Abelian Improvement Proposal 0011: mnemonic code for generating deterministic account

Abelian Foundation

October 23, 2024

Abstract. This AIP describes the implementation of a mnemonic code or mnemonic sentences, say a group of easy to remember words, for the deterministic generation and recovery of wallets/accounts. It consists of two parts: (1) a mapping between random bit-strings (referred to as Entropy-Seed) and mnemonics, and (2) a deterministic derivation of account-root-seeds from a given Entropy-Seed.

The first part, say the mapping between Entropy-Seeds and Mnemonics, is the same as that in BIP0039. This is to allow a mnemonic code to be used in a multiple-currency wallet which simultaneously supports multiple cryptocurrencies that follow the BIP0039.

The second part defines a rule on deriving account-root-seeds from a given Entropy-Seed, which is intently defined to be exclusively used by Abelian, particularly, Abelian does not re-use any existing rules, for example, BIP0039. This is to isolate the secret keys of different cryptocurrencies in one wallet, even if they use the same mnemonic, guaranteeing the security of each cryptocurrency even when other cryptocurrencies' keys are compromised.

Contents

1 Mapping Between Entropy-Seeds and Mnemonics

1.1 Sampling An Entropy-Seed

This proposal uses an entropy of 256 bits as the random bit-string seed for a deterministic wallet account, referred to as **Entropy-Seed**.
Note that it is required that **Entropy-Seed** is sampled randomly and uniformly from $\{0,1\}^{256}$, as shown in Algorithm 1.

Algorithm 1 $SampleEntropySeed()$

```
1: $entropyseed \xleftarrow{\$} \{0,1\}^{256}$
2: return $entropyseed$
```

1.2 From Entropy-Seed To Mnemonic

For a given **Entropy-Seed** *entropyseed*, the corresponding mnemonic is obtained by the following Algorithm 2.

Algorithm 2 $EntropySeedToMnemonic(entropyseed, wordlist)$

```
1: $cs \leftarrow the\ first\ 8\ bits\ of\ \mathsf{SHA256}(entropyseed)$
2: $ext \leftarrow entropyseed \| cs$
3: $ms_{23} \| ms_{22} \| \ldots \| ms_0 \leftarrow ext$
4: for $t = 0$ to 23 step 1 do
5:     $i_t \leftarrow \mathsf{BinaryToInt11}(ms_t)$
6:     $mnemonic[t] \leftarrow wordlist[i_t]$
7: end for
8: return $mnemonic$
```

Remark:

- *entropyseed* is an **Entropy-Seed** sampled as in Section 1.1.
- *ext* consists of 256+8 = 264 bits.

- *ext* is split into 24 groups of bits, say $ms_0, ms_2, \ldots, ms_{23}$, each consisting of 11 bits, such that $ms_{23} \| ms_{22} \| \ldots \| ms_0 = ext$.
- $\mathsf{BinaryToInt11}()$ is the **standard** algorithm that converts binary-string in $\{0,1\}^{11}$ to the corresponding decimal integer in $\{0, 2047\}$. In particular, $\mathsf{BinaryToInt11}(000, 0000, 0001) = 1$, $\mathsf{BinaryToInt11}(000, 0000, 1000) = 8$, $\mathsf{BinaryToInt11}(100, 0000, 0000) = 1024$, and so on.
- *wordlist* is the commonly used wordlist with 2048 words as in BIP0039 [4], as shown in Appendix A.
- As a result, the output *mnemonic* consists of 24 words in *wordlist*.

1.3 From Mnemonic To Entropy-Seed

The mapping from Mnemonic to **Entropy-Seed** is just the inverse procedure as shown in the following Algorithm 3.

Algorithm 3 $MnemonicToEntropySeed(mnemonic, wordlist)$

```
1: for $t = 0$ to 23 step 1 do
2:     $i_t \leftarrow \mathsf{LookupIndex}(mnemonic[t], wordlist)$
3:     if $i_t \notin [0, 2047]$ then
4:         return FAIL
5:     end if
6:     $ms_t \leftarrow \mathsf{IntToBinary11}(i_t)$
7: end for
8: $ext \leftarrow ms_{23} \| ms_{22} \| \ldots \| ms_0$
9: $entropyseed \leftarrow the\ first\ 256\ bits\ of\ ext$
10: $cs \leftarrow the\ last\ 8\ bits\ of\ ext$
11: $cs' \leftarrow the\ first\ 8\ bits\ of\ \mathsf{SHA256}(entropyseed)$
12: if $cs' \neq cs$ then
13:     return FAIL
14: end if
15: return $entropyseed$
```

Remark:

- *mnemonic* consists of 24 words.
- *wordlist* is the commonly used wordlist with 2048 words as in BIP0039, as shown in Appendix A.
- $\mathsf{LookupIndex}(word, wordlist)$ finds the index of *word* in *wordlist*. Note that if the output index is not in the scope $[0, 2047]$, it implies that an illegal word is used and FAIL is returned.
- $\mathsf{IntToBinary11}()$ is the inverse of $\mathsf{BinaryToInt11}()$, converting an integer in $[0, 2047]$ to a binary-string in $\{0,1\}^{11}$.
- The output *entropyseed* is a 256-bits string in $\{0,1\}^{256}$.

1.4 Known Answer Tests

todo

2 Derivation of Account-Root-Seeds from Entropy-Seed

2.1 Preliminaries on Abelian Wallet Account

Account and Account-Root-Seeds. In Abelian, as shown in Fig. 1, each **account** consists of a set of **root seeds/keys**, referred to as **Account-Root-Seeds**, say ($\mathsf{CoinSpKeyRootSeed}$, $\mathsf{CoinSnKeyRootSeed}$, $\mathsf{CoinDetectorRootKey}$, $\mathsf{CoinVKRootSeed}$), where $\mathsf{CoinVKRootSeed}$ and $\mathsf{CoinSnKeyRootSeed}$ are optional. In particular, for an account which will generate only semi-privacy (i.e., pseudonym) addresses, the $\mathsf{CoinSnKeyRootSeed}$ and $\mathsf{CoinVKRootSeed}$ are null. Note that $\mathsf{CoinSpKeyRootSeed}$, $\mathsf{CoinSnKeyRootSeed}$, $\mathsf{CoinDetectorRootKey}$, $\mathsf{CoinVKRootSeed}$ are all 512-bits.

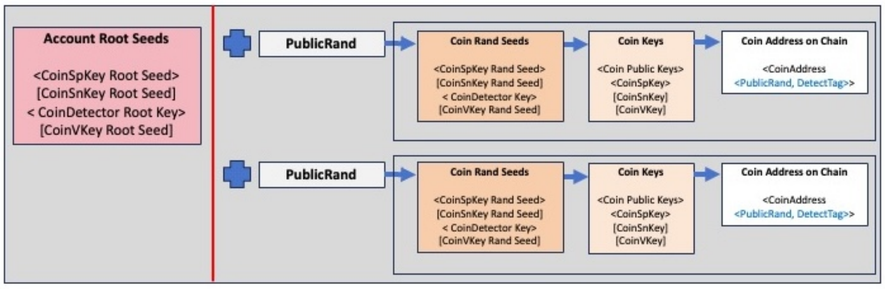

Figure 1: Account and (Address, key)

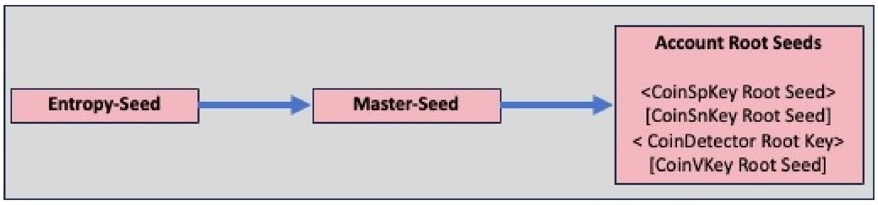

Figure 2: Derivation of Account Root Seeds

Public Rand and (Address, Key). As shown in Fig. 1, to generate an (Address, Key) pair under an account, a **Public Rand** (with 512 bits) needs to be introduced. In particular, for each given Public Rand, a corresponding (Address, Key) will be *deterministically* generated from the Account-Root-Seeds. Note that there are two ways to generate a (address, key) pair under an account, namely (1) given only the Account-Root-Seeds: sample a random Public Rand and generate the (address, key) pair from the given Account-Root-Seeds and sampled Public Rand, or (2) given the Account-Root-Seeds and a well-form Public Rand: generate the (address, key) pair from the given Account-Root-Seeds and Public Rand.

Derivation of Account Root Seeds. For such an account, it is ideal that CoinSpKeyRootSeed, CoinSnKeyRootSeed, CoinDetectorRootKey, CoinVKRootSeed are independent entropies. However, from the view of practice, it is desired that they are derived from an entropy, referred to as Master-Seed, as shown in Fig. 2.

Furthermore, to provide good user-friendliness (say, using mnemonic), this proposal derives a Master-Seed from Entropy-Seed, and then derives Account-Root-Seeds from Master-Seed.

Note that this proposal does not directly use Entropy-Seed as Master-Seed. This is to provide flexibility to potential extension, for example, deriving multiple accounts from one Entropy-Seed.

Notations. Below we define a PRF

$$\mathsf{PRF}(key, input) := \mathsf{KMAC256}(key, input, 512, \text{“}\mathsf{ABELIANPRF}\text{”})$$

where KMAC256 servers as a PRF [2] (approved by NIST) to generate 512-bits output, *input* serves as the context, “ABELIANPRF” specifies the Domain Separation Customization String. Note that here with 512-bits (resp. 256-bits) key and 512-bits output, the above PRF provides 256-bits (resp. XX-bits) security. todo: Why use KMAC256 with 256-bits input and 512-bits output rather than HMAC-SHA512? Have an explanation? We want to use SHA3 rather SHA2 (i.e., SHA512). But we only know that KMAC is a variable-length message authentication code algorithm based on KECCAK which is the underlying function of the SHA3 standard.

2.2 Derivation From Entropy-Seed to Master-Seed

Given an Entropy-Seed, a corresponding Master-Seed is deterministically derived as shown in the following Algorithm 4. todo: confirm: Note that this derivation has the following security features: given a master-seed, it is infeasible to distinguish the Entropy-Seed from a random $x \in \{0,1\}^{256}$. Could this be deduced from the security of PRF?

Remark:

Algorithm 4 $EntropySeedToMasterSeed(entropyseed, customizationContext)$

```
1: $masterseed \leftarrow \mathsf{PRF}(entropyseed, \text{“}\mathsf{AccountMasterSeed}\text{”} \| \mathsf{customizationContext})$
2: return $masterseed$
```

- *entropyseed* is 256-bits, determined by the 24-words mnemonic rule.
- *entropyseed* serves as the key.
- “AccountMasterSeed”‖customizationContext servers as the input, where different applications may use different customiztionContext (which is “” by default). This also allows to support the case of generating multiple accounts by one mnemonic.
- *masterseed* is 512-bits output.

Note that this derivation is very different from that of BIP0039.

2.3 Derivation From Master-Seed to Account-Root-Seeds

Given a Master-Seed, the corresponding Account-Root-Seeds is deterministically derived as shown in the following Algorithm 5. todo: confirm: Note that this derivation has the following security features: given an Account-Root-Seeds, it is infeasible to distinguish the Master-Seed from a random $x \in \{0,1\}^{512}$. Could this be deduced from the security of PRF?

Algorithm 5 $MasterSeedToAccountRootSeeds(masterseed)$

```
1: $\mathsf{coinSpKeyRootSeed} \leftarrow \mathsf{PRF}(masterseed, \text{“}\mathsf{CoinSpendKeyRootSeed}\text{”})$
2: $\mathsf{coinSnKeyRootSeed} \leftarrow \mathsf{PRF}(masterseed, \text{“}\mathsf{CoinSerialNumberKeyRootSeed}\text{”})$
3: $\mathsf{coinDetectorRootKey} \leftarrow \mathsf{PRF}(masterseed, \text{“}\mathsf{CoinDetectorRootKey}\text{”})$
4: $\mathsf{coinVKRootSeed} \leftarrow \mathsf{PRF}(masterseed, \text{“}\mathsf{CoinValueKeyRootSeed}\text{”})$
5: return $(\mathsf{coinSpKeyRootSeed}, \mathsf{coinSnKeyRootSeed}, \mathsf{coinDetectorRootKey}, \mathsf{coinVKRootSeed})$
```

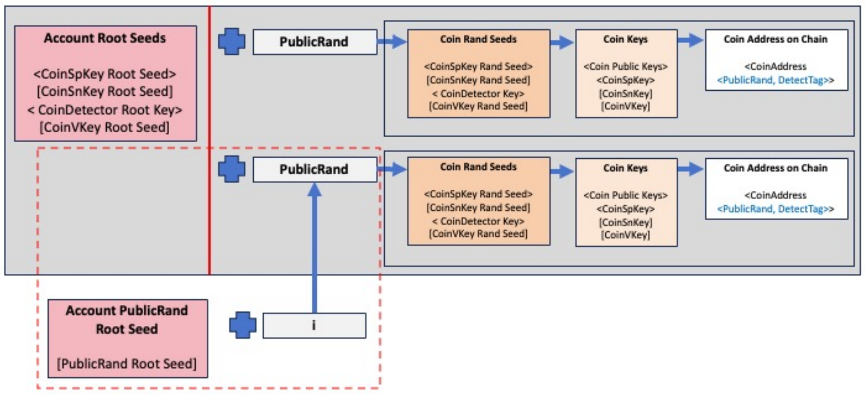

Figure 3: Account with Deterministic Public Rands

2.4 Deterministic Public Rands

In some scenario, it is desired that deterministic Public Rands are used. To support such a scenario, this proposal defines a rule to deterministically generate Public Rands from integers in $[0, 2^{32} - 1]$, referred to as ”Sequence Numbers”. In particular, this proposal derives a PublicRandRootSeed for an account from MasterSeed, and then derives Public Rands from given sequence numbers when needed, as shown in Fig. 3, Algorithm 6, and Algorithm 7.

Algorithm 6 $MasterSeedToAccountPublicRandRootSeed(masterseed)$

```
1: publicRandRootSeed ← PRF($masterseed$, “PublicRandRootSeed”)
2: return publicRandRootSeed
```

Algorithm 7 $DerivePublicRand(publicRandRootSeed, i)$

```
1: $seqNo$ ← EncodeSeqNo($i$)
2: publicRand ← PRF($publicRandRootSeed, seqNo$)
3: return publicRand
```

Remark:

- publicRandRootSeed output by $MasterSeedToAccountPublicRandRootSeed()$ is 512-bits.

- i is an integer in $[0, 2^{32} - 1]$.

- EncodeSeqNo(i) encodes $i \in [0, 2^{32} - 1]$ to a hex-string of length 8. In particular, EncodeSeqNo(1) = 00000001, EncodeSeqNo(10) = 0000000a, EncodeSeqNo(31) = 0000001e.

- The output publicRand is 512-bits.

2.5 Known Answer Tests

todo

3 Compatibility

3.1 A Previous Version

At $height = 300,000$, we released packages of AbewalletMLP-v1.0.1, which follow BIP0039, namely

- The mapping between Entropy-Seed and Mnemonic is the same as Algorithm 2 and Algorithm 3.

- The derivation from Entropy-Seed to Master-Seed is shown as the following Algorithm 8.

- The derivation from Master-Seed to Account-Root-Seeds is shown as the following Algorithm 9.

Remark:

Algorithm 8 $EntropySeedToMasterSeed_{BIP39}(entropyseed)$

```
1: $words \leftarrow EntropySeedToMnemonic(entropyseed)$
2: $key \leftarrow Use\ whitespace\ to\ splice\ words$
3: $masterseed \leftarrow \mathsf{PBKDF2}(key, \text{“}\mathsf{mnemonic}\text{”}, 2048, 64, \mathsf{SHA512})$
4: return $masterseed$
```

Algorithm 9 $MasterSeedToAccountRootSeeds_{old}(masterseed)$

```
1: coinSpKeyRootSeed ← PRFOLD($masterseed$, “spendkey”)
2: coinSnKeyRootSeed ← PRFOLD($masterseed$, “serialnumberkey”)
3: coinDetectorRootKey ← PRFOLD($masterseed$, “valuekey”)
4: coinVKRootSeed ← PRFOLD($masterseed$, “detectorkey”)
5: return (coinSpKeyRootSeed, coinSnKeyRootSeed, coinDetectorRootKey, coinVKRootSeed)
```

- PBKDF2 [1] applies a pseudorandom function (such as HMAC-SHA512) repeatedly to the salt and password to generate the key. The NIST Recommendation [5] approved PBKDF2 as the PBKDF using HMAC with any approved hash function as the PRF, and decided to revise it [3].

- PRFOLD is defined by :

  $$\mathsf{PRFOLD}(key, input) := \mathsf{KMAC256}(key, input, 512, \text{“}\mathsf{PQABELIAN\text{-}WALLET}\text{”})$$

  where KMAC256 servers as a PRF [2] to generate 512-bits output, *input* serves as the context, “PQABELIAN-WALLET” specifies the Domain Separation Customization String.

3.2 Compatibility Solution 1

- Make a standalone product line for AbewalletMLP-v1.0.1, which continue using the Algorithm 8 and Algorithm 9. And if necessary, Algorithm 6 and Algorithm 7 will be also used based on the output of Algorithm 8.

- Open a new product line which will use Algorithm 4, Algorithm 5, and if necessary Algorithm 6 and Algorithm 7.

- Pros. It is simple to let the user know which product line he is using.
- Cons. If the user wants to use the new-line-wallet, he has to use the mnemonic in a new-line-wallet, and explicitly transfer all his coins from old-line-wallet to new-line-wallet.

3.3 Compatibility Solution 2

- In updated wallet, both versions of Account-Root-Seeds are maintained, and when creating new addresses, the new version Account-Root-Seeds are used, so that in such an updated wallet, the coins on addresses generated from old version of Account-Root-Seeds will not increase any more, and may become less and less, and finally zero.
- Pros. The "upgrade" is transparent to users, and finally it is possible that all address hosting spendable coins are from new version of Account-Root-Seeds.
- Cons. This "double-track" model makes the wallet software a little complicated and less efficient. In particular, for each TXO on blockchain, the wallet has to use both the new version Detector-Root-Key and the old version Detector-Root-Key to check whether the coin belongs to the wallet (since the user may run multiple wallet instances on different devices), and furthermore, for each address/coin, there has to be a flag to imply which version Account-Root-Seeds it was generated. In addition, there is no mechanism to "terminate" the use of old version of Account-Root-Seeds, since it is possible that a user uses his old version wallet for a long time in the future. Is it possible to motivate users to upgrade all his wallet instances to new version and give an explicit signal to the wallet software?
  Even this is can be achieved? Is it possible to obtain this signal when wallet recovery?

3.4 Compatibility Solution 3

- When a new account is created, new version of Account-Root-Seeds is used and a flag is set to imply this point.
- When an account is recovered from a mnemonic, both old and new version of Account-Root-Seeds are re-covered. Then the wallet uses the old version of Account-Root-Seeds to scan the blocks on the blockchain in order
  - If any address on blockchain is detected as being derived from old version of Account-Root-Seeds, then a flag is set and the new version of Account-Root-Seeds is removed, and from that on, only old version of Account-Root-Seeds is used,
  - otherwise (there is not any address on blockchain is detected as derived from old version of Account-Root-Seeds), a flag is set and the old version of Account-Root-Seeds is removed, and from that on, only new version of Account-Root-Seeds is used.
- Pros. The "initial procedure of scan-and-decide" is transparent to users, and only one version of Account-Root-Seeds is used after the initial procedure. Note that the above "initial procedure of scan-and-decide" can be optimized as below: *the wallet uses the new and old versions of Account-Root-Seeds to scan the blocks on the blockchain in order*
  - If any address on blockchain is detected as derived from old version of Account-Root-Seeds, then a flag is set and the new version of Account-Root-Seeds is removed, and from that on, only old version of Account-Root-Seeds is used,
  - If any address on blockchain is detected as derived from new version of Account-Root-Seeds, a flag is set and the old version of Account-Root-

    Seeds is removed, and from that on, only new version of Account-Root-Seeds is used.

- Cons.
  - The above mechanism will fail in such an extreme situation: *A user runs an account with the old version of wallet, but never creates any address. Then he recovers the account on a different device with the new version of wallet.* For such a situation, the new wallet will use only new version of Account-Root-Seeds, while the old wallet uses only old versions of Account-Root-Seeds. The coins on the each device will not be synchronized to another.

    Could we prevent such a situation by asking users to update as soon as possible? or when recovery, asking the user did he use the account to receive or send coins?
  - This solution actually has the same effect as Solution 1, but in a transparent way. Namely, if a user has run an account using old version Account-Root-Seeds, then this account will continue use the old version of Account-Root-Seeds, i.e., sharing the master-seed with his other coin wallets, otherwise, new version of Account-Root-Seeds will be used.

    Which is better? explicitly or implicitly?

3.5 Compatibility Solution 4

Note that AbewalletMLP-v1.0.1 is released, but maybe only a few of users use it. How about use the solution 1? and just handle the cases of using AbewalletMLP-v1.0.1 case by case.

References

[1] B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898, Sept. 2000.

[2] J. Kelsey, S. jen Chang, and R. Perlner. Sha-3 derived functions: cshake, kmac, tuplehash and parallelhash, 2016-12-22 00:12:00 2016. `https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=922422`.

[3] NIST. Nist to revise sp 800-132, recommendation for password-based key derivation – part 1: Storage applications. Website, 2023. `https://csrc.nist.gov/News/2023/decision-to-revise-nist-sp-800-132`.

[4] M. Palatinus, P. Rusnak, A. Voisine, and S. Bowe. Mnemonic code for generating deterministic keys. `https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki`. Accessed 10 October 2024.

[5] M. S. Turan, E. B. Barker, W. E. Burr, and L. Chen. Recommendation for password-based key derivation ::part 1: storage applications, 2010-01-01 05:01:00 2010.

A WordList

The wordlist is the same as that of BIP0039 at `https://github.com/bitcoin/bips/blob/master/bip-0039/english.txt`. In particular,
todo