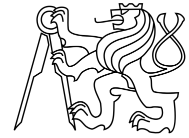
Czech Technical University in Prague
Faculty of Information Technology



ASSIGNMENT OF BACHELOR’S THESIS

Title:	Bitcoin Wallet for Android with TREZOR Hardware Wallet Support
Student:	Matouš Skála
Supervisor:	Mgr. Jan Starý, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2018/19

Instructions

1) Describe key derivation in a hierarchical deterministic wallet as defined by the BIP-0032 specification
2) Describe the Bitcoin transaction structure and signing workflow when using the TREZOR hardware wallet
3) Design an Android library for communication with TREZOR over an USB interface
4) Implement a Bitcoin Wallet for Android with at least the following functionality:
   a) Load the wallet content from TREZOR
   b) Create a transaction, sign with TREZOR and broadcast to the Bitcoin network
   c) Account, address and transaction labeling compatible with the SLIP-0015 standard

References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

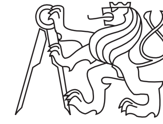
doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague October 30, 2017

Czech Technical University in Prague

Faculty of Information Technology

Department of Theoretical Computer Science



Bachelor's thesis

Bitcoin Wallet for Android with TREZOR Hardware Wallet Support

Matouš Skála

Supervisor: Mgr. Jan Starý, Ph.D.

14th May 2018

Acknowledgements

I would like to thank to Jiří Charvát for introducing me to Bitcoin in the first place, to Mgr. Pavol Rusnák for consultation regarding the mechanics of communicating with TREZOR, to SatoshiLabs s.r.o. for providing testing TREZOR devices, and finally to Mgr. Jan Starý, Ph.D. for giving continuous feedback during the whole process.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 14th May 2018 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2018 Matouš Skála. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Skála, Matouš. *Bitcoin Wallet for Android with TREZOR Hardware Wallet Support*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Tato práce se zabývá návrhem Bitcoinové peněženky pro OS Android, která využívá zařízení TREZOR jako úložiště privátních klíčů. Navržená aplikace umožňuje zobrazit seznam transakcí, vytvořit novou transakci, podepsat ji pomocí zařízení a odeslat do sítě. Vedlejším přínosem této práce je navržení Android knihovny pro usnadnění komunikace se zařízením TREZOR.

Klíčová slova mobilní aplikace, Android, Bitcoin, kryptoměna, hardwarová peněženka, TREZOR, kryptografie, Kotlin

Abstract

The goal of this thesis is to design a Bitcoin wallet for the Android OS, using the TREZOR device as a secure private key storage. The implemented application allows to see transactions history, to compose a new transaction, sign it and broadcast it to the network. The secondary contribution of this thesis is designing an Android library simplifying the communication with the TREZOR device.

Keywords mobile application, Android, Bitcoin, cryptocurrency, hardware wallet, TREZOR, cryptography, Kotlin

Contents

Introduction 1

1 Analysis and design 3

1.1 Public Key Cryptography in Bitcoin . . . 3
1.2 Hierarchical Deterministic Wallet . . . 8
1.3 Transaction Structure . . . 12
1.4 Transaction Scripts . . . 14
1.5 Segregated Witness . . . 16
1.6 TREZOR Hardware Wallet . . . 19

2 State of the art 23

2.1 Existing Wallets with TREZOR Support . . . 23
2.2 TREZOR Communication Library for Android . . . 25

3 Realisation 27

3.1 TREZOR Intents Library . . . 28
3.2 Account Discovery . . . 30
3.3 Bitcoin Network Connection . . . 30
3.4 Transaction Composition . . . 31
3.5 Bitcoin Metadata . . . 35

Conclusion 39

Future Work . . . 40

Bibliography 41

A Screenshots 45

B Acronyms 51

C Contents of enclosed SD 53

List of Figures

1.1 Addition of points P and Q on an elliptic curve 4
1.2 Base58Check Encoding 6
1.3 Hierarchical Deterministic Wallet 9
1.4 Signature workflow for TX with N inputs and M outputs 22

2.1 Mycelium 24
2.2 Sentinel 25

A.1 Initialization 46
A.2 PIN Request 46
A.3 Accounts List 47
A.4 Transactions List 47
A.5 Receiving Addresses 48
A.6 Transaction Composition 48
A.7 Address Detail 49
A.8 Transaction Detail 49

List of Tables

1.1 Transaction Format 13
1.2 Transaction Output Format 13
1.3 Transaction Input Format 13
1.4 P2PKH `scriptSig` Execution 15
1.5 P2PKH `scriptPubKey` Execution 15

List of Source Codes

1 Parent public key to child public key derivation 11
2 Construction of a P2SH-P2WPKH address from a public key . 18
3 FIFO Coin Selector 34
4 An example metadata JSON object 35

Introduction

Cryptocurrencies have been a frequently discussed topic recently. They are revolutionary mainly because they allow nearly instant, costless funds transfer between users worldwide. Importantly, it can all be done without the need to trust any intermediaries.

However, with increasing cryptocurrency popularity among users, there is also an increasing risk of having the funds stolen as a result of insufficient system security. To deal with this issue, hardware wallets have emerged, allowing users to keep private keys stored on an offline device, separately from a vulnerable operating system. When sending a transaction, the user has to connect the device to a computer over the USB interface and confirm the transaction signature by physically pressing a button. The signing process happens in the device itself and the private key never leaves the device.

The outcome of this thesis will be beneficial to users who own the TREZOR hardware wallet produced by SatoshiLabs and would like to manage their Bitcoin wallet from an Android smartphone.

I have chosen this topic because of my interest in decentralized systems, cryptography and particularly cryptocurrencies. Even though some Android wallets with TREZOR support already exist, they usually support only a subset of features from the full-featured desktop wallet and do not reflect recent Bitcoin development.

This thesis is dealing with a design and implementation of an Android application, which serves as a Bitcoin wallet and supports signing transactions with TREZOR. The app supports a hierarchy of accounts and a new SegWit transaction format. It also enables the user to label accounts, addresses, and transactions and synchronizes these metadata with Dropbox cloud storage.

The project begins with an analysis of Bitcoin wallets technology, a transaction structure and TREZOR signature process. Secondly, a library for communication between an Android device and TREZOR is designed. Finally, the Bitcoin wallet is implemented, demonstrating the usage of the library.

Chapter 1

Analysis and design

1.1 Public Key Cryptography in Bitcoin

Public key cryptography, or *asymmetrical cryptography*, is the foundation of computer security. In a public key cryptography system, the user owns a pair of keys: a *private key* known only by the owner, and a *public key* that can be shared with other parties. Such a system can be used either for encryption, where only the private key can be used to decrypt a message encrypted with the paired public key, or authentication, where a public key can verify if the message has been signed with the paired private key.

In Bitcoin [1], cryptography is used to control the ownership of funds. The user owns a private key, which is stored in a file called a *wallet*. Each private key has a corresponding public key that is used to derive a Bitcoin *address*. When the user wants to receive funds, he shares his address with a counterpart, who uses it as an output in a new transaction and broadcasts the transaction to the network.

When the user wants to spend the received funds, he can use them as an input in a next transaction. To prove ownership, he has to sign the transaction with his private key and provide the signature along with the corresponding public key. All network participants can then validate the authenticity of the transaction.

Contrary to popular belief, encryption is not a part of Bitcoin and all transaction details are publicly available in the *blockchain* (an immutable linked list of transaction blocks). To protect users' privacy, other cryptocurrencies have emerged, e.g. *Zcash*[1] or *Monero*[2]. In future, it's possible to further enhance Bitcoin anonymity with the implementation of *Confidential Transactions* [2], which ensure that the amount of funds transferred is visible only to the participants of the transaction while preserving the ability of the network to verify the transaction validity.

[1] https://z.cash/

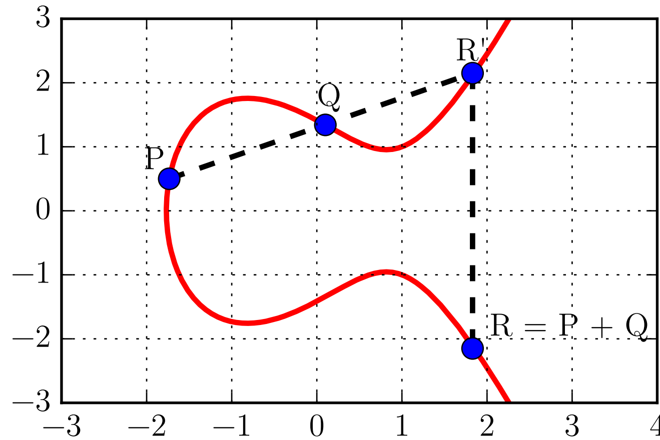
[2] https://getmonero.org/

1.1.1 Elliptic Curve Cryptography

Elliptic curve cryptography [3] is a type of asymmetric cryptography based on the algebraic structure of elliptic curves over finite fields. An elliptic curve is a plane curve consisting of the points satisfying the equation $y^2 = x^3 + ax + b$, along with a *point at infinity* $\mathcal{O}$.

We can define an addition operation for points on the curve. Given two points P and Q on the elliptic curve, there is a third point $R = P + Q$, also on the elliptic curve. Geometrically, R can be calculated by drawing a line between P and Q. The line will intersect the elliptic curve in exactly one new point $R' = (x, y)$. We reflect that point over the x-axis and get the point $R = (x, -y)$.

Figure 1.1: Addition of points P and Q on an elliptic curve



If P and Q are the same point, the line between them is a tangent on the curve at that point. It will intersect the curve in exactly one new point. If P and Q have the same x values but different y values, the tangent will be vertical and in such case, we set $R = \mathcal{O}$ (the point at infinity). If $P = \mathcal{O}$, then $P + Q = Q$. This shows how $\mathcal{O}$ acts like zero for addition.

We can also define a scalar multiplication as repeated addition of a point on the curve. Given a point P on the elliptic curve and a whole number k, we can calculate

$$k * P = P + P + ... + P \ (k \text{ times})$$

For cryptographic purposes, elliptic curves over finite fields, instead of real numbers, are used. Those are elliptic curves that consist only of the points having coordinates in a finite field F_p.

Bitcoin uses a specific elliptic curve called **secp256k1** defined in *Standards for Efficient Cryptography* [4]. It specifies the curve parameters as $a = 0$, $b = 7$, therefore the curve can be defined by the equation $y^2 = x^3 + 7$ over F_p. The finite field is defined by the prime order $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$. The specification also defines a *generator point* G.

1.1.2 Private and Public Keys

In Bitcoin, the private key k is a 256-bit number, usually picked at random or derived from a random seed. The public key K is calculated from the private key using elliptic curve multiplication:

$$K = k * G$$

where k is the private key, G is the generator point and K is the resulting public key, represented by a point on the elliptic curve. It's important to note that the elliptic curve multiplication is a cryptographic one-way function, therefore while it's easy to calculate the public key from the private key, it's nearly impossible to calculate the private key from the public key. The reverse operation, also known as the *discrete logarithm problem*, is as difficult as a brute-force search.

1.1.3 Bitcoin Address

A *bitcoin address* [5] is a string of alphanumeric characters representing a possible destination for a Bitcoin payment and can be shared with anyone who wants to send you money. There are currently three address formats in use:

1. **P2PKH** (Pay to Public Key Hash), starting with the number `1`
2. **P2SH** (Pay to Script Hash), starting with the number `3`
3. **Bech32**, a native SegWit address format, starting with `bc1`

Let's start with the first case, where an address represents an owner of a private/public key pair. We can derive an address from the public key using a *hash function*. A hash function is a cryptographic one-way function that produces a fingerprint of an arbitrary-sized input. The specific algorithms used are **SHA256** (*Secure Hash Algorithm*) and **RIPEMD160** (*RACE Integrity Primitives Evaluation Message Digest*).

Starting with a public key K, we first apply the SHA256 function, then calculate RIPEMD160 and end up with a 160-bit number:

$$A = \text{RIPEMD160}(\text{SHA256}(K))$$

Finally, the address is encoded using the *Base58Check encoding*, which uses 58 characters and a checksum to improve readability and to protect against errors in address transcription.

1.1.4 Base58Check Encoding

To represent long numbers in a compact way, many computer systems use representations with a base higher than 10. For example, *Base64* representation uses both lowercase and capital letters, numerals and two additional

characters. Numbers encoded in such a way are more compact than those in decimal representation, as each Base64 digit represents exactly 6 bits of data.

For usage in Bitcoin, a new encoding scheme called *Base58* has been developed. It is similar to Base64, but it uses only alphanumeric characters and some letters that look ambiguous have been omitted. Compared to Base64, the following characters are excluded: 0 (zero), O (capital o), I (capital i), l (lower case L), + (plus) and / (slash).

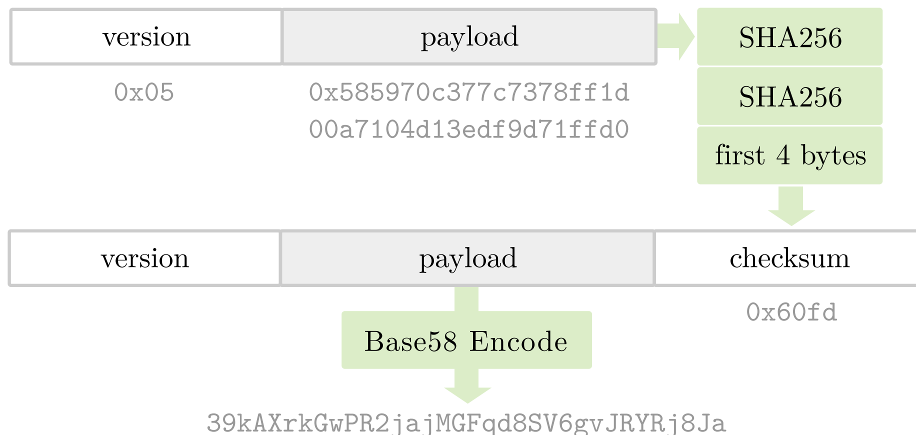
Base58Check is an encoding format extending Base58 with a version number and an error checking code. To convert a number into the Base58Check format, we first add a prefix called the *version byte*, identifying the type of data encoded. For a P2PKH address, a `0x00` prefix is used, resulting in a prefix of `1` when encoded with Base58. For P2SH addresses, the version byte is `0x05`, resulting in a prefix of `3`.

Next, we compute a double hash of a version byte concatenated with the actual data (a *payload*):

$$hash = \text{SHA256}(\text{SHA256}(version + payload))$$

We take only the first 4 bytes of the resulting hash and append it as a *checksum*. Finally, we encode the result consisting of three items (the version, the payload, and the checksum) using Base58 encoding. The whole encoding process is visualized in Figure 1.2.

Figure 1.2: Base58Check Encoding



During the decoding process, we calculate the checksum from the data and compare it to the checksum included in the code. If they do not match, an error has been introduced and the data is invalid.

1.1.5 Digital Signatures

A *digital signature* is a scheme that is used to ensure message *authenticity*. It proves that the message came from the stated sender and has not been modified. The scheme consists of two components – a *signing algorithm* and a *verification algorithm*.

Given a message m and a private key k, the signing algorithm produces a digital signature s.

The verification algorithm then takes a message m, a signature s, a public key K and verifies whether the private key corresponding to the public key K has been used to produce the signature s of the message m. If a different key was used or the provided message does not correspond to the signed message, the verification fails.

In Bitcoin, *Elliptic Curve Digital Signature Algorithm* (ECDSA) is used to ensure that funds can be spent only be their owners. To sign a message m with a private key k, using an elliptic curve over F_p with a generating point G of order n, these steps are followed:

1. Calculate a hash z of the message m

2. Select a random number t from $[1, n-1]$ which is used as an *ephemeral* (temporary) private key

3. Generate an ephemeral public key $P = t * G$

4. Set R to be the x coordinate of the ephemeral public key P

5. $S = t^{-1} * (z + k * R) \bmod p$

6. The signature is the pair (R, S)

To verify the signature, an inverse function is used to calculate a value P from the signature (R, S), a public key K, a message hash z and a generator point G:

$$P = S^{-1} * z * G + S^{-1} * R * K$$

If the x coordinate of the calculated point P is equal to R, the signature is valid, otherwise it's not.

It's crucial to make sure the number t is randomly generated. If a single ephemeral private key was used to sign multiple messages, it would be trivial to calculate the private key given the signature and the public key. An improper random-number generator initialization has led to some serious security flaws in the past. [6]

1.2 Hierarchical Deterministic Wallet

A Bitcoin *wallet* is a program that manages user's keys and addresses, allows to track the balance and to create and sign transactions. It's a common misconception that the wallet stores user's bitcoins, while it only keeps track of his private and public key pairs. Therefore, it would be more accurate to refer to it as a *keychain*. The coins itself are represented by the outputs of transactions stored in the blockchain and they can be unlocked by providing a valid signature.

In the early days of Bitcoin, the wallets generated all private keys randomly, which meant that for each new address, a new key pair had to be generated and stored independently. That made wallets difficult to back up, as with each new generated key, a new backup had to be created. Such a wallet is called *nondeterministic*, or *JBOK*[3].

To make the wallet backup easier, an idea of a *deterministic* wallet has been introduced, in which all private keys are generated from a single *root seed*. It is enough to back up the seed during the wallet initialization and then all keys can be recreated at any time just by providing the seed. Moreover, thanks to elliptic curve mathematics, a parent public key can generate a sequence of child public keys without knowing the private key. This enables a concept of *watch-only* wallets and it is particularly useful when combined with a hardware wallet.

The most advanced wallet form is a *hierarchical deterministic* (HD) wallet defined by the BIP-0032 standard [7], where keys are derived in a tree structure, such that each parent key can generate a sequence of child keys, as shown in Figure 1.3. The advantage of the tree structure is that different branches can be used for different purposes, e.g. one branch for incoming payments and another one to receive change from outgoing payments. Additionally, a structure of multiple accounts can be set up, e.g. to distinguish a personal and savings account, or to represent different accounting categories or departments in corporate settings.

1.2.1 HMAC

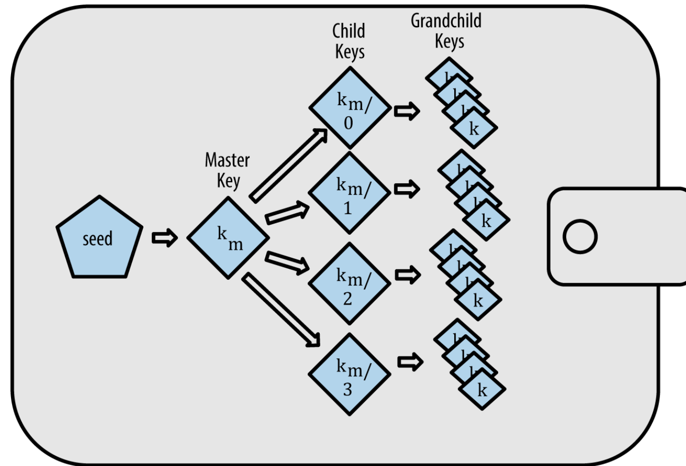
HMAC [8] stands for a *key-hashed message authentication code*. It's defined as a cryptographic hash of a message m combined with a secret key K:

$$\mathrm{HMAC}(K, m) = H((K \oplus opad || H((K \oplus ipad) || m))$$

H can be any cryptographic hash function, $ipad$ and $opad$ are predefined constants, $||$ denotes concatenation, and $\oplus$ represents XOR. Usually, HMAC is used as a mechanism for message authentication, but in our context, it's a building block of key derivation functions.

[3] Just a Bunch Of Keys

Figure 1.3: Hierarchical Deterministic Wallet [5]



1.2.2 Master Key Generation

Keys in an HD wallet are generated from the root seed, which can be a 128-, 256- or 512-bit random number. The root seed is first hashed with HMAC-SHA512, taking the "Bitcoin seed" string as the key. The resulting 512-bit hash is used to create a *master private key* m and a *master chain code* c by taking its left 256 bits and right 256 bits, respectively. A *master public key* M is then generated from the master private key using elliptic curve multiplication: $M = m * G$.

1.2.3 Extended Keys

The chain code is used to introduce entropy in the child key derivation function. We define an *extended private key* (k, c) as a private key k extended with a chain code c. An *extended public key* (K, c) is defined as a public key K extended with a chain code c.

Given a parent extended key and a child index i, it is possible to compute the corresponding child extended key. Each extended key has 2^{31} normal child keys and 2^{31} *hardened* child keys. The normal child keys use indices 0 to $2^{31}-1$ and hardened child keys use indices 2^{31} to $2^{32} - 1$.

1.2.4 Key Derivation

The child derivation function definition depends on whether we are working with public or private keys and whether these keys are hardened. An extended private key can be used to derive any child private and public keys (both hardened and normal), whereas an extended public key can derive only normal (non-hardened) child public keys.

For the purpose of our Bitcoin wallet, we will only need to derive child public keys from a parent public key. The derivation algorithm starts by checking if the child index is non-hardened. Then, a parent public key is encoded and concatenated with a child index. The resulting string is hashed with HMAC-SHA512, using a chain code as the key. The left 256 bits are used to create a child public key by multiplying the generator point and adding it with the parent public key. The right 256 bits are representing the child chain code. There is an implementation in the Kotlin language provided in Source Code 1.

1.2.5 Multi-Account Hierarchy for Deterministic Wallets

Keys in an HD wallet are identified by a path that was used for their derivation. To make different wallets compatible with each other, the BIP-0043 [9] and BIP-0044 [10] standards have been proposed, defining the logical hierarchy that should be used for key derivation.

BIP-0043 proposes that the first level of the tree structure should serve as a *purpose*. The purpose determines the further structure beneath that node. Each derivation scheme should be described in a separate BIP and its number should be used in the purpose field.

BIP-0044 defines the following 5 levels:

```
m / purpose' / coin_type' / account' / change / address_index
```

An apostrophe in the path indicates that the hardened derivation is used, so a private key is required to derive the child node.

Each level in the path has a special meaning:

Purpose is a constant set to 44, indicating that the path is following the scheme defined by the BIP-0044 specification.

Coin type specifies a cryptocurrency type, as one master seed can be used for multiple cryptocurrencies. A list of registered coin types is maintained in SLIP-0044 [11] and Bitcoin is defined as 0.

Account allows separating funds similarly to bank accounts. The wallet never mixes coins across different accounts. Accounts are numbered from the index 0. The wallet should prevent the creation of a new account if the previous account has no transactions history.

Change is a constant of 0 for an *external chain* (publicly available addresses for receiving payments) and a constant of 1 for change addresses.

Index specifies an address index, numbered from 0 in an increasing manner. The wallet should not allow creating more than 20 (*address gap limit*) consecutive addresses without any transaction history.

```
fun deriveChildKey(parentKey: ExtendedPublicKey, index: Int):
        ExtendedPublicKey {

    if (isHardened(index)) {
        throw IllegalArgumentException(
            "Defined only for non-hardened child keys")
    }

    val key = parentKey.chainCode
    val data = ByteBuffer.allocate(37)
    data.put(parentKey.publicKey.getEncoded(true))
    data.putInt(index)
    val i = hmacSha512(key, data.array())

    // Split 64-bit i into two 32-bit sequences
    val il = BigInteger(1, Arrays.copyOfRange(i, 0, 32))
    val ir = Arrays.copyOfRange(i, 32, 64)

    val curveParams = CustomNamedCurves.getByName(SECP256K1)

    if (il > curveParams.n) {
        throw InvalidKeyException(
            "il is larger that the curve order")
    }

    val childPublicKey = FixedPointCombMultiplier()
        .multiply(curveParams.g, il)
        .add(parentKey.publicKey)

    if (childPublicKey.isInfinity) {
        throw InvalidKeyException(
            "Child public key point is at infinity")
    }

    return ExtendedPublicKey(childPublicKey, ir)
}
```

Source Code 1: Parent public key to child public key derivation

1.3 Transaction Structure

A transaction is the fundamental part of Bitcoin. It serves as a transfer of Bitcoin value between users. Once the user creates and signs a transaction, it can be broadcast to the network. After that, it is validated by all full nodes, added to the unconfirmed transactions pool (*mempool*) and eventually included in a block by *miners*. There is a new block mined approximately every 10 minutes. The miner who created the block receives a *block reward* (currently 12.5 BTC, halving every 4 years) and the transaction fees of all included transactions.

1.3.1 Transaction Outputs and Inputs

The basic building block of a transaction is a *transaction output*. It represents an indivisible amount of Bitcoin recorded in the blockchain. Outputs that have not been spent yet are known as *unspent transaction outputs*, or *UTXO*. All Bitcoin full nodes keep track of the UTXO set and use it for validating incoming transactions. Every transaction spends some previously unspent outputs and creates new ones.

A transaction can have any number of inputs and outputs. Each output specifies an amount and has an associated *locking script* (`scriptPubKey`), which specifies the conditions under which the output can be spent. An input contains a reference to some previous transaction output, also known as an *outpoint* (a structure consisting of a transaction hash and an output index). The input also provides an *unlocking script* (`scriptSig`) that satisfies the conditions placed by the corresponding output locking script.

The transaction output value is indivisible, which means it has to be spent in whole. If the output is larger than the desired value of the transaction, a change output is created, sending the remaining value back to the sender's address. As an example, Alice sends 5 BTC to Bob. Later, Bob wants to send 1 BTC to Charlie. He has to use the whole 5 BTC output as a transaction input and create 2 transaction outputs – 1 BTC output going to Charlie and 4 BTC change output going back to Bob's wallet.

All transactions are stored and transferred in a binary format described in Table 1.1, 1.2 and 1.3. All multi-byte integers are in little-endian order.

1.3.2 Transaction Fees

It can be seen that there is no field for specifying a transaction fee in the transaction structure. Instead, the fee is calculated implicitly as a difference between the sum of transaction inputs and the sum of transaction outputs:

$$fee = sum(inputs) - sum(outputs)$$

The fee serves two purposes. Firstly, it's an incentive for miners to actually include transactions into blocks, instead of just mining empty blocks. Today,

Size	Field	Description
4 bytes	version	Transaction version number, currently 1
1–9 bytes	inputs count	Number of transaction inputs
variable	inputs	Transaction inputs
1–9 bytes	outputs count	Number of transaction outputs
variable	outputs	Transaction outputs
4 bytes	lock time	A timestamp or block number

Table 1.1: Transaction Format

Size	Field	Description
8 bytes	amount	Value in satoshis (10^{-8} BTC)
1–9 bytes	locking script size	The length of the locking script in bytes
variable	locking script	Conditions for spending the output

Table 1.2: Transaction Output Format

Size	Field	Description
32 bytes	transaction hash	Pointer to TX containing the UTXO
4 bytes	output index	The index of the output to be spent
1–9 bytes	unlocking script size	The size of unlocking script in bytes
variable	unlocking script	A script that satisfies the locking script
4 bytes	sequence number	Used for updating unconfirmed TX

Table 1.3: Transaction Input Format

fees represent only a fraction of miner's income, but as the block reward is gradually decreasing over time, fees will eventually become the primary incentive for miners.

Secondly, it's a protection against a *flood attack*, in which an attacker would flood the network with spam transactions, preventing real transactions from being included in blocks. Each block can only fit 1 MB of transactions. When miners are deciding which transactions to include in the next block, transactions are sorted by **fee per byte** and those with the highest fee are prioritized. That makes the flood attack expensive and ineffective.

The fee is dependent on the actual transaction size in bytes rather than its value, so transactions with multiple inputs and outputs will be usually more expensive than a simple transaction with a single input.

1.4 Transaction Scripts

In the previous section, we have mentioned the concept of the locking and unlocking scripts. Now we will introduce the script language and explain how these scripts are executed during the transaction validation.

Both locking and unlocking scripts are written in a language called *Script* [5]. It's a *stack-based* language similar to *Forth*. Scripts are processed from left to right and they consist of constants and operators. Constants are pushed onto the stack. Operators can pop one or more items from the stack, perform an operation with them and push the result back onto the stack.

When a transaction is being validated, for each input, a locking script is executed together with the corresponding unlocking script to check if the spending conditions are satisfied. The transaction is valid if the result on the stack is `TRUE`, any non-zero value or if the stack is empty.

This scripting language is the reason why Bitcoin is sometimes being referred to as *programmable money*. It can be used to express a vast amount of conditions, but it's intentionally not *Turing-complete*. It does not contain any loops to make execution times predictable because each transaction must be validated by all nodes in the network.

1.4.1 Pay to Public Key Hash (P2PKH)

Most of the transactions nowadays are based on **Pay to Public Key Hash** script, which locks the output to a public key hash:

```
OP_DUP OP_HASH160 <Public Key Hash> OP_EQUALVERIFY OP_CHECKSIG
```

To unlock such an output, an unlocking script must provide a signature and a public key corresponding to the private key used to make that signature:

```
<Signature> <Public Key>
```

Tables 1.4, 1.5 show a step-by-step execution of the P2PKH locking and unlocking script.

1.4.2 Pay to Script Hash (P2SH)

Pay to Script Hash transaction type was standardized in BIP-0016 [12]. It allows the user to send funds to the hash of the locking script, which is here being referred to as a *redeem script*. Usually, the recipient constructs the redeem script and provides its hash to the sender. No matter how complex the script itself is, the sender just needs to know its hash to construct the locking script:

```
OP_HASH160 <Redeem Script Hash> OP_EQUAL
```

Stack	**Remaining Script**	**Description**
	`<Signature>` `<Public Key>`	
`<Signature>`	`<Public Key>`	`<Signature>` pushed onto the stack.
`<Public Key>` `<Signature>`		`<Public Key>` pushed onto the stack.

Table 1.4: P2PKH `scriptSig` Execution

Stack	**Remaining Script**	**Description**

`<Public Key>`
`<Signature>`
	`OP_DUP`
	`OP_HASH160`
	`<Public Key Hash>`
	`OP_EQUALVERIFY`
	`OP_CHECKSIG`	

`<Public Key>`
`<Public Key>`
`<Signature>`
	`OP_HASH160`
	`<Public Key Hash>`
	`OP_EQUALVERIFY`
	`OP_CHECKSIG`
		The top stack item duplicated.

`<Public Key Hash>`
`<Public Key>`
`<Signature>`
	`<Public Key Hash>`
	`OP_EQUALVERIFY`
	`OP_CHECKSIG`
		The top stack item hashed with SHA256 and RIPEMD160.

`<Public Key Hash>`
`<Public Key Hash>`
`<Public Key>`
`<Signature>`
	`OP_EQUALVERIFY`
	`OP_CHECKSIG`
		`<Public Key Hash>` pushed onto the stack.

`<Public Key>` `<Signature>`	`OP_CHECKSIG`	Equality of the two top stack items is checked. The script fails if not equal.
`TRUE`		The transaction is hashed and with the two top stack items as a public key and a signature, the signature is verified.

Table 1.5: P2PKH `scriptPubKey` Execution

To unlock the output, an unlocking script must provide the signatures satisfying the redeem script conditions, together with the serialized redeem script:

```
<Signatures>... <Redeem Script>
```

The validation is performed in two steps. First, the redeem script is run together with the locking script to check if the hash matches. If the hash matches, then the unlocking script is executed on its own.

Effectively, P2SH transactions move the responsibility of providing the spending conditions from the sender to the recipient. One common use of P2SH is for *multisignature* transactions, where M of N signatures are required to spend the output (an implementation of *Shamir's secret sharing scheme* [3]). With P2SH, instead of providing all N public keys to the sender, the recipient creates a multi-signature script on his own and provides only its hash in form of a P2SH address. This simplifies the process of composing the transaction and reduces a transaction fee for the sender because the hash is much shorter than the original script.

1.5 Segregated Witness

Segregated Witness, abbreviated as *SegWit*, is a Bitcoin protocol upgrade that was activated in 2017 as a *softfork* (a backward-compatible change) and comes with several improvements.

It adds a new structure called a *witness*, which contains unlocking scripts previously present in transaction inputs. The witness is stored in blocks separately from the transaction structure and it is not counted towards the 1 MB block size limit. That results in a larger effective block size and lower transaction fees.

However, the main motivation behind SegWit was to fix the *transaction malleability* [13], a design flaw that allowed an attacker to change the hash of an unconfirmed transaction by altering the unlocking script. Even though the transaction is signed, the signature does not contain all data, specifically the unlocking script, which contains the signature itself. However, the unlocking script is included in the serialized transaction that is used to compute the transaction hash. Therefore, it's possible to tweak the unlocking script in a way that the transaction is still valid, but has a different hash. SegWit fixes that by segregating the unlocking script (witness) into a separate structure not included in the transaction hash.

There is a new signature algorithm for SegWit scripts [14]. Previously, the signature did not involve an amount spent by the input. This was a problem especially for hardware wallets because they had to request and hash all transactions referenced by inputs to verify the amount spent and to reliably

calculate the transaction fee. Now, as the amount is included in the signature, a hardware wallet can use the amount value from an untrusted source. In case an invalid value is provided, the signature is invalid and no funds are lost. This makes the signature process faster and easier to implement.

Another benefit is an introduction of script versioning. Every SegWit script begins with a version number, so it will be easier to introduce new operators in future. A locking script that consists of a number followed by 20 or 32 bytes of data has a new meaning. The value of the first number is called the *version byte*, the data following are called the *witness program*. [15]

1.5.1 Pay to Witness Public Key Hash (P2WPKH)

If the version byte is 0 and the witness program is 20 bytes, it is interpreted as a **Pay to Witness Public Key Hash** (P2WPKH) program:

```
0 <Public Key Hash>
```

The witness must consist of a signature and a public key:

```
<Signature> <Public Key>
```

The HASH160 of the public key must match the 20-byte witness program. The signature is then verified using the public key.

1.5.2 Pay to Witness Script Hash (P2WSH)

If the version byte is 0 and the witness program is 32 bytes, it is interpreted as a **Pay to Witness Script Hash** (P2WSH) program:

```
0 <Script Hash>
```

The witness must consist of an input stack satisfying the script conditions, followed by the serialized script (the *witness script*):

```
<Signatures>... <Witness Script>
```

SHA256 of the witness script must match the 32-byte witness program. Then the script is deserialized and executed with the remaining witness stack. It must result in exactly a single `TRUE` on the stack.

1.5.3 Pay to Witness Public Key Hash Nested in P2SH (P2SH-P2WPKH)

For P2WPKH and P2WSH payments, both sender's and recipient's wallet must have support for SegWit. The sender's wallet has to be able to create SegWit type outputs, while the recipient's wallet must be able to spend these outputs by constructing a SegWit transaction. Because Segregated Witness is a backward-compatible upgrade, there will be a period where both non-upgraded and upgraded clients exist. That brings some compatibility issues we have to solve.

First, there needs to be a way for a legacy wallet to send a payment to a wallet with SegWit support. The recipient's wallet can construct a P2SH address that embeds the witness script inside it. For a sender, it appears as a regular P2SH address, however, the recipient is able to spend such an output in a SegWit transaction. The type of P2SH script that embeds P2WPKH or P2WSH script is noted as P2SH-P2WPKH or P2SH-P2WSH. [15]

We construct the P2SH-P2WPKH address as follows. First, we construct a witness program consisting of the version number and a 20-byte public key hash. Then, we hash the witness program with HASH160, producing a 20-byte hash. Finally, the script hash is converted to a P2SH address. The implementation in Kotlin is provided in Source Code 2.

```
fun getSegwitAddress(publicKey: ECPoint): String {
    val publicKeyEncoded = publicKey.getEncoded(true)
    val publicKeyHash = hash160(publicKeyEncoded)
    val scriptSig = ByteArray(publicKeyHash.size + 2)
    scriptSig[0] = 0x00 // version 0
    scriptSig[1] = 0x14 // push 20 bytes
    System.arraycopy(publicKeyHash, 0, scriptSig, 2,
        publicKeyHash.size)
    val scriptSigHash = hash160(scriptSig)
    return encodeBase58Check(scriptSigHash, 5)
}
```

Source Code 2: Construction of a P2SH-P2WPKH address from a public key

1.5.4 Derivation Scheme for P2SH-P2WPKH Based Accounts

A new key derivation scheme is introduced for generating SegWit addresses in HD wallets, to distinguish between SegWit and non-SegWit accounts. BIP-0049 [16] specifies the purpose field in a derivation path to be 49' for SegWit accounts. The rest of the path is the same as for BIP-0044 [10] accounts, which are now called *legacy*. Having a different derivation path for SegWit

accounts instead of just using the same path with different address encoding prevents issues when an account with SegWit transactions would be imported into a wallet not yet supporting SegWit.

1.5.5 Native SegWit address

Once more of the wallets become compatible with SegWit, it will be better to use witness scripts directly instead of embedding them in P2SH. However, we still need a way to make sure that both sender's and recipient's wallet support SegWit. For this reason, a new address format has been proposed in BIP-0173 [17]. As opposed to the Base58Check encoding used in the original address format, the new format uses a *Bech32* encoding, which is Base32 with a BCH[4] code checksum. It is more efficient to encode in a QR code because an *alphanumeric mode* can be used due to the fact it does not contain mixed letter case. The BCH code is error-correcting, which means it can not only detect introduced errors but also correct them. Even though a wrong address should not be corrected automatically, it is useful for highlighting an incorrectly entered part of the address in the UI.

1.6 TREZOR Hardware Wallet

TREZOR[5] is a device developed and produced by **SatoshiLabs**, a Czech company headquartered in Prague. The first model, TREZOR One, was introduced in 2013 as the world's first hardware wallet. While its original purpose was to serve as a hardware Bitcoin wallet, over the years, with firmware updates it evolved into a more generic cryptographic device. Today, it supports several cryptocurrencies and can be used for other purposes as well, e.g. for two-factor authentication or to encrypt passwords in a password manager.

In 2018, the next-generation TREZOR Model T was released, featuring a touchscreen, USB-C port, microSD card slot and a new firmware based on MicroPython[6].

1.6.1 TREZOR API

TREZOR communicates using a synchronous request–response protocol. In practice, the computer sends a request message and waits for the response from TREZOR. The response can be a success, a failure or a message containing the requested data. Furthermore, the response can also be a request for entering a PIN, passphrase or pressing a button. In that case, the computer should send an acknowledgment message with the requested information and wait for the next response. [18]

[4] Bose–Chaudhuri–Hocquenghem
[5] https://trezor.io/
[6] https://micropython.org/

The messages are serialized using *Protocol Buffers*[7] and exchanged over a USB HID[8] interface. Since the introduction of a USB host support in Android 3.1, TREZOR can be connected to any supported mobile device or tablet using a **USB On-The-Go** (OTG) cable.

1.6.2 Pin Matrix

Most of the requests require the user to enter a PIN to unlock the device. When a PIN is needed, TREZOR responds with a `PinMatrixRequest` message and shows a *PIN matrix* (a 3×3 grid with digits arranged at random) on its display. The computer should present the user with an empty 3×3 grid. The user then looks at the matrix on TREZOR and presses the corresponding matrix items on the computer. This prevents keyloggers from capturing the PIN, as the matrix arrangement is different each time.

After the PIN is entered, the computer sends a `PinMatrixAck` message with the positions of the pressed matrix elements.

1.6.3 Passphrase

A *passphrase* serves as an optional additional security level. The passphrase can be any string up to 50 characters long. It is never stored in TREZOR. Instead, it acts as a part of the root seed used to generate the keys. That means there is no wrong passphrase, but every passphrase gives access to a different wallet.

When enabled, TREZOR asks for a passphrase after entering the PIN by sending a `PassphraseRequest` message. After the user enters the passphrase, the computer sends it in a `PassphraseAck` message.

1.6.4 Button

TREZOR can require the user to press the button to confirm a sensitive operation. In that case, it will reply with a `ButtonRequest` message. The computer should immediately send a `ButtonAck` message to acknowledge the request and display instructions in the UI. Once the user presses the button, TREZOR continues by sending the response to the previous request.

1.6.5 Get Public Key

A `GetPublicKey` message can be used to export a public key from TREZOR by specifying its derivation path in the `address_n` field. The device responds with a `PublicKey` message containing the requested public key node. It does not require any user interaction if TREZOR has been previously unlocked.

[7] https://developers.google.com/protocol-buffers/

[8] Human Interface Device

1.6.6 Sign Transaction

A general transaction signature workflow is shown in Figure 1.4. Because the transaction size can be larger than the memory of the device itself, it has to be split into parts and those parts are being sent separately as they are being requested by TREZOR.

The signing workflow starts by sending a `SignTx` message specifying the transaction metadata in `tx.version`, `tx.lock_time`, `tx.inputs_cnt` and `tx.outputs_cnt` fields. If TREZOR has not been previously unlocked, it requests a PIN and a passphrase as described in previous sections.

Then the whole flow is managed by TREZOR. It keeps sending `TxRequest` messages, requesting parts of the transaction to be signed. It can also request information about a transaction referenced by a non-SegWit input to calculate the input amount. The requested information should be provided by a computer in a `TxAck` message.

There are several request types that can be specified by the `request_type` field of the `TxRequest` message:

- `TXMETA` requests the metadata of a referenced transaction specified by its TXID in the `details.tx_hash` field. A computer should respond with `TxAck` containing `tx.version`, `tx.lock_time`, `tx.inputs_cnt` and `tx.outputs_cnt`.

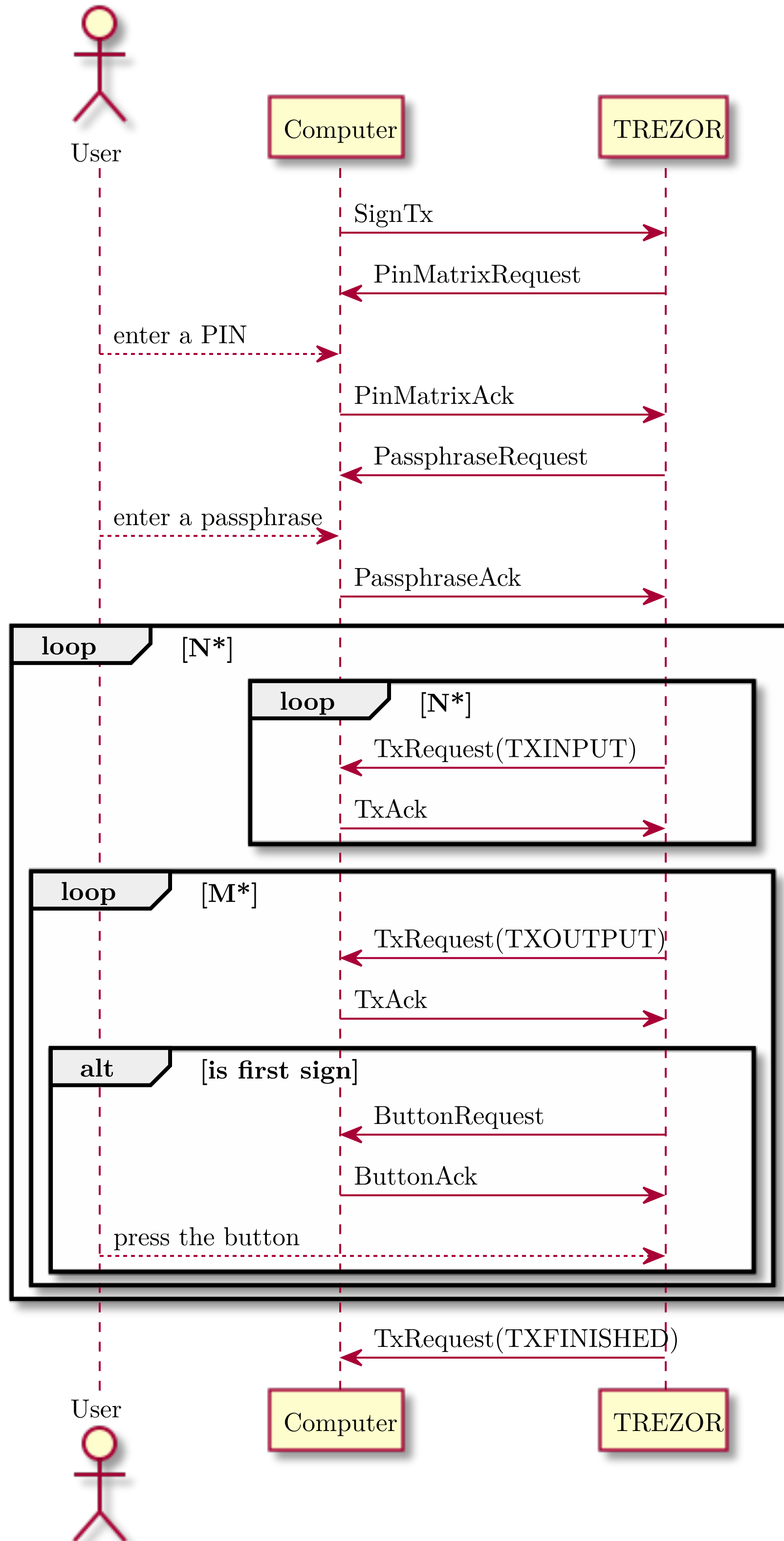
- `TXINPUT` requests the information about a transaction input with an index of `details.request_index`. The reply should fill `tx.inputs[0]` with `prev_hash`, `prev_index` and `sequence`. For a referenced transaction (when `details.tx_hash` is set), `script_sig` must be set. Otherwise, the path to the private key should be specified in `address_n` and `script_type` should describe the type of the corresponding output locking script. For SegWit inputs, also the `amount` field must be filled, matching the amount of the input transaction in satoshis.

- `TXOUTPUT` requests the information about a transaction output with an index of `details.request_index`. For a referenced transaction (when `details.tx_hash` is set), `tx.bin_outputs[0]` must be filled. Otherwise, `tx.outputs[0]` must be filled with the output data. For a change output, the address should be specified by its path in `address_n` and `script_type` should describe the locking script type. Otherwise, the `address` should be filled and `script_type` set to `PAYTOADDRESS`.

- `TXFINISHED` notes a message containing the last chunk of the signed transaction. A computer should not respond with `TxAck` to this message.

The `TxRequest` messages can contain chunks of the signed transaction in the `serialized.serialized_tx` field. Those chunks have to be concatenated to form the serialized signed transaction.

Figure 1.4: Signature workflow for TX with N inputs and M outputs



Chapter 2

State of the art

2.1 Existing Wallets with TREZOR Support

In this chapter, we will list currently existing Bitcoin wallets supporting TREZOR in any way, explore their features and analyze the weak points that could be improved on.

2.1.1 TREZOR Wallet (Web)

TREZOR Wallet[9] is the official wallet for TREZOR developed by SatoshiLabs. It supports not only Bitcoin but also some of its forks and several other cryptocurrencies. It provides a user-friendly interface and it's probably the most complete wallet for TREZOR supporting all its features, including:

- multiple legacy and SegWit accounts
- sending to P2PKH, P2SH and Bech32 addresses
- advanced sending options (multiple recipients, lock time...)
- account, transaction and address labels synchronized with Dropbox[10]

It is a web-based app supporting Google Chrome and Firefox browsers. It can use either **TREZOR Bridge**[11] or the **WebUSB**[12] standard to handle the communication between the web browser and the TREZOR device.

Upon the first visit, the user has to connect TREZOR over the USB interface to load the public key and initialize the wallet. When the device is disconnected, the user can either choose to forget it, or to save the public key and continue using the wallet as *watch-only*. In a watch-only mode, the user can see transactions history, but cannot send new transactions.

[9] https://wallet.trezor.io/
[10] https://www.dropbox.com/
[11] https://wallet.trezor.io/#/bridge
[12] https://wicg.github.io/webusb/

2.1.2 Electrum (Linux, Windows, macOS)

One of the most widely used Bitcoin wallets is **Electrum**[13], a multi-platform wallet available for Linux, Windows and macOS. It can be used on its own or in combination with various hardware wallets, including TREZOR. The setup is a little bit complicated for an average user because it cannot perform account discovery and each account has to be added manually by defining the derivation path. It has SegWit support since version 3.

2.1.3 Mycelium (Android)

Mycelium[14] is a Bitcoin wallet for Android supporting multiple hardware wallets. After connecting TREZOR, it loads the accounts list and allows to import them one by one. At the moment, it is the only wallet for Android that allows signing transactions with TREZOR. However, it only supports legacy accounts and does not have support for SegWit. The user can label accounts and transactions, but the labels are only stored locally and cannot be synchronized in any way.

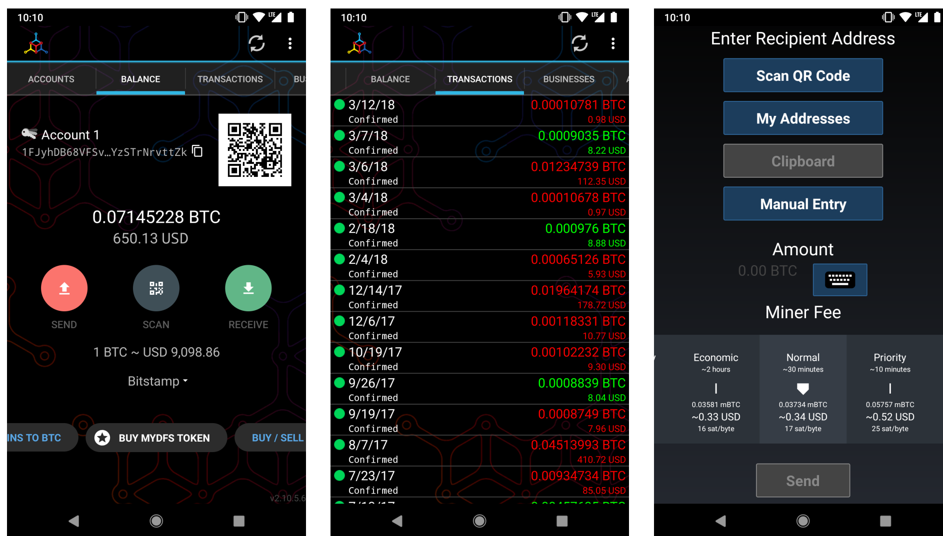


Figure 2.1: Mycelium

2.1.4 Sentinel (Android)

Sentinel[15] is a watch-only wallet for Android. An account is imported by entering an extended public key (xpub). To get an xpub for a TREZOR account,

[13]https://electrum.org/
[14]https://wallet.mycelium.com/
[15]https://samouraiwallet.com/sentinel.html

the user can scan a QR code available in the web TREZOR Wallet. Sentinel then loads transactions history and alerts the user whenever it detects a new transaction on any of its addresses. It also allows generating a new address for receving a payment. Both legacy and SegWit accounts are supported.

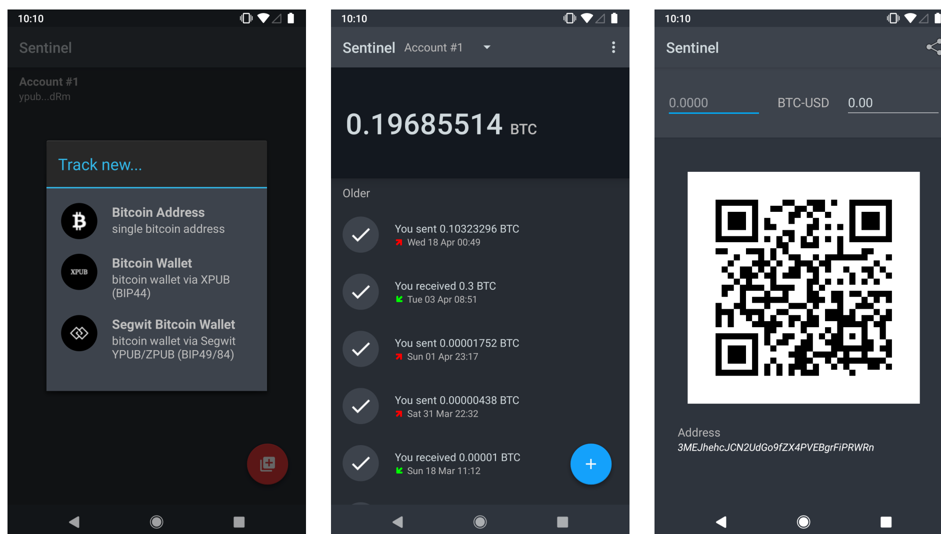


Figure 2.2: Sentinel

2.1.5 TREZOR Manager (Android)

TREZOR Manager[16] is an Android app by SatoshiLabs allowing to initialize a new TREZOR device, update the firmware, change the PIN, set up a passphrase, and customize some other things. However, it does not serve as a wallet and does not have any Bitcoin-specific features.

2.2 TREZOR Communication Library for Android

There is a **trezor-lib**[16] library developed by SatoshiLabs that facilitates communication between TREZOR and an Android device. However, the library handles the communication only to the extent of establishing the connection and exchanging messages. The API workflow and the UI for entering the PIN and a passphrase must still be implemented by the app developer.

Both Mycelium and TREZOR Manager are using this library to communicate with TREZOR. We will use it as a basis for building a higher-level library in the next chapter.

[16]https://github.com/trezor/trezor-android

Chapter 3

Realisation

We will now take advantage of the principles introduced in the first chapter and build a Bitcoin wallet as an application for Android OS. Because we are concerned with security, the app should not store any private keys on a mobile device, as its storage is potentially insecure and we would risk keys compromise. Instead, the app should only store public keys and utilize the TREZOR device as a secure private key storage.

For a long time, Java was the only supported language for writing native Android apps. At Google I/O 2017, Google announced official support for the **Kotlin**[17] language on Android. Kotlin is a statically typed language developed by JetBrains. It provides a concise syntax, a null safety, higher-order functions to allow functional programming style and it is interoperable with Java. It's expected to overtake Java as the most used language for Android development in 2018. [19] For these reasons, Kotlin has been chosen as the language for implementing our Android app.

Android uses the **Bouncy Castle**[18] library to implement some cryptographic functions in the `java.security` package. However, it uses an old, stripped-down version of the library and the implementation can differ across different Android versions. Moreover, there is a 128-bit key length limit set in Java Cryptography Extension, but we will need a 256-bit key for metadata encryption in Section 3.5. Thus, we bundle the latest version of Bouncy Castle with the app. The library provides a collection of cryptography APIs including RIPEMD160, SHA256 hash functions, an elliptic curve cryptography and AES encryption.

The app is based on **Android Architecture Components**[19], a collection of libraries by Google for implementing an MVVM (*Model-View-ViewModel*) architecture. It helps to separate a UI and application logic and manage the app lifecycle. The **Room** library is used to persist data in an SQLite database.

[17]https://kotlinlang.org
[18]https://www.bouncycastle.org
[19]https://developer.android.com/topic/libraries/architecture

3.1 TREZOR Intents Library

We start by designing an Android library facilitating communication between an Android app and TREZOR. The library should make integration with TREZOR effortless, by abstracting away the whole communication workflow. It provides the UI for entering a PIN, a passphrase and shows instructions when waiting for the user to press the button.

The library is based on **Intents**, a part of **Android SDK** used for communication between app components, or even different applications. In our context, we use `Intent` to represent a request to launch an activity managed by the library. After performing the desired action, the launched activity responds with a result `Intent`, sending the data back to the original activity.

3.1.1 Generic Request

From the application point of view, starting an `Intent` is as simple as:

```
val request = GenericRequest(
        TrezorMessage.Initialize.getDefaultInstance())
val intent = TrezorActivity.createIntent(this, request)
startActivityForResult(intent, RC_INIT)
```

It says we want to send an `Initialize` message, without specifying any fields. `RC_INIT` can be an arbitrary integer constant representing a *request code*, which is later used to match the request with a result.

Once `TrezorActivity` sends a request to TREZOR and receives a response, it delivers the result back to the calling activity, which receives an `onActivityResult` callback:

```
override fun onActivityResult(requestCode: Int, resultCode: Int,
        data: Intent?) {
    if (requestCode == RC_INIT && resultCode == RESULT_OK) {
        val message = TrezorActivity.getMessage(data) as
                TrezorMessage.Features
    }
}
```

First, we check whether the result belongs to our request by comparing the `requestCode` with a code used to start the activity. Then, we check if the result is successful by comparing the `resultCode` with an `Activity.RESULT_OK` constant. Finally, we use a `TrezorActivity.getMessage` method to extract the response message from the `Intent`.

These snippets can be used to obtain a response to almost any request message supported by TREZOR. [20] The common messages `PinMatrixRequest`, `PassphraseRequest` and `ButtonRequest` are handled by the library and are never returned as a result.

3.1.2 Transaction Signature

The library implements the transaction signature workflow described in Section 1.6.1.5. All the app has to do is to provide a transaction to sign, and a map of TXIDs to transactions referenced by non-SegWit inputs. The reason why referenced transactions are needed is that TREZOR has to verify the input amount. For SegWit inputs, this is not needed because the signature already includes the amount, as previously discussed in Section 1.5.

An `Intent` for signing a transaction is started in the following way:

```
val inputs = mutableListOf<TrezorType.TxInputType>()
val outputs = mutableListOf<TrezorType.TxOutputType>()

inputs += TrezorType.TxInputType.newBuilder()
    .addAllAddressN(address)
    .setPrevHash(prevHash)
    .setPrevIndex(prevIndex)
    .setScriptType(scriptType)
    .build()

outputs += TrezorType.TxOutputBinType.newBuilder()
    .setAmount(amount)
    .setScriptPubkey(scriptPubKey)
    .build()

val tx = TrezorType.TransactionType.newBuilder()
    .addAllInputs(inputs)
    .addAllOutputs(outputs)
    .setInputsCnt(inputs.size)
    .setOutputsCnt(outputs.size)
    .build()

val referencedTxs = mutableMapOf<String,
        TrezorType.TransactionType>()
...

val request = SignTxRequest(tx, referencedTxs)
val intent = TrezorActivity.createIntent(this, request)
startActivityForResult(intent, RC_SIGN_TX)
```

In `onActivityResult`, the signed transaction serialized as a hexadecimal string can be obtained in the following way:

```
val signedTx = TrezorActivity.getSignedTx(data)
```

3.2 Account Discovery

On the first launch, the app asks the user to connect TREZOR to the mobile device over a USB cable. Then, it performs an *account discovery* [10], a process of searching for accounts having any transaction history. The algorithm works as follows:

1. Get a public key for the first account by sending a `GetPublicKey` request to TREZOR.
2. Derive a public key for the external chain.
3. Successively derive addresses on the external chain and scan them for transactions. Stop scanning once 20 (address gap limit) unused addresses in a row are found.
4. If no transactions are found, stop discovery. If there are some transactions, increment the account index and go to step 1.

The app then saves all discovered accounts, derived addresses, fetched transactions and shows them in the UI.

3.3 Bitcoin Network Connection

To fetch transactions for a Bitcoin address, we need a way to connect to the Bitcoin network. In general, there are three ways how apps can access the blockchain:

- A **full node** fetches and stores the whole blockchain. This is the most secure solution, but infeasible for a mobile app due to storage constraints. The Bitcoin blockchain takes over 160 GB as of April 2018.
- An **SPV** (Simplified Payment Verification) node downloads the block headers and then requests only transactions it is interested in by using *Bloom filters* [5]. While it cannot validate transactions, it can prove that the received transaction has been included in a block thanks to the *Merkle tree* structure.
- A custom backend service that allows requesting information from the blockchain over the REST API. A client using such a service cannot perform any validation on its own, so it's important it's connecting to a trusted backend.

We will use **Insight API**[20], a Bitcoin blockchain REST and WebSocket API developed by Bitpay. It is easy for implementation and provides considerable security when connected to a trusted backend.

[20] https://github.com/bitpay/insight-api

By default, the app is connecting to the backend run by SatoshiLabs at the address `https://btc-bitcore1.trezor.io`. For users especially concerned with privacy who are willing to run their own backend service, there is an option to set a custom backend address.

The app is using the following Insight API methods:

3.3.1 Transactions for Multiple Addresses

```
GET /addrs/[:addrs]/txs[?from=&to=]
```

Returns a list of transactions related to any of the specified addresses. Multiple addresses can be provided, separated by a comma. Parameters `from` and `to` are used for pagination.

3.3.2 Estimate Fee

```
GET /utils/estimatefee[?nbBlocks=]
```

Returns a recommended fee rate that should result in a transaction being mined within the next `nbBlock` blocks. Multiple block numbers can be provided, seperated by a comma.

3.3.3 Transaction Broadcasting

```
POST /tx/send
```

Broadcasts a signed transaction to the network. The request should send a serialized transaction encoded as a hexadecimal string in the `rawtx` parameter.

3.4 Transaction Composition

The process of composing a transaction consists of several steps. First, the user enters a destination address. That can be done by typing it, pasting the previously copied text or, most frequently, scanning a QR code. For QR code scanning, we are using a scan `Intent` of the **Barcode Scanner**[21] app. In case the app is not installed, its listing on Google Play is opened.

The QR code usually encodes a Bitcoin URI scheme [21], which can contain not only the address, but optionally also a requested `amount` and a `message` describing the transaction:

```
bitcoin:3DvYRiEmqdQW1i4a27Pjm6vnsx2fKSFS1F&amount=100
```

[21] https://play.google.com/store/apps/details?id=com.google.zxing.client.android

Then, the user enters the desired transaction amount, if it has not been pre-filled from the QR code yet. The amount can be set either in BTC, or in USD and automatically converted to BTC according to the current exchange rate. We are using **CoinMarketCap API**[22] to get the recent exchange rate, which is calculated by taking the volume average of prices at several Bitcoin exchanges.

3.4.1 Fee Estimation

The next step is specifying a transaction fee rate. All advanced wallets should use some form of a dynamic fee that is calculated from the current network congestion. The fee rate determines the time in which the transaction will be included in a block.

In our wallet, we provide 4 recommended fee levels: high (included within the next 2 blocks $\sim$ 20 minutes), normal (6 blocks), economy (25 blocks) and low (50 blocks). The actual fee rate is fetched from the Insight API. Its calculation is based on the fees of transactions included in the last few blocks and the fees of unconfirmed transactions. [22] There is also an option to set a custom fee rate.

3.4.2 Coin Selection

Then, the task of the wallet is to select UTXOs with a total value equal or greater than the *target* (the amount to be spent). There are a few things to consider during the process.

Firstly, we need to select enough coins to meet the target. Secondly, we want to minimize transaction fees by selecting as few inputs as possible because each input increases the transaction size. Lastly, we want to reduce the UTXO set by creating less outputs than inputs because all full nodes have to keep the UTXO set in memory to validate transactions and growth of the UTXO set increases hardware requirements. This is an *optimization problem* with partly opposed goals.

3.4.3 Subset Sum Problem

The optimal solution would be to find UTXOs with a total value closely matching the desired amount, so we don't have to generate a change output, as the change output increases the size of the transaction (and consequently the fee) and further fragments the UTXO set.

This is an instance of a *subset sum problem*, whose goal is to find a subset of integers that sum up to a given target. However, the problem is *NP-complete*, meaning there is no known algorithm that could solve it in a polynomial-time.

[22] https://coinmarketcap.com/api/

3.4.4 Coin Selection Strategies

There are several prevailing strategies that can be used for UTXO selection:

- **FIFO** (First In, First Out) algorithm accumulates UTXOs from the oldest, until the target is reached or exceeded. This simple approach produces stable results and it is used by TREZOR Wallet and Electrum.

- **FIFO with pruning** extends the FIFO algorithm by adding a final pass in which any small inputs not needed to fund the transaction are removed from the candidate set. While it can benefit from lower transaction fees, it also leads to a UTXO set bloated with small outputs. It is used by Mycelium.

- **Branch and Bound** [23] tries to find the exact match, so a change output is not needed. First, UTXOs are sorted by their value in descending order. A binary tree is constructed, where each level denotes an inclusion or omission of a UTXO. The tree is explored in a depth-first search by randomly selecting nodes for expansion. Paths with a sum exceeding the target are cut and not explored further. The search stops when an exact match is found. As the tree grows exponentially with a number of UTXOs, it is also useful to set a limit of tested combinations. If no match is found, we fall back to a traditional algorithm. This strategy has been recently implemented in Bitcoin Core.

In Source Code 3 we implement a simple FIFO coin selection strategy, as it has been shown to produce stable results for different use cases and it is also being used in the TREZOR Wallet web app. In future, it could be improved on by first trying to find the exact match using Branch and Bound, with a fallback to FIFO.

3.4.5 Dust

A transaction output is labeled as *dust* when the cost of spending it would be similar to its value. Bitcoin Core (the reference Bitcoin client) defines dust as an output whose fees would exceed 1/3 of its value. When considering a minimum fee of 1 sat/B, a typical P2PKH input of 148 bytes and P2PKH input of 34 bytes, we get $3 * (148 + 34) = 546$ as a threshold for dust outputs. If a change output should be less than that, we do not create it and instead leave a higher transaction fee.

```
fun select(utxoSet: List<TransactionOutput>,
        outputs: List<TrezorType.TxOutputType>,
        feeRate: Int, segwit: Boolean):
        Pair<List<TransactionOutput>, Int> {

    val target = outputs.sumBy { it.amount.toInt() }

    val inputs = mutableListOf<TransactionOutput>()
    var inputsValue = 0L

    var fee = 0

    for (utxo in utxoSet) {
        // We have enough funds already
        if (inputsValue >= target + fee) {
            break
        }

        // Add UTXO to inputs
        inputs += utxo
        inputsValue += utxo.value

        // Update the transaction fee
        fee = calculateFee(inputs.size, outputs, feeRate, segwit)

        // If a change output is needed, increase the fee
        if (inputsValue - target - fee > DUST_THRESHOLD) {
            fee += changeOutputBytes(segwit) * feeRate
        }
    }

    // Not enough funds selected
    if (inputsValue < target + fee) {
        throw InsufficientFundsException()
    }

    return Pair(inputs, fee)
}
```

Source Code 3: FIFO Coin Selector

3.5 Bitcoin Metadata

To keep track of your spending habits, it can be useful to label the transactions and addresses, so you can easily identify the transaction subject and participants later on. Most wallets have some sort of labeling, but they store the labels either only locally, or allow to export them in a proprietary format. To make various wallets compatible, there is a need for a common standard. SatoshiLabs proposed a new format for Bitcoin metadata and its encryption in HD wallets in the SLIP-0015 standard [24].

3.5.1 Data Format

Each account has its own metadata file with data stored in the JSON[23] format. The JSON object has the following fields:

- `version`: `string` – the metadata format version, currently `1.0.0`
- `accountLabel`: `string` – the account label
- `addressLabels`: `object` – a map of addresses to labels
- `outputLabels`: `object` – a map of TXIDs to outputs labels

An example metadata JSON object is shown in Source Code 4.

```
{
  "version": "1.0.0",
  "accountLabel": "Savings Account",
  "addressLabels": {
    "3DvYRiEmqdQW1i4a27Pjm6vnsx2fKSFS1F": "My Donation Address"
  },
  "outputLabels": {
    "2b13030bd5f79ffb44d09b2d4e9540e1029c36c25c02fb8403dc...": {
      "0": "Money to Alice",
      "1": "Money to Bob"
    }
  }
}
```

Source Code 4: An example metadata JSON object

The files are further encrypted with a *password* derived from a private key located in the deterministic hierarchy. We are not encrypting with the private key directly, so labeling can be used with a hardware wallet effortlessly because the file encryption and decryption happens in the computer. The

[23] JavaScript Object Notation

user only needs to connect the hardware wallet once to generate a *master key*. We then use the master key to derive a *filename* and a *password* for each account. The metadata should be encrypted with the password used as a symmetric encryption key and stored in a file with the specific filename. The encrypted files can be eventually backed up to any untrusted cloud storage service, without its provider being able to read them.

3.5.2 Master Key

First, we derive the master key from a private key in the deterministic hierarchy. When using TREZOR, we send a `CipherKeyValue` message defined in SLIP-0011 [25]. Simply said, it encrypts a provided value with a private key at the specified path in the deterministic hierarchy. Specifically, to get the master key, we have to encrypt the value of a constant specified by the standard using the key at the `m/10015'/0'` path. The resulting master key is a sequence of 32 bytes.

3.5.3 Account Key

We derive an *account key* for every account using the HMAC-SHA256 function. We take the master key as the secret key and the account extended public key (xpub) as the message. The result is converted to a string using the Base58 encoding:

$$\mathit{account\ key} = \text{Base58}(\text{HMAC-SHA256}(\mathit{master\ key},\ \mathit{xpub}))$$

3.5.4 Filename and Password

We use the account key to derive the filename and password.

First, we use the HMAC-SHA512 function with the account key as the secret key and the constant of `0123456789abcdeffedcba9876543210` as the message. We get a 64-byte result.

The left 32 bytes are converted to a hexadecimal string, which is used as the filename with the ".mtdt" extension. The right 32 bytes are representing the password used for symmetric encryption.

3.5.5 Encryption

The AES-256-GCM algorithm is used for encryption. It's the AES *block cipher* with a 256-bit key, using *Galois/Counter Mode* (GCM) as a mode of operation. GCM combines the counter mode of encryption with the Galois mode of authentication. [26] This form of encryption is called an *authenticated encryption* because it provides not only confidentiality but also authenticity of the data. It ensures the data have not been modified by any third party during transfer over an insecure connection.

The authenticated encryption function takes a key, an *initialization vector* and a *plaintext* on input. It produces a *ciphertext* and an *authentication tag* on output. The authenticated decryption function takes a key, an initialization vector, a ciphertext, an authentication tag and produces the plaintext or an indication of inauthenticity.

The resulting metadata file consists of a random 12-byte initialization vector used for encryption, a 16-byte authentication tag and a ciphertext.

3.5.6 Dropbox Synchronization

We are using Dropbox cloud storage for metadata synchronization, to be compatible with the web TREZOR Wallet. In this way, the user can add labels via the web wallet and see them in our Android wallet after synchronization, and vice versa.

When the user wants to enable labeling, the app requests the user to login with their Dropbox account and authorize the app. Then the app saves a received *OAuth token* and uses it for synchronizing the metadata files using **Dropbox Core SDK**[24]. The metadata are stored in the `/Apps/TREZOR Wallet` folder in the previously described format.

[24] https://github.com/dropbox/dropbox-sdk-java

Conclusion

The goal of this thesis was to design a library simplifying communication between an Android app and the TREZOR device, and subsequently implement a Bitcoin wallet with TREZOR support to demonstrate the usage of the library.

We used a low-level TREZOR Communication Library as a basis to build a higher-level TREZOR Intents Library. With TREZOR Intents, an app developer provides a message they want to send to TREZOR. The library initiates the connection, sends the message and handles common requests from the device such as a PIN or passphrase request, and returns the final response received from TREZOR. The library also implements a transaction signature workflow, which is a bit more complicated and consists of sending several messages and concatenating the signed transaction from the received messages. A developer just needs to specify the transaction to sign and receives the signed transaction.

Next, we designed and implemented a Bitcoin wallet app. Upon the first start, the user connects TREZOR, the app performs an account discovery, saves public keys for all discovered accounts and presents the user with the accounts list. It has support for both legacy and SegWit accounts. A list of user's receiving addresses is generated and all transactions received or sent from those addresses are fetched using Insight API. The app calculates the account balance using the fetched transactions.

The app serves not only as a watch-only wallet, but it also allows to compose a new transaction. It allows the user to scan a Bitcoin address from the QR code, enter an amount in BTC or USD and choose from a list of recommended transaction fee rates. Finally, the transaction is signed with TREZOR and broadcast to the Bitcoin network.

Finally, the user can add labels to accounts, addresses and transaction outputs. Those metadata are synchronized in a standardized encrypted form with Dropbox cloud storage. Therefore, when a label is added via the web TREZOR Wallet, it is also visible in the Android app after synchronization.

The screenshots of the developed app are attached in Appendix A. The source code along with the compiled APK file can be found on an enclosed SD card or at `https//github.com/MattSkala/trezor-wallet`.

Future Work

While the app already provides all basic features of an Android wallet, there is still room for improvement to provide even better user experience.

In future, Insight WebSocket API could be used to receive real-time notifications about changes in the network. This would enable to show any incoming transaction instantly, without need to manually synchronize via REST API. The same principle could be applied to metadata synchronization, by using Dropbox API to monitor file changes in a Dropbox folder.

A QR code scanner could be integrated directly into the app, to get rid of the dependency on the 3rd party Barcode Scanner app.

The wallet could also support more advanced sending features, such as an option to add multiple recipients to a single transaction, to specify a transaction lock time or to add OP_RETURN output type allowing to store arbitrary data in the blockchain in form of an unspendable output.

The next step should be to improve support for the new format of native SegWit addresses using Bech32 encoding. While it's already possible to send transactions to Bech32 addresses, such transactions do not currently show the target address in the UI because Insight API does not support the new address format yet. Either the address could be extracted from the locking script on the app side, or Insight API could be extended with Bech32 address support.

Currently, the app supports legacy (P2PKH) and compatibility SegWit (P2SH-P2WPKH) accounts. In future, there will probably be need to also add support for native SegWit (P2WPKH) accounts as defined in BIP-0084 [27], that use the Bech32 format for receiving addresses. However, this is not supported even in the web TREZOR Wallet yet.

Bibliography

[1] Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. 2009, [Accessed: 5 December 2017]. Available from: http://bitcoin.org/bitcoin.pdf

[2] Maxwell, G. Confidential Transactions. [Accessed: 27 February 2018]. Available from: https://people.xiph.org/~greg/confidential_values.txt

[3] Hoffstein, J.; Pipher, J.; et al. *An Introduction to Mathematical Cryptography*. Springer, 2008, ISBN 978-0-387-77994-2.

[4] Standards for Efficient Cryptography Group. SEC2: Recommended Elliptic Curve Domain Parameters. [Accessed: 12 March 2018]. Available from: http://www.secg.org/sec2-v2.pdf

[5] Antonopoulos, A. M. *Mastering Bitcoin: Programming the Open Blockchain*. O'Reilly Media, second edition, 2017, ISBN 978-1491954386.

[6] Klyubin, A. Some SecureRandom Thoughts. [Accessed: 10 May 2018], 2013. Available from: https://android-developers.googleblog.com/2013/08/some-securerandom-thoughts.html

[7] Wuille, P. Hierarchical Deterministic Wallets. [Accessed: 2 April 2018], 2012. Available from: https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki

[8] Krawczyk, H.; Bellare, M.; et al. HMAC: Keyed-Hashing for Message Authentication. [Accessed: 12 May 2018], 1997. Available from: https://tools.ietf.org/html/rfc2104

[9] Palatinus, M.; Rusnak, P. Purpose Field for Deterministic Wallets. [Accessed: 2 April 2018], 2014. Available from: https://github.com/bitcoin/bips/blob/master/bip-0043.mediawiki

[10] Palatinus, M.; Rusnak, P. Multi-Account Hierarchy for Deterministic Wallets. [Accessed: 2 April 2018], 2014. Available from: https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki

[11] Rusnak, P.; Palatinus, M. Registered coin types for BIP-0044. [Accessed: 2 April 2018], 2014. Available from: https://github.com/satoshilabs/slips/blob/master/slip-0044.md

[12] Andresen, G. Pay to Script Hash. [Accessed: 7 April 2018], 2012. Available from: https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki

[13] Wuille, P. Dealing with malleability. [Accessed: 7 April 2018], 2014. Available from: https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki

[14] Lau, J.; Wuille, P. Transaction Signature Verification for Version 0 Witness Program. [Accessed: 8 April 2018], 2016. Available from: https://github.com/bitcoin/bips/blob/master/bip-0143.mediawiki

[15] Lombrozo, E.; Lau, J.; et al. Segregated Witness (Consensus layer). [Accessed: 8 April 2018], 2015. Available from: https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki

[16] Weigl, D. Derivation Scheme for P2WPKH-nested-in-P2SH based accounts. [Accessed: 2 April 2018], 2016. Available from: https://github.com/bitcoin/bips/blob/master/bip-0049.mediawiki

[17] Wuille, P.; Maxwell, G. Base32 address format for native v0-16 witness outputs. [Accessed: 8 April 2018], 2017. Available from: https://github.com/bitcoin/bips/blob/master/bip-0173.mediawiki

[18] SatoshiLabs. API Workflows. [Accessed: 14 April 2018]. Available from: http://doc.satoshilabs.com/trezor-tech/api-workflows.html

[19] Realm Report Q4 2017. [Accessed: 5 December 2017]. Available from: https://realm.io/realm-report/2017-q4/

[20] Rusnak, P.; et al. Messages for TREZOR communication. [Accessed: 16 April 2018]. Available from: https://github.com/trezor/trezor-common/blob/master/protob/messages.proto

[21] Schneider, N.; Corallo, M. URI Scheme. [Accessed: 17 April 2018], 2012. Available from: https://github.com/bitcoin/bips/blob/master/bip-0021.mediawiki

[22] Morcos, A. Bitcoin Core Fee Estimation Algorithm. [Accessed: 22 April 2018]. Available from: https://gist.github.com/morcos/d3637f015bc4e607e1fd10d8351e9f41

[23] Erhardt, M. An Evaluation of Coin Selection Strategies. [Accessed: 18 April 2018], 2016. Available from: http://murch.one/wp-content/uploads/2016/11/erhardt2016coinselection.pdf

[24] Bilek, K. Format for Bitcoin metadata and its encryption in HD wallets. [Accessed: 22 April 2018], 2015. Available from: https://github.com/satoshilabs/slips/blob/master/slip-0015.md

[25] Rusnak, P.; Palatinus, M.; et al. Symmetric encryption of key-value pairs using deterministic hierarchy. [Accessed: 22 April 2018], 2014. Available from: https://github.com/satoshilabs/slips/blob/master/slip-0011.md

[26] Dworkin, M. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. [Accessed: 23 April 2018], 2007. Available from: https://csrc.nist.gov/publications/detail/sp/800-38d/final

[27] Rusnak, P. Derivation scheme for P2WPKH based accounts. [Accessed: 24 April 2018], 2017. Available from: https://github.com/bitcoin/bips/blob/master/bip-0084.mediawiki

Appendix A

Screenshots

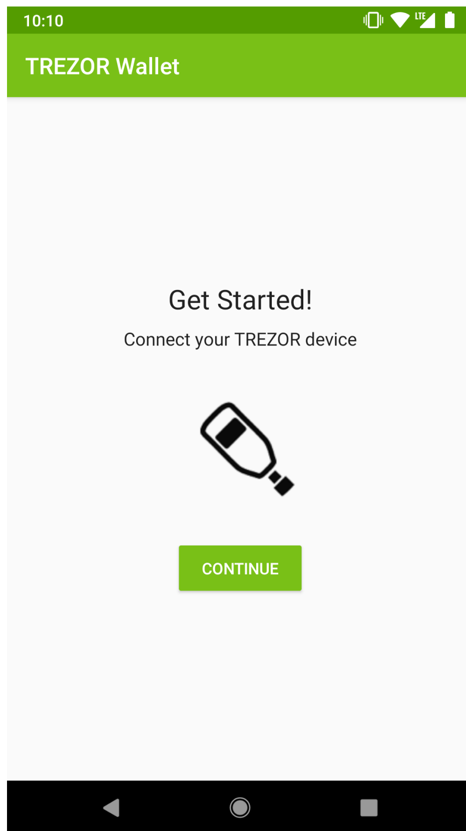


Figure A.1: Initialization

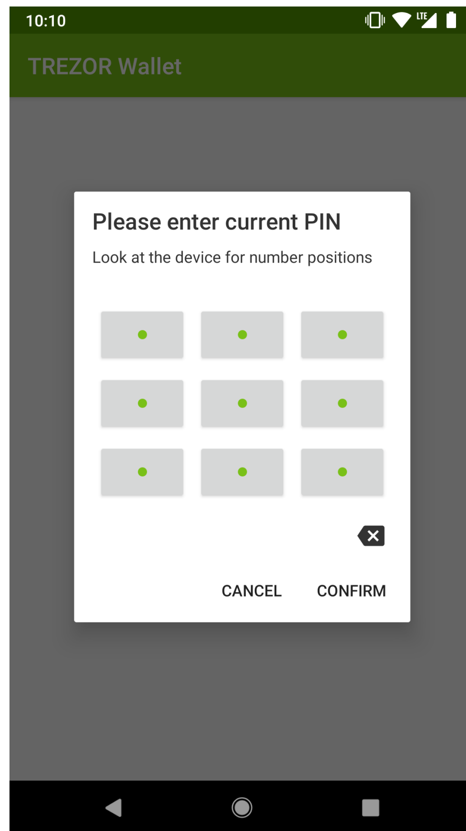


Figure A.2: PIN Request

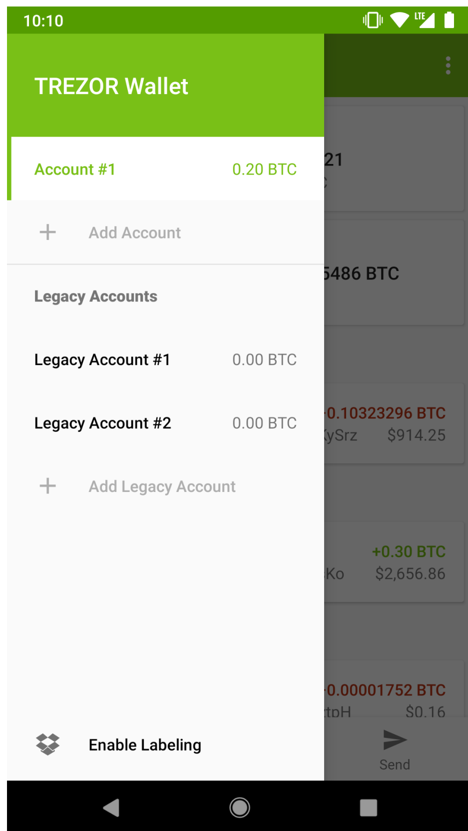


Figure A.3: Accounts List

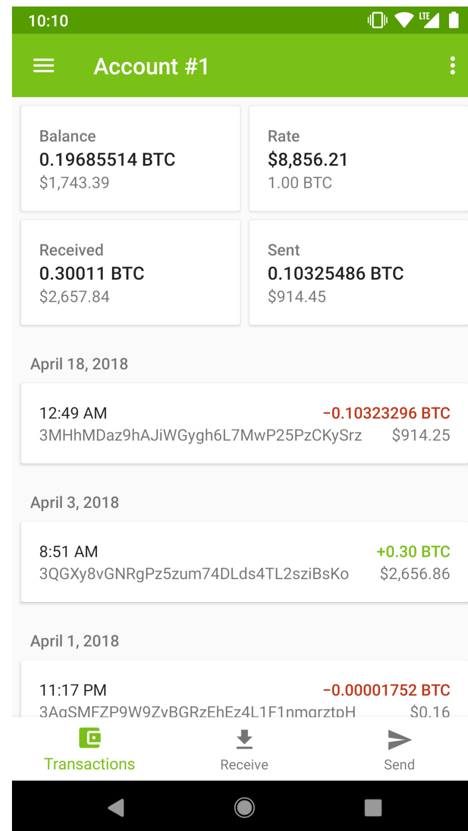


Figure A.4: Transactions List

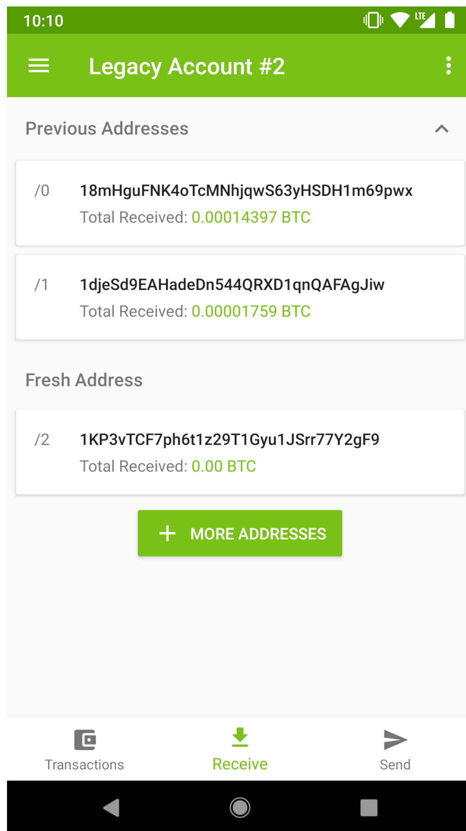


Figure A.5: Receiving Addresses

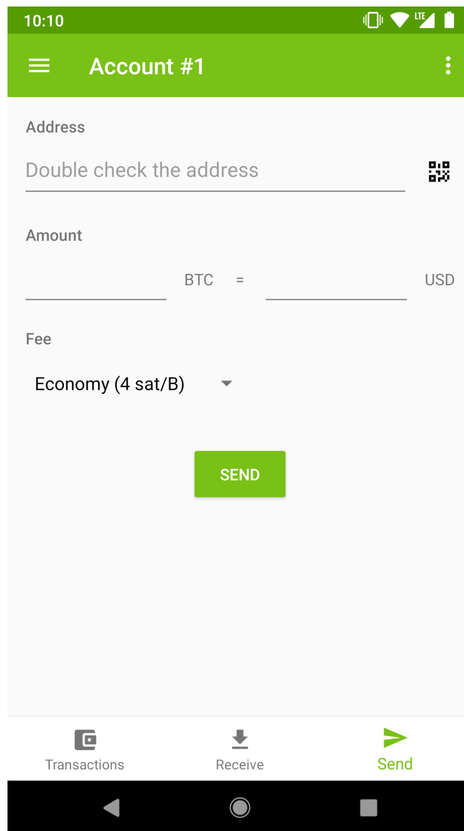


Figure A.6: Transaction Composition

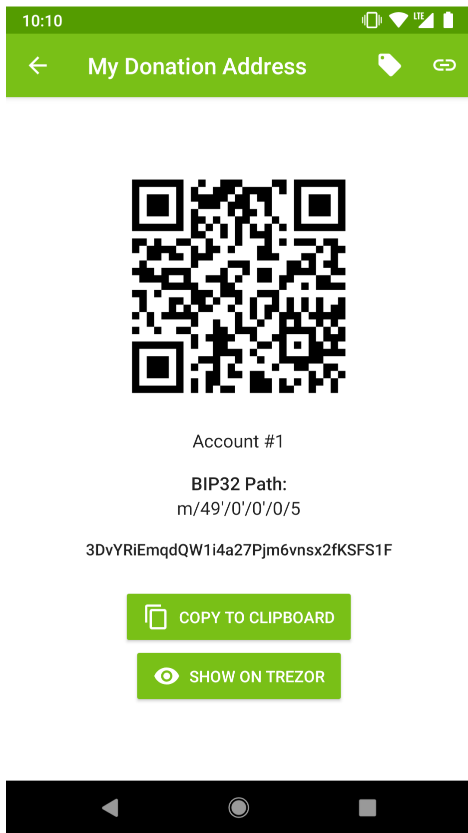


Figure A.7: Address Detail

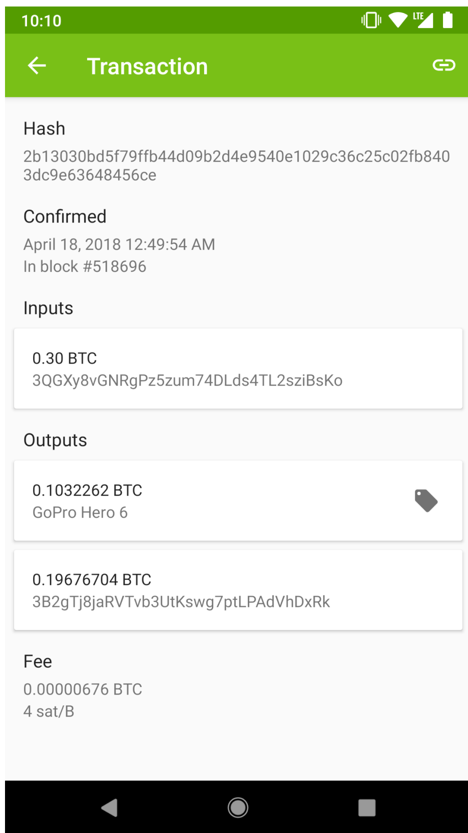


Figure A.8: Transaction Detail

Appendix B

Acronyms

BTC Bitcoin

TX Transaction

UTXO Unspent Transaction Output

SegWit Segregated Witness

HD Hierarchical Deterministic

BIP Bitcoin Improvement Proposal

SLIP SatoshiLabs Improvement Proposal

API Application Programming Interface

SDK Software Development Kit

QR Quick Response

Appendix C

Contents of enclosed SD

```
src
├── trezor-wallet ........................ implementation source codes
│   ├── README.md ............................ installation instructions
│   ├── app ............................................ the wallet app
│   └── trezor-android
│       ├── trezor-lib ....................... the communication library
│       └── trezor-intents .......................... the intents library
└── thesis ............................ LaTeX source codes of the thesis
build
└── trezor-wallet.apk .......................... the compiled APK file
text
└── BP_Skala_Matous_2018.pdf ............ the thesis text in PDF format
```