Design and development of an iOS app for management of wallets based on Keycard

Luigi Borchia

Contents

1 Basic concept

Cryptocurrencies such as Bitcoin and Ether are increasingly gaining popularity in these years, and this brings to the need of having tools for managing them which can satisfy high standards in terms of security and functionalities; the most important of these tools are crypto wallets, which must provide to users a strong solution to store their public and private keys, keeping their funds safe.
On the market many hardware wallets are available with different features, such as Ledger Nano X and Trezor, offering users a wide choice of solutions based on their specific needs; but a great limitation is given by the fact that all the available hardware wallets are only compatible with computers, working through USB interfaces. This can be a problem to users who want to spend an manage their cryptocurrencies on the go, maybe through their mobile devices, which nowadays play a central role in daily life routine.
Until now the only way to safely use crypto wallets in mobility was using hot wallets with just a small amount of the total funds charged, leaving the major part on a cold wallet to keep it secure from any attack (this solution couldn't even be defined as safe, but rather less risky in terms of losess).
The following sections will provide a detailed presentation of the work done.

2 Keycard for iPhone

The project main app is Keycard for iPhone. As the name may suggest, it's an iOS application made to interact with a Keycard through iPhone. The applications features both tools for card management and for its usage as crypto hardware wallet (transaction signing, key derivation ecc.).

2.1 Design

Cryptocurrencies are not the easiest subject to deal with, and most users decide to stay away because of the struggle in getting familiar with them. The aim of dApps and crypto-oriented apps should be to offer a simple user experience that can help users in getting more confident with this kind of technology. Based on this principle, the app was designed as the simplest it could be (even though its usage is won't be too much easy for anyone that hasn't ever heard of cryptocurrencies). It includes easy-to-use functions and ones intended for advanced users.

The app is based on three main sections, each one intended for different type of operations, following this structure:

2.1.1 Info

The "Info" screen is the first screen the user sees after authentication with the card. Here general info about the card are displayed: those include informations about the master key, applet version and free pairing slots. Such informations

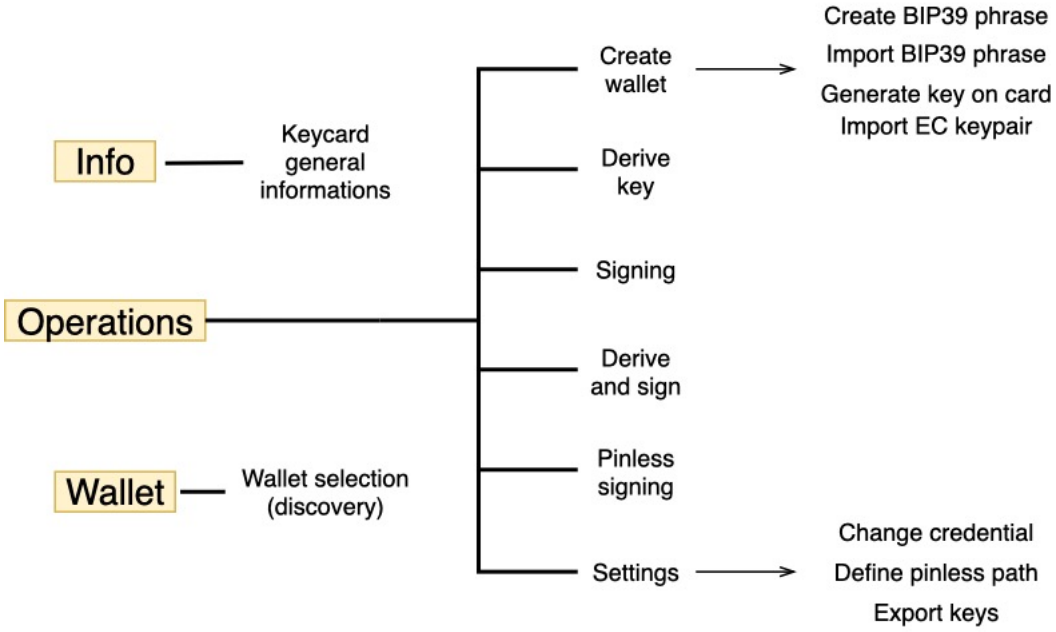


Figure 1: Keycard for iPhone UI structure.

can be useful for card management, expecially regarding the pairing slots, as the card will pair with a maximum number of five devices. This has been considered during the application development, as necessary measures were needed to be implemented in order to avoid lack of free pairing slots.

2.1.2 Operations

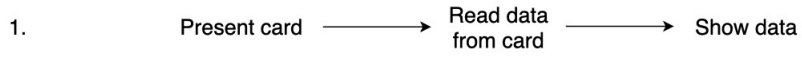
This section can be seen as a control panel for the card where the most important operations can be performed, including transaction signing and key derivation. Here the user has full control of the card: it's possible to create new wallets in different modalities, perform key derivation following the BIP32 standard for hierarchical deterministic wallets, sign transactions, set a pinless path (for operations to be done without pin verification), change the card credentials and export the card keys or Ethereum address. Users will mainly use this part of the application, that was built simple and clean; for many of the available operations short descriptions are provided in the respective controllers.

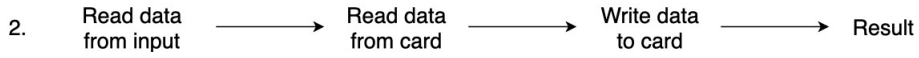
2.1.3 Wallet

This is the most "experimental" part of the application, as it provides a feature that was not built in the Keycard framework. Here the user can change the

current wallet with another one of another cryptocurrency: actually, once the currency is chosen, there's no creation of a new wallet, instead the application looks for any existing wallet for that currency associated with the card (a wallet is available when it has been previously used for at least one transaction); if one or more available wallets are found, the user can choose the one to use. This can be useful to users that use multiple currencies for transactions and want to quickly switch from one wallet to another.
How this is achieved will be explained in details in the implementation section.

All operations with the card follow two simple protocols:





Tipically all operations that will follow the protocol 2 will take more time to be performed then protocol 1 operations as writing data on the card takes secured procedures that use complex algorithms of the Keycard framework.

2.2 Implementation

Figure 2: Keycard for iPhone logo.

For the application development a Swift implementation of the Keycard framework was used (`https://github.com/status-im/Keycard.swift`).

2.2.1 NFC sessions

All operations involving the card require opening a NFC session that must be kept active until the process is complete, otherwise the operation will fail and an error message will be returned. Sometimes the failure of an operation could led to the loss of a free pairing slot, as all operations need the card to be paired with the device and on a failure occurrency the function could return with an error before it has unpaired; because of this I developed a separate application that

could only reset all the pairing slots, which I used when too many NFC session failed consecutively in order to avoid not being able to use the card anymore due to lack of free slots.
Almost all the NFC sessions are managed in the same way, as the majority of operations require first to:

1. **pair the card with the device**: clients wishing to communicate with the card, need to pair with it first. This allows creating secure channels resistant not only to passive but also to active MITM attacks. To establish the pairing, the client needs to know the pairing password. After it is established, the pairing info should be stored as securely as possible on the client for subsequent sessions;

2. **open a secure channel**: communication with the card happens over a Secure Channel to protect sensitive information being transmitted. The Secure Channel relies on a pairing mechanism for mutual authentication[?];

3. **authenticate the user**: most operations with the card (all involving operations with the wallet or credentials) require authenticating the user. After authentication, the user remains authenticated until the card is powered off or the application is re-selected. Authentication is performed by verifying the user PIN.

```
import UIKit
import Foundation
import Keycard

//the object that controls the NFC session
var session: KeycardController?

@IBAction func scanAction(_ sender: Any) {
   //the session is opened
   session = KeycardController(onConnect: onConnection, onFailure:
       onDisconnection)
   session?.start(alertMessage: "Hold your Keycard near iPhone.")
}


func onConnection(cc: CardChannel) {
   let cmdSet = KeycardCommandSet(cardChannel: cc)

   do {
      let info = try ApplicationInfo(cmdSet.select().checkOK().data)

      //pairing
      try cmdSet.autoPair(password: pp!)

      //opening secure channel
```

```
        try cmdSet.autoOpenSecureChannel()

        //pin verifying
        try cmdSet.verifyPIN(pin: pin!).checkAuthOK()

                                . . .

        //unpairing
        try cmdSet.autoUnpair()
    } catch let error {
        session?.stop(errorMessage: error.localizedDescription)
    }
    session?.stop(alertMessage: "Operation completed")
}
```

2.2.2 Authentication

When the app is launched, the first screen the user sees is the authentication screen (figure 4). Authentication needs PIN and pairing password verifying to be performed.
When the required fields are filled, the card can be scanned to verify the entered values. After this scan there are two possible scenarios:

1. the card is already initialized;
2. the card needs to be initialized.

So this is the first check the app makes. If the card is already initialized and the entered values are correct, the authentication succeeds and the user is brought to the app main screen; if it results that the card has not been initialized yet, then it needs to be done: an alert will appear, asking the user to insert a PIN, a PUK and a pairing password to be associated with the card. Once the new credentials are set, another scan is needed to load them into the card and then perform authentication.

```
//Card connection
func onConnection(cardChannel: CardChannel) {
    let cmdSet = KeycardCommandSet(cardChannel: cardChannel)

    do {
        //Preparation
        applicationInfo = try
            ApplicationInfo(cmdSet.select().checkOK().data)

        //control if card is already initialized
        if(!applicationInfo!.initializedCard) {
                    . . .
            //new credentials insertion
            insertCredentials()
```

```
            . . .
        //card gets initialized with new credentials
        try cmdSet.initialize(pin: pin!, puk: puk!, pairingPassword:
            pp!).checkOK()

      } else {
        print("Keycard already initialized")
      }

    } catch let error {
      session?.stop(errorMessage: error.localizedDescription)
    }

    session?.stop(alertMessage: "Keycard connected")
  }
```

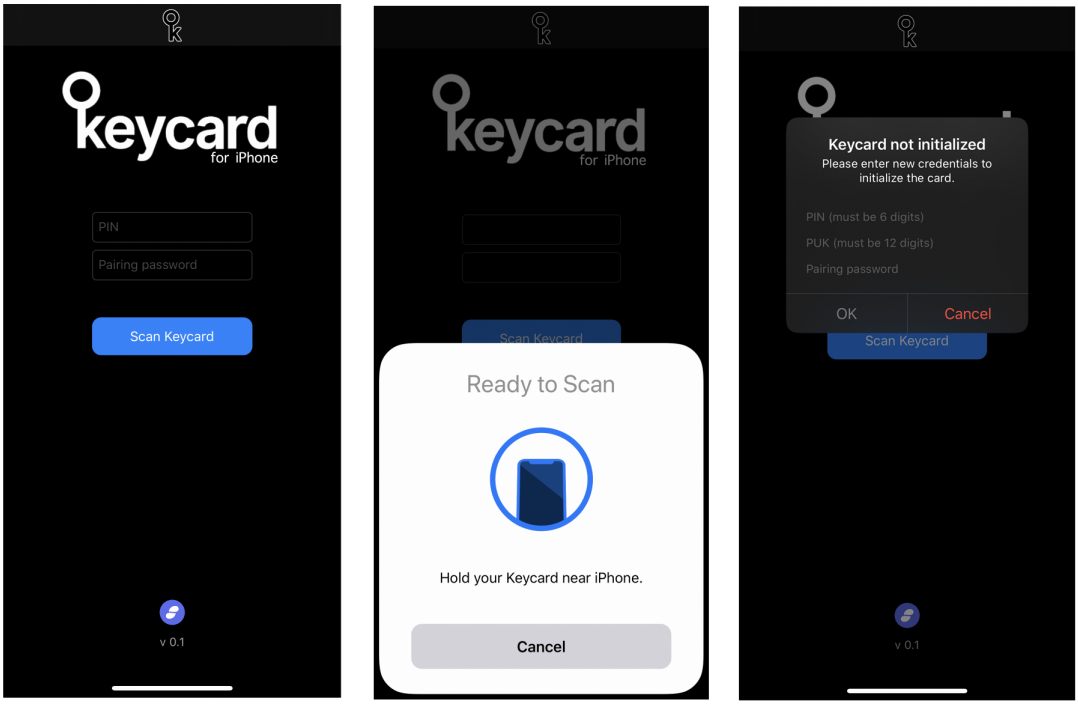


Figure 3: Authentication screen.

2.2.3 Info

Here some informations about the card are displayed:

- master key present: tells if the card has a master key, so if a wallet has alredy been created on the card;
- free pairing slots: the number of free pairing slots left;
- applet version: version of the applet installed on the card;
- instance UID of the applet: this can be used to identify this specific applet instance;
- master key UID: a value generated starting from the public key useful to identify if the card has the expected wallet.

All those informations can be obtained using the Keycard framework functions as follows:

```
func onConnection(cardChannel: CardChannel) {
                    . . .
   instanceUID = applicationInfo?.instanceUID
   appletVersion = applicationInfo?.appVersionString
   keyUID = applicationInfo?.keyUID
   hasMasterKey = applicationInfo.hasMasterKey
   freeSlots = applicationInfo.freePairingSlots
                    . . .
}
```

2.2.4 Operations

In this section all the main card functions are available for the user. Every operation needs the card to be presented and scanned in a NFC session; the sessions duration is variable, depending on the type of operation that is being performed.

Create wallet: to actually use the Keycard, it needs to have a wallet. This can be achieved in several different ways, which one the user choose depends on the usage scenario. Creating a wallet requires user authentication and is possible even if a wallet already exists on the card (the new wallet replaces the old one).

The available methods for wallets creation are:

1. Create BIP39 phrase;
2. Import BIP39 phrase;
3. Generate key on card;
4. Import EC keypair.

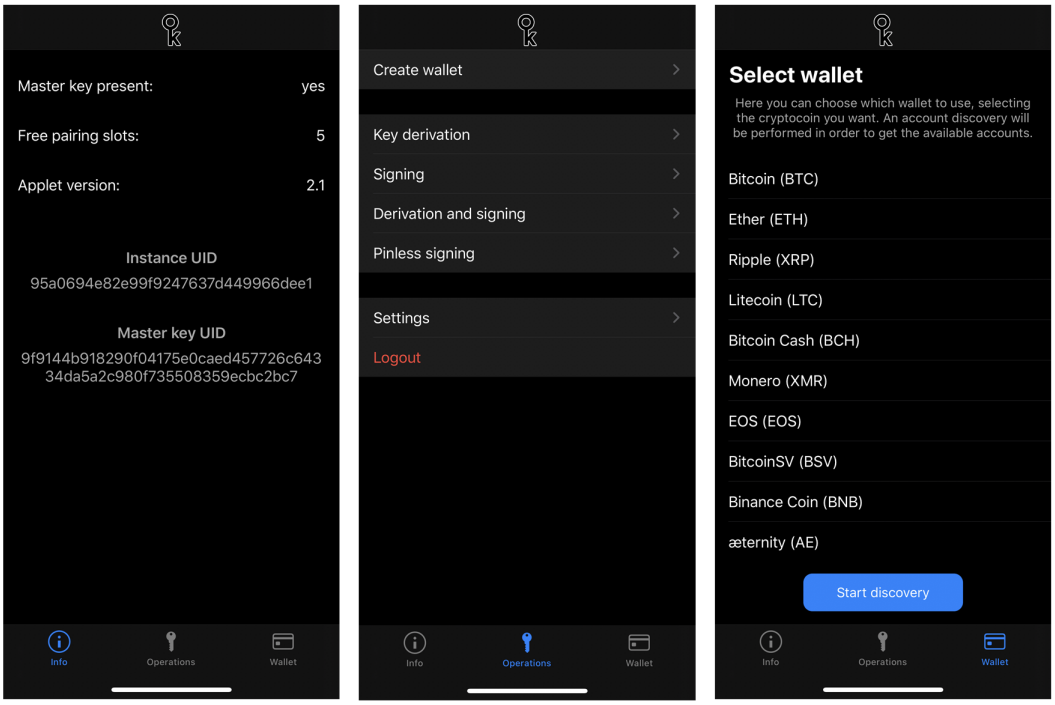


Figure 4: Application main screens.

1. With this option it's possible to generate BIP39 compliant mnemonic phrase, as the card feature a True Random Number Generator. Phrase generation doesn't require authentication, but loading the key derived from it does. The user can choose to create a phrase of 12, 15, 18, 21 or 24 words. The mnemonic phrase can be used to recover the wallet so, once generated, it should be written on paper and securely stored. After generated, the passphrase can be copied or exported via Mail, AirDrop, WhatsApp ecc.

The generation function uses the `Mnemonic` class of the Keycard framework, which will generate the phrase with words taken by an internal English word list compliant with the BIP39 standards.
`generateMnemonic` function takes as input an hex value that indicates the number of words to be generated.
For export the mnemonic is converted to readable format, while is converted to a BIP32 keypair when loaded on the card.

```
func onConnection(cardChannel: CardChannel) {
                    . . .
  //the user chooses how long the passphrase should be
```

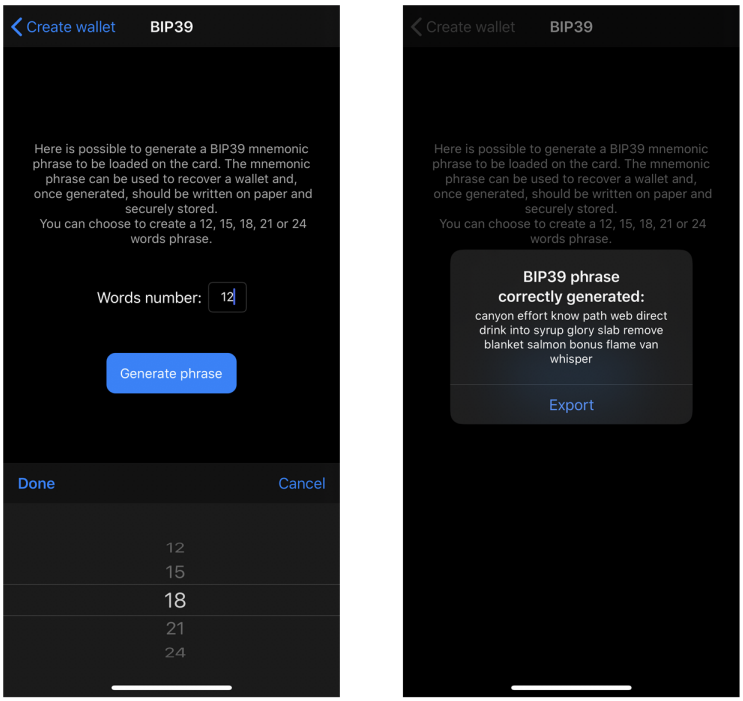


Figure 5: BIP39 passphrase generation.

```
switch wordsNumber {
   case 15:
      uint = 0x05
   case 18:
      uint = 0x06
   case 21:
      uint = 0x07
   case 24:
      uint = 0x08
   default:
      break
}

//the mnemonic is generated
let mnemonic = Mnemonic(rawData: try cmdSet.generateMnemonic(p1:
    uint).checkOK().data)
mnemonic.useBIP39EnglishWordlist()

generatedPhrase = mnemonic.toMnemonicPhrase()

//pin verification...
```

```
  //key loading
  try cmdSet.loadKey(keyPair: mnemonic.toBIP32KeyPair()).checkOK()
                    . . .
  //the generate passphrase can be shared
  share(str: (generatedPhrase), instance: self)

                    . . .
}
```

2. A wallet can even be imported through a BIP39 passphrase. When this option is selected, an alert will appear asking the user to insert the passphrase (all words separated by a single space) and the password (it can be any non-null string, usually is empty).
In this section multiple operations share the same scanning function, which will perform the required action dependin on the value of a flag. For BIP39 passphrase import the flag value is 0.

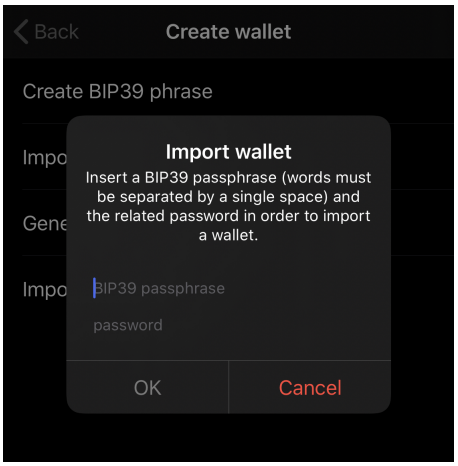


Figure 6: Wallet import through BIP39 passphrase.

```
func importPassphrase() {
  let alert = UIAlertController(title: "Import wallet", message:
      "Insert a BIP39 passphrase (words must be separated by a single
      space) and the related password in order to import a wallet.",
      preferredStyle: .alert)
```

```
  alert.addTextField { (passphrase) in
    passphrase.placeholder = "BIP39 passphrase"
  }

  alert.addTextField { (password) in
    password.placeholder = "password"
    password.isSecureTextEntry = true
  }

  alert.addAction(UIAlertAction(title: "OK", style: .default, handler:
      { [weak alert] (_) in
    self.keyOperation = 0  //flag setting
    self.importingPhrase = alert?.textFields![0].text
    self.pass = alert?.textFields![1].text
    self.beginScanning()  //call to scan function
  }))
  alert.addAction(UIAlertAction(title: "Cancel", style: .destructive))

                  . . .
}

func onConnection(cc: CardChannel) {
                  . . .

  if keyOperation == 0 {
    try cmdSet.loadKey(seed: Mnemonic.toBinarySeed(mnemonicPhrase:
        importedPhrase, password: importedPassword))
  }
                  . . .
}
```

3. This is the simplest and safest method, because the generated wallet never leaves the card and there is no "paper backup" to keep secure. When this option is selected a simple alert will show and the user can choose to start the session for generation or discard.
Code side the operation is very simple as it's done with a single instruction:

```
                  . . .
  //flag value for on-card key generation
   } else if keyOperation == 1 {
    try cmdSet.generateKey().checkOK()
  }
                  . . .
```

4. It's possible to import on the card any EC keypair on the SECP256k1

curve, with or without the BIP32 extension. If the user imports a key without the BIP32 extension, then key derivation will not work, but it will still be possible to use the Keycard for signing transactions using the imported key. This scenario can be useful when migrating from a wallet not using BIP39 passphrases or for wallets following some custom generation rules. It is however generally preferable to use one of the methods presented above.

```
                . . .
//flag value for EC keypair import
else if keyOperation == 2 {

  // privKey is the S component of the key, as a 32-byte long byte
      array
  // chainCode is the extension to the keypair defined by BIP32,
      this is another 32-byte long byte array. Can be null, in
  // which case the created wallet won't be BIP32 compatible.
  // pubKey is the DER encoded, uncompressed public key. Can be
      null, in which case it is automatically calculated from
  // the private key.

  BIP32KeyPair keypair = BIP32KeyPair(p1: privKey, chainCode:
      chainCode, p2: pubKey)

  // Loads the keypair
  cmdSet.loadKey(keypair).checkOK();
}
                . . .
```

Key derivation: derivation is used to select the active key to a specific BIP32 key path and to create a hierarchical deterministic wallet. The active key is persisted across sessions, meaning that a power loss or applet reselection does not reset it. When creating or importing a wallet to the Keycard, the active key is the master key. Unless a non-BIP32 compatible wallet was imported, the user usually wants to set the active key to a currency account by following the BIP44 specifications for paths. Note that the maximum depth of the key path is 10, excluding the master key.

In this screen the user can see the current active path. It's possible to choose the cryptocurrency to use for the new wallet from a list that includes all the major cryptocurrencies on the market (see 2.3 for all available cryptocurrencies). It's also possible to create a new account when performing the derivation: this will result in an increment of the corresponding index inside the path; the index increment happens even with the last element of the path, that indicates the address index: this index will be incremented at every new derivation (see **??** to more details about the paths).

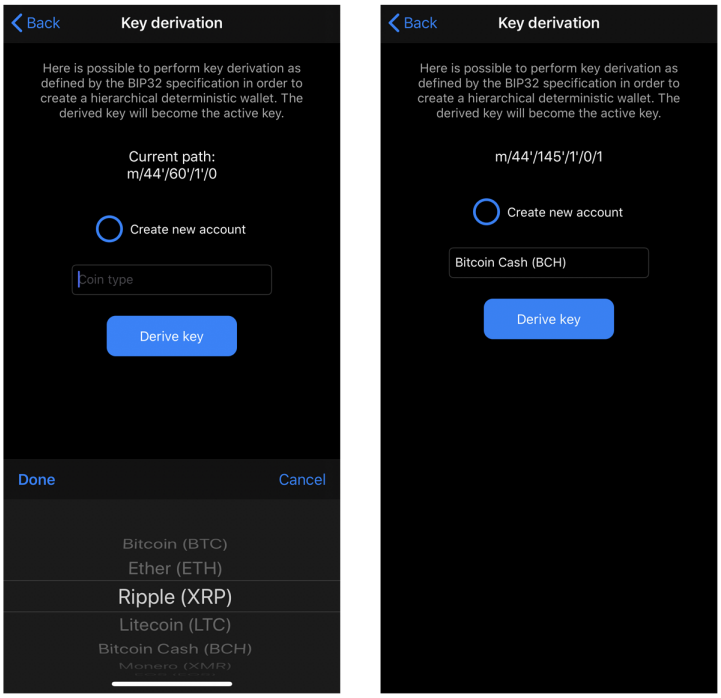


Figure 7: Key derivation. In this example the chosen coin was Bitcoin Cash so, after the derivation was complete, the current path coinType index changed according to the corresponding index in the registered coin type table (`https://github.com/satoshilabs/slips/blob/master/slip-0044.md`).

```
                . . .
//retrieving the current active key path
let keyPath = KeyPath(data: try cmdSet.getStatus(info:
    0x01).checkOK().data)

                . . .
//building the new path to derive with string interpolation
if currentKeypath != "m" {
   let splits = currentKeypath.components(separatedBy: "/")
   let addressIndex = (newAccountChecked) ? 0 :
       (Int(splits[splits.count - 1])!) + 1
   let accountIndex = Int(splits[3].replacingOccurrences(of: "'",
       with: ""))

   pathToDerive += (splits[0] + "/")
   pathToDerive += (splits[1] + "/")
   pathToDerive += "\(selectedCoin)'/"
   pathToDerive += (newAccountChecked) ? String(accountIndex! + 1) +
```

```
        "'/" : (String(accountIndex!) + "'/")
    pathToDerive += (splits[4] + "/")
    pathToDerive += String(addressIndex)
} else {
    pathToDerive = "m/44'/\(selectedCoin)'/0'/0/0"
}

//key derivation
try cmdSet.deriveKey(path: pathToDerive).checkOK()

                . . .
```

Deriving a key is an expensive operation, so in this case the NFC required session could last up to 5 seconds.

Signing: signing is the main goal of the card (it can handle anything which requires ECDSA signatures over SECP256k1 curve, regardless of the used hashing algorithm). In the app transaction are signed as follows:

```
                . . .
// hash is the hash to sign, for example the Keccak-256 hash of an
    Ethereum transaction
// the signature object contains r, s, recId and the public key
    associated to this signature
signature = try RecoverableSignature(hash: hash, data: try
    cmdSet.sign(hash: hash).checkOK().data)
                . . .
```

Signing requires user authentication.

Derivation and signing: it's possible to combine derivation and signing in a single step, and it's also possible to choose whether the given path becomes the current path or not. This section is intendend to more experienced users, as the path to derive can be manually inserted as a string, so a good knowledge of BIP44 is needed to use this function.
Derivation and signing are combined as follows:

```
                . . .
// hash is the hash to sign, the second argument is the path to use,
    in the same format as for the DERIVE KEY command
// the third argument is a flag indicating whether the derived key
    should become the current key or not
let resp: APDUResponse = try cmdSet.sign(hash: hash, path:
    pathTF.text!, makeCurrent: setToActive)
signature = try RecoverableSignature(hash: hash, data:
    (resp?.checkOK().data)!)
                . . .
```

Pinless signing: the card offers the option to define a BIP32 path which can be used to sign transactions without requiring a PIN. The SIGN command also has the option to sign with this pinless path (if defined) without having to know its value in advance. Since in such a scenario no sensitive data is being transferred, signing using the pinless path can be done without opening a Secure Channel.
Since the wallet assigned to pinless signing is basically unprotected, it must not hold great value. One option is to consider the funds in that account like pocket cash.
In order to complete a pinless signing a pinless path must have been defined. This can be done in the app settings, at "Define pinless path".

```
                . . .
//in the settings a pinless path must be set
try cmdSet?.setPinlessPath(path: path).checkOK()
                . . .
//then signing can be performed
try cmdSet.signPinless(hash: hash).checkOK()
```

Settings: here three possible operations are available:

- **change credentials**: this allows the user to set a new PIN, PUK and pairing password. Users must be very careful when changing credentials, as a typing error could led to impossibility to use the card;
- **define pinless path**: a BIP44 compliant path can be entered to be set as path for performing pinless transactions;
- **export keys**: it's possible to export any public key as well as the private key of keypaths defined in the EIP-1581 specifications. Those keys, by design, are not to be used for transactions but are instead usable for operations with lower security concerns. It's even possible to export the Ethereum address associated with the wallet on the card. Users can export keys/address of the current active key or can perform a derive and export operation: in this last case the user inserts the path he wants to derive, then the export is performed. Key or address can be exported through Mail, WhatsApp, AirDrop ecc. or can simply be copied to clipboard.

```
                . . .
//credentials change
try cmdSet.changePIN(pin: newPin!).checkOK()
try cmdSet.changePUK(puk: newPuk!).checkOK()
try cmdSet.changePairingPassword(pairingPassword:
    newPairingPass!).checkOK()

                . . .

//key export
```

```
//the flag selectedOption depends on the user's choice and defines
    wheter it will be performed a simple export or a derivatiion with
    export
  if self.selectedOption == 0 {
  //publicOnly function parameter is a boolean variable which
      defines if only the public key or even the private key must be
      exported (only allowed for key path following the EIP-1581
      definition)
  publicKey = try BIP32KeyPair(fromTLV: try
      cmdSet.exportCurrentKey(publicOnly:
      onlyCurrentKeypair!).checkOK().data)
} else if self.selectedOption == 1 {
  publicKey = try BIP32KeyPair(fromTLV: try cmdSet.exportKey(path:
      self.path!, makeCurrent: false, publicOnly:
      onlyCurrentKeypair!).checkOK().data)
}

            . . .
//depending on the user's choice, it will be exported the Ethereum
    address or the public key
if self.exportEthAddress {
  self.key = "0x" + (publicKey?.toEthereumAddress().toHexString() ??
      "null")
} else {
  self.key = publicKey?.publicKey.toHexString()
}

share(self.key)
```

2.2.5 Wallet

Here the user can choose which wallet to use, selecting the desired cryptocurrency from a list that includes all major currencies. Choosing a wallet means to select a wallet from the ones available that the application finds: a wallet is available if it has been used at least for one transaction. In order to get the available accounts, the app performs an account discovery: the algorithm is an implementation of the BIP44 Account Discovery algorithm:

1. derive the first account's node (index = 0);
2. derive the external chain node of this account;
3. scan addresses of the external chain respecting the gap limit;
4. if no transactions are found on the external chain, stop discovery;
5. if there are some transactions, increase the account index and go to step 1.

This algorithm is successful because software should disallow creation of new accounts if previous one has no transaction history.

In BIP44 it's suggested to set the address gap limit at 20. If the software hits 20 unused addresses in a row, it expects there are no used addresses beyond this point and stops searching the address chain.

In the development process this algorithm had to be adapted to the Keycard use case. BIP44 version finds available account referring to the BIP44 standard for HD wallet paths, where the account is indicated by the third numeric index of the path, while in Keycard for iPhone's version the aim is to find all Ethereum addresses derived from the card that are available. The discovery algorithm has then been implemented as Algorithm 1.

Algorithm 1 Account Discovery

```
1:  procedure startDiscovery
2:      addressesFound ← []
3:      availableAddresses ← []
4:      addr
5:      addressIndex ← 0
6:      loop:
7:      addr ← []
8:      for i ← addressIndex to gapLimit do
9:          address ← deriveAndExport(path[i])
10:         addressesFound.append(address)
11:     end for
12:     addr ← validAddressesIn(addressesFound)
13:     availableAddresses.append(addr)
14:     if sizeof(addr) == gapLimit then
15:         addressIndex ← addressIndex + gapLimit
16:         goto loop
17:     end if
18:     return availableAddresses
19: end procedure
```

There are three different arrays: `addressesFound` contains the addresses derived from the card, `availableAddresses` contains the addresses that have been used for at least one transaction (so are available) and `addr` is a support array. `addr` gets initialized as empty array at the beginning of the loop at step 7, then a for loop performs consecutive derivations (iteration number equals address gap limit) in order to get a group of possibly valid accounts; `addr` is then filled by the function `validAddressesIn()`, that takes as input the derived addresses and returns the available ones. `addr` is then appended to `availableAddresses`. On line 14 a control is made on `addr` size: in fact, if its size equals the address gap limit, it means that all the derived addresses are available, and there could be others deriving from paths that go over the last path derived in the for loop. For example: in the for loop are derived address from `m/44'/0'/0'/0/0` to `m/44'/0'/0'/0/9` (assuming that the address gap limit is 10); if all the addresses

obtained from those derivations are available, this means that could also be available the addresses deriving from path `m/44'/0'/0'/0/10` and above; so the derivation process must be performed again for n paths, where n is the address gap limit (see 2.3 at "Issues encountered" for details about the choice of gap limit value).
When the condition on line 14 is not satisfied, `availableAddresses` is returned.

In the app the function `validAddressesIn()` is implemented with the function `etherscanSearch()`: this function, for each element in `addressesFound[]`, performs an online search to check if the considered element can be appended to `availableAddresses`. The search is made with a GET request to an Etherscan API; Etherscan (`https://etherscan.io`) is a Block Explorer and Analytics Platform for Ethereum which gives access to some of its APIs for blockchain exploring. In this case the API exploited for the discovery process was:

```
http://api.etherscan.io/api?module=account&action=txlist&address=
[ethereumaddress]&sort=asc&apikey=[AKT]
```

where "AKT" is the Api Key Token, a token that is given to developers on Etherscan registration, necessary for using the APIs. This API returns a list of normal transactions by address, with a response in the json format:

```
 {
"status": "",
"message": "",
"result": []
}
```

where the transactions list is returned in the "result" array, and the attribute "status" assumes value "1" if the size of result is greater or equal then 1; so, an addres is considered valid if, when the Etherscan API is called on it, the "status" attribute of the respons is "1".

Once the discovery is complete, if no available addresses were found then it won't be possible to select a wallet, otherwise a list of the available addresses will be presented and the user wil choose the one to use by loading it on the card.

2.3 Additional informations

The project was organized following the MVC pattern and fully written in Swift 5, using Xcode on macOS Catalina. The application was successfully tested on iPhone 7, iPhone X, iPhone XS and iPhone 11.

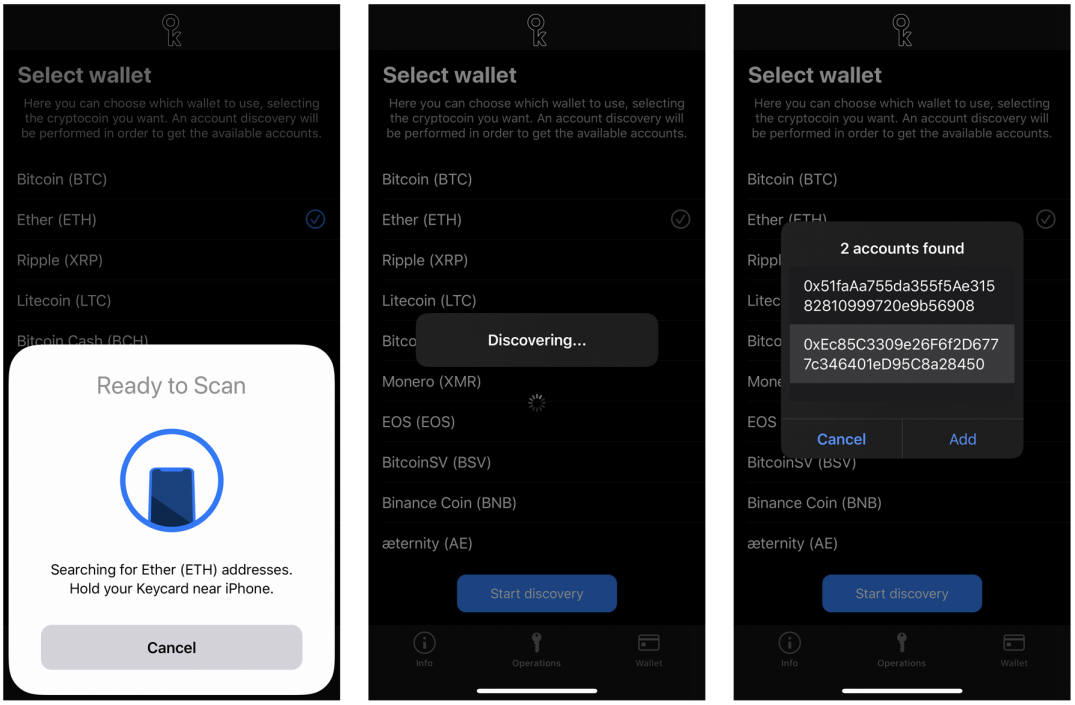


Figure 8: Wallet selection through account discovery.

2.3.1 Available currencies

For wallet selection and key derivation at the moment the available cryptocurrencies are:

- Bitcoin (BTC)
- Ether (ETH)
- Ripple (XRP)
- Litecoin (LTC)
- Bitcoin Cash (BCH)
- Monero (XMR)
- EOS (EOS)
- BitcoinSV (BSV)
- Binance Coin (BNB)

- æternity (AE)
- Lisk (LSK)
- NEO (NEO)

2.3.2 Usage requirements

- iOS 13 or later;
- a Keycard with at least one free pairing slot;
- it's not a requirement to use iPhone 8 or later, but issues were encountered with iPhone 7 during the development.

2.3.3 Encountered issues

As the Keycard Swift framework is not as supported from the Status team as the Java version, some issues emerged during the development. Those included minor bugs in some functions but even more important problems. Where possible, action was directly taken by modifying the framework code. Major issues derive from the card applet: for example, the BIP44 standard suggests the address gap limit to be 20; this wasn't possible in the app implementation, as all the consecutive derivations must be performed on the card, it's not possible to use a local algorithm, and the derivation process is very expensive (a single derivation can take up to 3 seconds to be completed). This means that an address gap limit of 20 would mean to take 20 consecutive derivations, for a total required time that could go up to 60 seconds, an amount of time not compatible with the timeout of a NFC session (20 seconds). A workaround for this problem was to use the most efficient form of the derivation function (derivation not starting from the master key, for example "`m/44'/0'/0'/0/0`", but instead from the parent of the current key, for example "`../1`") and to lower the address gap limit to 10. In this way the discovery it's possible as the NFC session timeout is not exceeded.
Another problem was given by the pairing slots. As already said, the card can be paired only with five clients at once, so if no free pairing slots are available then the card can't be used. The problem is that, as the card is not fully stable yet and it's only intended for developers, NFC communication does not happens at its best: it will often happen that the session will be interrupted with the message "Tag connection lost". This error does not depend from the app. If this happens after a pairing was made but before the unpairing is done, this will led to the loss of a pairing slot. During the development, two cards ran out of free slots and went unusable (a reinstallation of the applet was necessary). The loss of slots can be prevented by pairing the card with the device and not unpair it, internally storing the pairing info that will e used in the NFC sessions, but, in a real use case of the app, a user may not want to pair its card with an untrusted client.
As mentioned in the usage requirements, issues were encountered when testing

the app with iPhone 7: the NFC session would fail 9 times out of 10, making the app (and the card) totally unusable. Not even this problem was attributable to the application, as it didn't exist when testing with iPhone 8 and newer. By doing some research it turned out that NFC does not properly work with iPhone 7 running iOS 13.2, so the only possible workaround was to change the devices for testing.