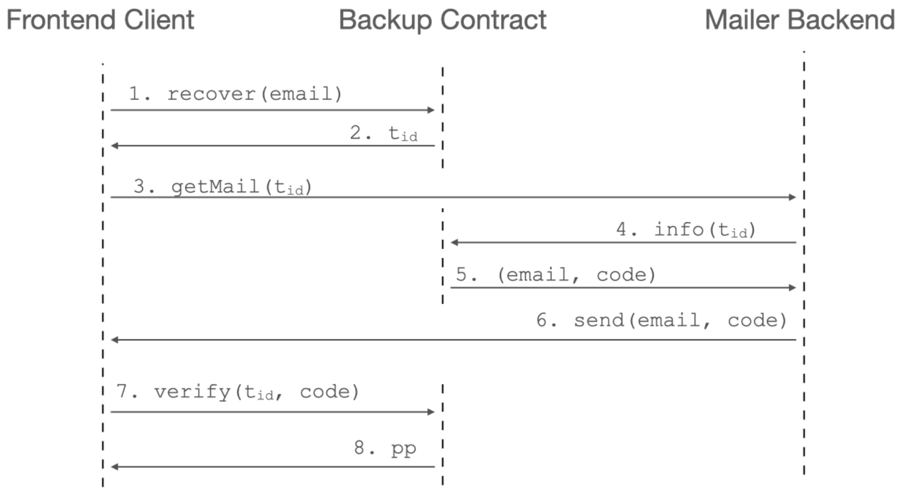

Fig. 2. Iteration 1 - Recovery

running inside the webpage, sends a transaction confirm$(email, pp, code)$ to the Backup Contract.

8. The Backup Contract verifies the code and stores the record $(email, pp)$ in the `confirm` dataset.

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

1. Alice enters her *email* and, after pressing the submit button, a script running inside the webpage sends a transaction recover$(email)$ to the Backup Contract to the Secret Network using a throwaway Secret wallet that Alice has just provisioned.
2. The Backup Contract generates a transaction id t_{id}, a random confirmation code, stores the record $(t_{id}, email, code)$ in the `recover` dataset, and returns the transaction id to Alice.
3. Alice forwards the transaction id to the Mailer Backend.
4. The Mailer Backend queries the Backup Contract using that transaction id.
5. The Backup Contract returns the email and confirmation code associated to the transaction id.
6. The Mailer Backend sends the confirmation code to Alice by email.
7. Alice opens her email and copies and pastes that verification code into her browser. After pressing the submit button, a script, running inside the webpage, sends a query verify$(email, code)$ to the Backup Contract.
8. The Backup Contract verifies the verification code, retrieves the corresponding record from the `recover` dataset and returns the passphrase.

2.3 Security Analysis

First and foremost, we are assuming that an attacker cannot retrieve data during transportation, storage, and execution without having the appropriate private keys thanks to the Secret Network protocol and the Intel SGX Trusted Execution Environment. That said, it is good to acknowledge that several attacks against Intel SGX have been published in the past few years [7,13,12,3] but those vulnerabilities have been mitigated by the Secret Network.

First, the attacker could breach into Alice's mailbox or the Mailer Backend directly. 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 another that the attacker can access. Then, any new asset that Alice puts in her wallet can be stolen by the attacker from now. During the recovery phase it is even worst since the attacker could trigger and use the verification code to query the Backup Contract directly and get the passphrase back.

Secondly, the attacker could compromise either Alice's wallet or the Admin's wallet that was used to deploy the Backup Contract. In the case of Alice, it is recommended that she uses a throwaway Secret wallet when registering a passphrase as explained earlier. As soon as the passphrase has been recorded, she can forget about this wallet since she will not need it to recover her passphrase. However, if that wallet is compromised afterward, the attacker could look into the transaction history using that key and get Alice's passphrase. The same problem holds if the attacker can compromise the Admin's wallet. It is worse than compromising a single user's wallet since the attacker could retrieve all users' passphrases from the transaction history directly.

3 Iteration 2: Security Hardening

Our first iteration captures the right user experience but fails in terms of security. Two main security threats need to be addressed: 1) prevent the attacker from taking advantage of having access to emails and 2) prevent the attacker from recovering passphrase from compromised wallets.

3.1 The Secret NFT

The first security problem is an authentication problem. Alice's identity can be compromised if the attacker breaches into Alice's mailbox, or worse, into the Mailer Backend directly. One way to prevent that is to add a second authentication factor. After careful consideration, we have chosen to rely on the blockchain directly. The idea is for Alice to store her passphrase in a Non-Fungible Token (NFT) [18] during backup and transfer that NFT to her friend Charlie. When doomsday comes, Alice could recover her passphrase by asking Charlie to send that NFT back. With this solution, we are taking advantage of Secret Network NFTs that protect the confidentiality of its content and the privacy of its owners. This is an important security aspect since the attacker would not know who owns Alice's NFT.

Yet, one problem remains. When Charlie becomes the owner, he could access its content and Alice's passphrase. A solution to that is to have Alice encrypt her passphrase when minting the NFT so that Charlie cannot decrypt it without the key. However, we have another problem now, where to store the key when Alice needs it to recover her passphrase? We could adopt the same approach as described in iteration 1. Instead of storing all passphrases in the Backup Contract, we could store all users' encryption keys.

3.2 The Diffie-Hellman Key Exchange

This Secret NFT solution helps us mitigate the second security issue as well. If the passphrase is never sent in clear to the network, the attacker cannot read it even if Alice's wallet or the Admin's wallet is compromised. The attacker could only retrieve the encryption key without being able to locate the encrypted content. For stronger security, we should prevent the attacker from getting that encryption key using a compromised wallet. This problem can be fixed implementing the *Diffie-Hellman Key Exchange* protocol to agree on the key without sharing it. The goal is to have Alice's wallet and the Backup Contract negotiate a key without exchanging it explicitly during the backup phase. The *Diffie–Hellman 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 [6]. That channel is considered as insecure because we assume that an attacker can eavesdrop on the communication and read all messages sent back and forth between Alice and Bob.

In practice, we use the *Elliptic-curve Diffie–Hellman* (abbreviated ECDH) that relies on Elliptic-curve cryptography [2]. In a nutshell, Alice generates an asymmetric key pair (sec_A, pub_A) and sends the public one to Bob over the insecure channel. When Bob receives Alice's public key, he will also generate its own pair (sec_B, pub_b) and send his public key back to Alice. Once the public keys have been exchanged, Alice and Bob can combine the public key with their private key to 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 resides in the fact that an attacker cannot compute that secret value s even if pub_A and pub_B are known but not 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 the HMAC-based extract-and-expand key derivation function HKDF [8] that generates the same cryptographic key based on the secret value.

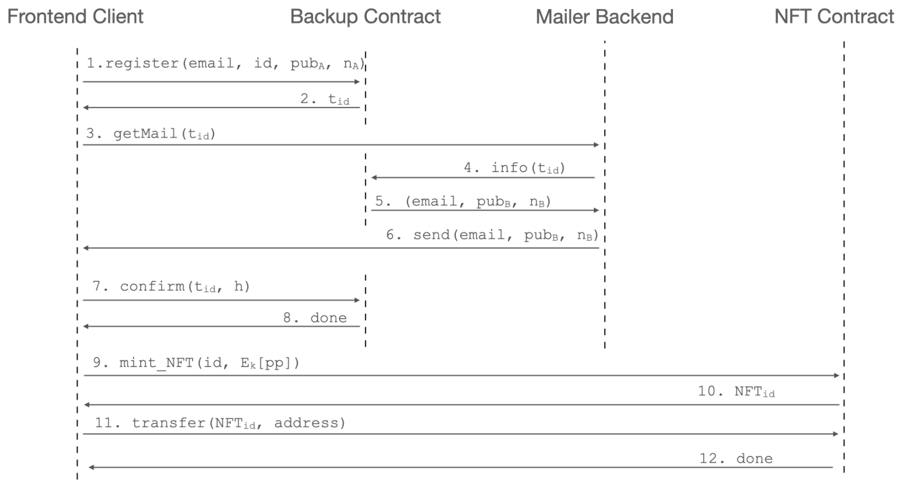

Fig. 3. Iteration 2 - Backup

3.3 The Protocol

Moreover, we are adding two additional features to our NFT-based Mnemonic Backup System. First, Alice can now back up 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 for the Backup 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 made her original request to the Backup Contract.

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

Backup

1. Alice enters her $email$, her passphrase m, and 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 wallet that Alice has just provisioned.
2. The Backup Contract generates a transaction id t_{id}, 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 HKDF and the concatenation of n_A and n_B as a salt $k = \mathsf{HKDF}(s, n_A || n_B)$. Finally, the contract stores the record $(email, id, t_{id}, b_{id}, pub_B, n_B)$ in the `register` dataset and returns the transaction id to Alice.

3. Alice forwards the transaction id to the Mailer Backend.
4. The Mailer Backend queries the Backup Contract $\mathsf{info}(t_id)$.
5. The Backup Contract checks that the query comes from the Mailer Backend wallet's address, retrieves the record from the `register` dataset, and checks that the query has not expired 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 to the Mailer Backend.
6. The Mailer Backend sends an email to Alice with the confirmation code (pub_B, n_B).
7. Alice opens her email and copies and pastes the confirmation code into her browser and enters her passphrase pp. 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 AES symmetric key k using HKDF and the concatenation of n_A and n_B as a salt $k = \mathsf{HKDF}(s, n_A||n_B)$. 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 $\mathsf{confirm}(t_{id}, h)$ to the Backup Contract.
8. The Backup Contract checks that the query has not expired 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 from the request. Finally, the contract stores the tuple $(email, id, k)$ in the `confirm` dataset.
9. The client-side script can now encrypt (using AES in GCM mode) the passphrase m with the key k to obtain the ciphertext $c = E_{\mathsf{AES}}(k, m)$ and send the id and the ciphertext to the NFT contract.
10. The NFT Contract mints a Secret NFT and returns the NFT_{id}.
11. Alice transfers the NFT to her friend Charlie.

Recovery

1. Alice creates a new Secret Wallet and asks Charlie to send her NFT back. Then, she goes to the recovery page that asks her to select the NFT id. A script running inside the webpage queries the NFT contract for that NFT id.
2. The NFT contract returns the id and the ciphertext from that NFT id.
3. The client's script generates a nonce n_A and sends the transaction $\mathsf{request}(email, id, pub_A, n_A)$ to the Secret Network.
4. The Backup 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 and returns the transaction id to Alice.
5. Alice forwards the transaction id to the Mailer Backend.
6. The Mailer Backend queries the Backup Contract $\mathsf{info}(t_id)$.

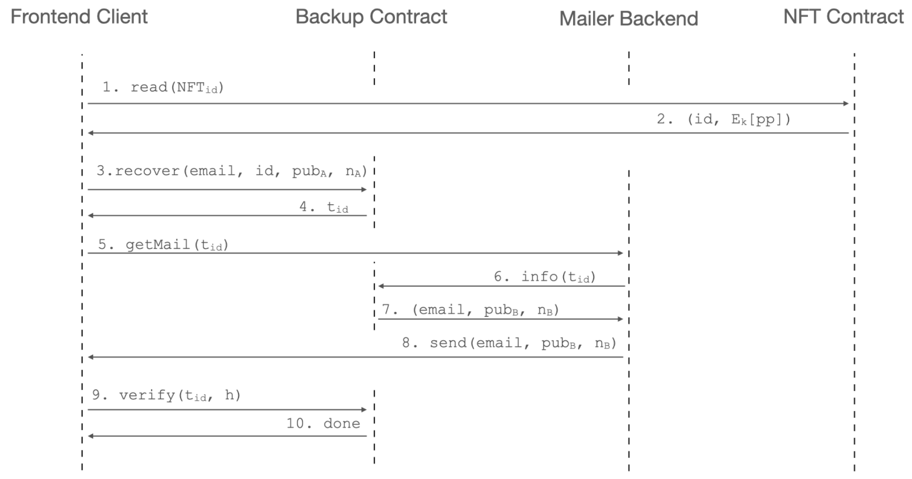

Fig. 4. Iteration 2 - Recovery

7. The Backup Contract checks that the query comes from the Mailer Backend wallet's address, retrieves the record from the `register` dataset, and checks that the query has not expired 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 to the Mailer Backend.
8. The Mailer Backend sends an email to Alice with the confirmation code (pub_B, n_B).
9. Alice opens her email and copies and pastes 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 Backup Contract.
10. The Backup Contract checks that the query has not expired 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 from the request. Finally, it returns the key k to the client.

Once the protocol is completed and Alice has received the key back, the client-side script can decrypt the ciphertext c to obtain the passphrase $m = D_{\mathsf{AES}}(k, c)$.

3.4 Security Analysis

Let's assume that the attacker can breach into Alice's mailbox or the Mailer Backend directly. 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 the ECDH secret. During the recovery phase, the attacker could retrieve the encryption key but will not be able to get the passphrase without locating and getting the Secret NFT back first.

Let's assume that an attacker has compromised Alice's wallet or the Admin's wallet. The attacker would be able to locate the NFT but will have to 1) convince that person to send the NFTs back and 2) hijack the email to retrieve the encryption key.

In the end, carrying an attack is technically possible but way harder than in the first iteration. The attacker should first compromise one of the wallets to locate the NFT, convince the person holding the NFT to send it back to the attacker's wallet (phishing attack), and finally compromised the victim's email to retrieve the verification code.

However, having a friend holding the Secret NFT introduces another problem: what if that friend does not or cannot return the NFT? 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 feasible 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 address the availability problem while preserving the security of our system by using *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 multiple parts, called shares, which individually do not give any information away about that secret [16]. To recover that secret, not all shares are needed but a minimum of shares that is called *"the threshold"*. 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. Moreover, this is perfect from the security perspective since any attacker who can retrieve any number of shares less than the threshold will not be able to break the secret.

4.2 The Protocol

For the backup, Alice needs to specify the number of shares i she wants to generate and the minimum number of shares j needed to retrieve the passphrase

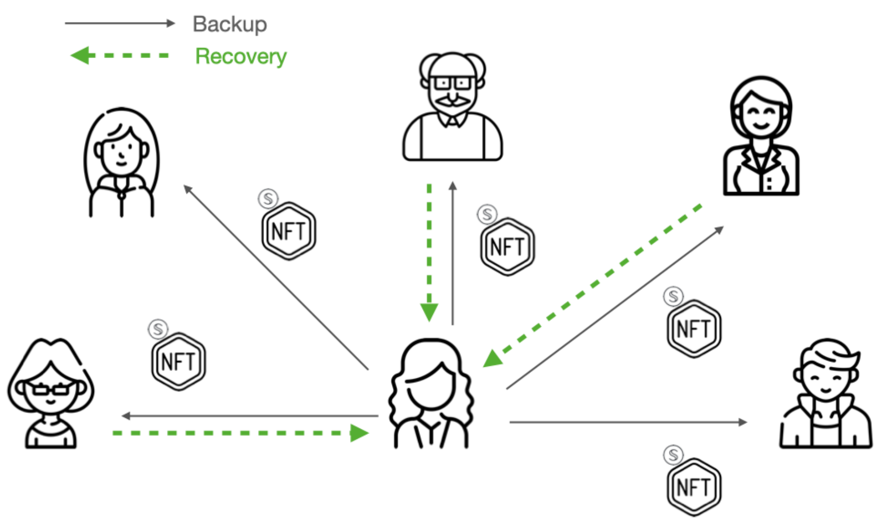

Fig. 5. Iteration 3 - Shamir's Secret Sharing NFTs

(threshold value). Once, the encryption key has been fully registered in the contract, the script encrypts the passphrase, and 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 acquaintances.

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 in 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 Related Work

Users must keep the wallet's mnemonic phrase safe because whoever gets access to that can access all of the crypto assets held in the wallet. To the best of our knowledge, there is only one significant proposal addressing the same issue. In [15], Rezaeighaleh and al. propose using a second wallet for backup. They propose a protocol based on Elliptic-Curve Diffie-Hellman to back up the private keys of the first wallet into a second wallet. They recommend having that secondary wallet be a "cold" wallet such as a hardware USB dongle or a smart card. This approach is technically sound but again relies on storing a physical object in a safe place which is hard in practice as shown in [17].

6 Conclusion and Future Work

In this paper, we propose a Decentralized Mnemonic Backup system that anyone can use to give custody of any blockchain passphrase to the Secret Network and recover it using a simple email. The idea is to encrypt the passphrase and split it into multiple Secret NFTs using *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 which is similarly used in well-known protocols such as TLS and the *Signal* messaging app.

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 them using an email. For instance, this can be used to encrypt files on the InterPlanetary File System (IPFS) [11] and manage the access using a Secret Contract that would hold custody of the encryption key. 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 a feature is desired.

Acknowledgments This work has been funded through a grant from the *Secret Network*. We want to thank the people from *Secret Labs* for giving us early feedback on our proposal and helping us develop the prototype.

References

1. Azar, P.D., Baughman, G., Carapella, F., Gerszten, J., Lubis, A., Perez-Sangimino, J., Rappoport, D.E., Scotti, C., Swem, N., Vardoulakis, A., et al.: The financial stability implications of digital assets (2022)
2. Bernstein, D.J.: Curve25519: new diffie-hellman speed records. In: International Workshop on Public Key Cryptography. pp. 207–228. Springer (2006)
3. Biondo, A., Conti, M., Davi, L., Frassetto, T., Sadeghi, A.R.: The guard's dilemma: Efficient {Code-Reuse} attacks against intel {SGX}. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 1213–1227 (2018)
4. Buterin, V., et al.: Ethereum: a next generation smart contract and decentralized application platform (2013). URL {http://ethereum. org/ethereum. html} (2017)
5. Cai, W., Wang, Z., Ernst, J.B., Hong, Z., Feng, C., Leung, V.C.: Decentralized applications: The blockchain-empowered software system. IEEE Access **6**, 53019–53033 (2018)
6. Diffie, W., Hellman, M.E.: New directions in cryptography. In: Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman, pp. 365–390 (2022)
7. Götzfried, J., Eckert, M., Schinzel, S., Müller, T.: Cache attacks on intel sgx. In: Proceedings of the 10th European Workshop on Systems Security. pp. 1–6 (2017)
8. Krawczyk, H., Eronen, P.: Hmac-based extract-and-expand key derivation function (hkdf). Tech. rep. (2010)

9. Li, Z., He, W., Akhawe, D., Song, D.: The {Emperor's} new password manager: Security analysis of web-based password managers. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 465–479 (2014)
10. McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C.V., Shafi, H., Shanbhogue, V., Savagaonkar, U.R.: Innovative instructions and software model for isolated execution. Hasp@ isca **10**(1) (2013)
11. Muralidharan, S., Ko, H.: An interplanetary file system (ipfs) based iot framework. In: 2019 IEEE international conference on consumer electronics (ICCE). pp. 1–2. IEEE (2019)
12. Murdock, K., Oswald, D., Garcia, F.D., Van Bulck, J., Gruss, D., Piessens, F.: Plundervolt: Software-based fault injection attacks against intel sgx. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 1466–1482. IEEE (2020)
13. Nilsson, A., Bideh, P.N., Brorsson, J.: A survey of published attacks on intel sgx. arXiv preprint arXiv:2006.13598 (2020)
14. Palatinus, M., Rusnak, P., Voisine, A., Bowe, S.: Mnemonic code for generating deterministic keys. Online at https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki (2013)
15. Rezaeighaleh, H., Zou, C.C.: New secure approach to backup cryptocurrency wallets. In: 2019 IEEE Global Communications Conference (GLOBECOM). pp. 1–6. IEEE (2019)
16. Shamir, A.: How to share a secret. Communications of the ACM **22**(11), 612–613 (1979)
17. Voskobojnikov, A., Wiese, O., Mehrabi Koushki, M., Roth, V., Beznosov, K.: The u in crypto stands for usable: An empirical study of user experience with mobile cryptocurrency wallets. In: Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems. pp. 1–14 (2021)
18. Wang, Q., Li, R., Wang, Q., Chen, S.: Non-fungible token (nft): Overview, evaluation, opportunities and challenges. arXiv preprint arXiv:2105.07447 (2021)
19. Zyskind, G., Nathan, O., et al.: Decentralizing privacy: Using blockchain to protect personal data. In: 2015 IEEE Security and Privacy Workshops. pp. 180–184. IEEE (2015)