Internship Report
to obtain the degree of
Master Cryptis

by
Viet-Sang Nguyen

Secure Wallet Application for Cryptocurrency and Blockchain Transactions

supervised by

Dr. Matthieu Rivain
CEO, Senior Cryptography Expert at CryptoExperts
Dr. Aleksei Udovenko
Cryptography Expert at CryptoExperts
Dr. Junwei Wang
Cryptography Expert at CryptoExperts

Enseignant référant

Assc. Prof. Olivier Blazy
University of Limoges

Paris, August 2021

Acknowledgments

First of all, I would like to wholeheartedly express my gratitude to Matthieu Rivain, Aleksei Udovenko and Junwei Wang for giving me the opportunity to do the internship at CryptoExperts and for guiding me throughout 6 months. Without their support, it could not be possible for me to accomplish my internship. I would like to particularly thank Matthieu for always trying to find possible solutions to help me when I asked for something, not only in terms of knowledge but also in other problems. I would like to particularly thank Aleksei for all of his clear explanations whenever I had questions. Although we have not met physically yet because of the pandemic, he always came up with detailed answers very soon after I posed my questions. I also would like to particularly thank Junwei for the meticulous care he gave to me and for always saying "yes" when I needed an additional meeting.

Working at CryptoExperts will definitely be a memorable duration in my career path. I would like to acknowledge Pascal Paillier and Louis Goubin for always showing the warmest and nicest welcome to me as soon as I started working at their company although I did not work with them directly. I am thankful for Sonia Belaïd who always encourages me and cares about my comfort with the working environment. I thank Thibauld Feneuil and Abdul Rahman Taleb for all the relaxing gossip we had at lunch time. Special thanks to Abdul for your help and advice to my problems with the living in Paris.

I am grateful to my master advisors Emmanuel Conchon and Duong-Hieu Phan. They have helped me a lot both in academic and in culture adaptation since the first day of my arrival in France. I would like to thank Olivier Blazy for being the reference professor for my internship.

It was very exciting to enjoy studying Information Security and Cryptography with my friends: Linh Le, Hoa Nguyen, Huyen Tran, Dung Bui, Ky Nguyen, Hoang-Minh Nguyen, Duc-Anh Nguyen, Tien Nguyen, Julien Maillard, Adel Aouabed, Jérémy Lagorce, Valentin Puech Hasnae Elaz and many others. Thank you all.

Finally, my warmest gratitude is to my family for everything.

Abstract

In this work, we study the cryptographic methods used in Bitcoin and Ethereum including key generation and ECDSA signature. Then, we study in details the components of a transaction in these two cryptocurrencies. Based on this background knowledge, we design and develop a cryptocurrency wallet application with the high considerations of security and privacy running on Android smartphones. In our implementation, we focus on the protection of secret keys in the ECDSA signature with the orientation of leveraging white-box cryptography. Each secret key stored in the wallet application is protected by a secure container, namely, a token. We develop a trusted server which is responsible for generating tokens. We also survey some typical attacks and corresponding countermeasures on ECDSA, then apply these countermeasures to the implementation.

For the results, our wallet application is capable of receiving coins from others and sending coins to others by creating new transactions and broadcasting them to the decentralised network. The trusted server is able to generate and transfer tokens when it receives requests from the wallet application. A token can only be operated by the corresponding ECDSA signature generator of the server. Regarding the white-box implementation of ECDSA, it is still in progress of researching and developing.

Contents

1. Introduction **1**
1.1. Cryptocurrency and Blockchain 1
1.2. The need of a secure wallet application 2
1.3. White-box cryptography 2
1.4. Our Main Work 3
1.5. Chapter Organisation 4

2. Elliptic Curve Group Operation **6**
2.1. Definition of Elliptic Curve 6
2.2. Point Addition on Elliptic Curve 6
2.3. Scalar Multiplication on Elliptic Curve 7

3. Managing Keys and Addresses **10**
3.1. Roles of Keys and Addresses 10
3.2. Derivation of Address from Key 11
3.3. Privacy Problem of Reusing Addresses and Solution 12
3.4. Deterministic Wallet and Key Derivation 13
3.5. Address Balance and Account Balance 19

4. Creating Bitcoin Transaction **21**
4.1. Transaction at a first glance 21
4.2. Transaction Components 22
4.3. Create a New Transaction 23

5. Creating Ethereum Transaction **28**
5.1. Blockchain and World State 28
5.2. Transaction Components 30
5.3. Create a new transaction 31
5.4. Public Key Recovery 32

6. Wallet Implementation **34**
6.1. Design Pattern: MVVM 34
6.2. Database 35
6.3. API Usages 36
6.4. Workflow of Wallet Application 38
6.5. Usecase: Bitcoin Transaction Creation 41

7. Secure Wallet Architecture **44**
7.1. Offline Server 45
7.2. Secure Wallet Application 47
7.3. Security Analysis 48

8. Attacks and Countermeasures on ECDSA **50**
8.1. Reusing nonce 50
8.2. Simple Power Analysis 51
8.3. Correlation Power Analysis 52
8.4. Differential Computation Analysis 54

9. Conclusion and Future Work **56**

Bibliography **57**

A. APIs used in the project **59**
A.1. Bitcoin 59
A.2. Ethereum 59
A.3. Exchange Price 60

B. Serialisation **61**
B.1. Extended Key 61
B.2. Bitcoin Transaction 61
B.3. Ethereum Transaction 63

1. Introduction

1.1. Cryptocurrency and Blockchain

A cryptocurrency is an electronic and entirely virtual currency consisting of many concepts and technologies. Unlike traditional currencies, a cryptocurrency is formed by a distributed, peer-to-peer system and users of the network communicate with each other via the internet. Coins of a cryptocurrency are used as assets and can be transmitted among participants in the network. For digital money, cryptography is the main factor which provides the basis for trusting the legitimacy of a user's claim to value.

The cornerstone of a cryptocurrency is the so-called blockchain which is at the heart of the peer-to-peer network. Blockchain is a distributed ledger recording all valid transactions. In simple terms, a transaction tells the network that a user has transferred some coins under his control to another user. A cryptocurrency account is identified by a string called an *address* derived from a public key and the corresponding private key can be used by its owner to sign a transaction that transfers coins to another account. A transaction is considered as a valid one by the network only if it includes a proof of ownership for the amount of coins whose value is being sent. This proof is called a digital signature and can be validated by anyone with the owner's public key. To spend coins, one must create a valid signature by his secret key and attach this signature in the transaction. Many cryptocurrencies are based on Elliptic Curve Digital Signature Algorithm (ECDSA) (Johnson et al., 2001).

New coins are created through a process called "mining". In this process, miners validate each transaction and gather them into a new block, then involve competing to find a solution for a mathematical problem. This problem takes the information of the new block and the latest block in the ledger as arguments in order to establish a link between the new block and the previous one (which explains the term "blockchain"). The solution is called a "proof-of-work" and found out by using the computer's processing power. In addition to the value sent to a recipient, one normally has to include an amount of fee for the miner who validates his transaction.

1.2. The need of a secure wallet application

Cryptocurrencies have emerged in the last decade after the publication and development of the Bitcoin system (Nakamoto, 2009). As of today, there exists more than one hundred cryptocurrencies with a capitalisation beyond 50 million USD, and the capitalization of the Bitcoin (the highest one) reaches 830 billion USD[1]. At the moment of this writing, one Bitcoin is worth 44,517 USD which is an appealing number and definitely a motivation for attackers to attempt stealing. In addition to solve the longstanding e-cash issue, many new applications have emerged from the blockchain technology. In particular, the ledger capability offered by the blockchain enables a reduction of the cost attributed to the verification of transactions by removing the need for a trusted third party in many applications.

The main security issue with cryptocurrencies resides in the protection of the users' secret keys, which reveals to be a hard task in practice. Even the most ephemeral key exposure allows an adversary to sign a transaction transferring all the coins on the account to another (pirate) account. The signed transaction is then instantly pushed to the peer-to-peer network and quickly added to the blockchain, which makes it irreversible. Many malware programs have been detected to steal cryptocurrencies from owners who use weak protection mechanisms[2].

Secret keys are often stored in a digital wallet on each user's computer or smartphone. Especially, a smartphone is an open and vulnerable environment. An application on a smartphone usually suffers from diverse attacks (Hur and Shamsi, 2017). Therefore, to protect secret keys on smartphones and hence protect users' assets, it is essential to have a secure wallet application.

1.3. White-box cryptography

Traditionally, we have worked with a security model in which the cryptographic primitive is considered as a *black-box* by the attacker. In a black-box model, an attacker can only observe and perform attacks with the input and output of the cryptographic primitives. However, as pointed out in (P. Kocher et al., 1998) and (P. C. Kocher et al., 1999), real-world deployments may leak some sensitive information through timing or power consumption that attackers can exploit to recover the secret key. This gave rise to the gray-box attack model which allows an adversary to recover the key based on side-channel information.

The white-box model considers the assumption that an adversary has full control over the implementation as well as the execution environment. This enables an adversary to perform static analysis on the software, inspect and alter

[1] See at: https://coinmarketcap.com

[2] See at: https://www.coindesk.com/malware-anubis-cryptocurrency-wallets

the memory used, and even alter intermediate results. The goal of white-box cryptography is to protect secret keys embedded in a cryptographic software deployed in an untrusted environment where the software might be attacked by such a powerful adversary.

1.4. Our Main Work

The goal of the internship is to develop a secure cryptocurrency wallet application running on Android devices. The most sensitive component in our wallet, namely, the ECDSA signature generator, is protected in such a way that private keys of users are ensured not to leak during the signing procedure. Moreover, keys are not directly generated as well as stored in the wallet application. The foreseen architecture makes use of a dynamic ECDSA white-box implementation that can operate transactions from tokens. A token can be seen as a secure container for an ECDSA private key which can only be operated by a predetermined white-box implementation and which is locked by a password and environmental fingerprint. A trusted server is responsible for securely generating tokens and transferring them to the wallet application. By doing so, keys are stored in the wallet under the forms of tokens and securely used inside a white-box implementation of ECDSA. Figure 1.1 shows an overview of the wallet architecture.

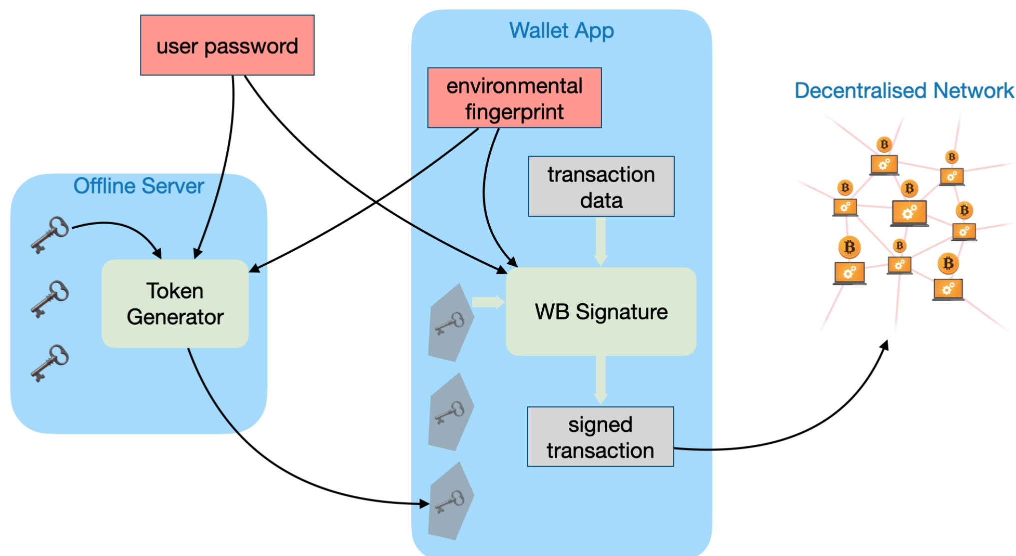


Figure 1.1.: Overview of the secure cryptocurrency wallet architecture

Following the goal described above, the main work we have done in this project during the 6-month internship includes:

1. Studied the key architecture behind cryptocurrencies, then implemented

the key generation scheme which is based on an original seedphrase. We also gave efficient solutions for key management as a wallet application operates with a number of different keys. Moreover, we considered the privacy-preserving aspect and found out solutions for our scheme with the problem of reusing a key for several transactions.
2. Studied the transactions of Bitcoin and Ethereum in detail, then built an Android application which is capable of creating new transactions, sending coins to others by broadcasting transactions to the decentralised network, and receiving coins from others.
3. Designed and developed a token generator running on a server and an ECDSA signature generator in the wallet application with considerations of security. For each pair of these two generators, we established a connection in such a way that tokens generated by a generator are only able to be operated by the corresponding signature generator.
4. Studied possible attacks and countermeasures on ECDSA, then implemented the countermeasures and applied to the signature generator.

1.5. Chapter Organisation

This report is organised as follows:

- Chapter 2 presents the mathematical concepts that we use throughout this report. In particular, this chapter introduces the elliptic curve and the computations on it including point addition and scalar point multiplication.
- Chapter 3 analyses the key and address generation in detail. This generation is mainly based on elliptic curve cryptography and follows the standards of Bitcoin and Ethereum communities which are widely used in almost other wallet applications.
- Chapter 4 first describes the context of a transaction in Bitcoin network, then presents the components of a Bitcoin transaction. After that, we show how to create a new transaction in our wallet and broadcast it onto the decentralised network.
- Chapter 5 first gives an overview of Ethereum network, then presents the components of a Ethereum transaction. Similar to Bitcoin, we also shows the steps to create a new Ethereum transaction and broadcast it.
- Chapter 6 gives the details of the application development. We present the application architecture and the general workflow of the wallet. We also analyse the implementation of a typical usecase, namely, Bitcoin transaction creation.
- Chapter 7 discusses the ECDSA signer in the wallet application and the token generator in the server. We show how to implement these two generators in detail and analyse the security perspectives.

- Chapter 8 surveys some typical attacks on ECDSA and their countermeasure. We also apply the countermeasures for our wallet application.
- Chapter 9 concludes our work and discusses the future work.

2. Elliptic Curve Group Operation

In this chapter, we introduce the mathematical concepts about elliptic curves. Cryptographic computations throughout this project is mostly based on elliptic curve cryptography. We first present the definition, then the addition of two points and the scalar multiplication. We also review some efficient algorithms using for scalar multiplication.

2.1. Definition of Elliptic Curve

An elliptic curve over a field K is a set of points (x, y) which are solutions of a bivariate cubic equation (Menezes, 1992) and defined by a Weierstrass equation:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{2.1}$$

where $a_1, a_2, a_3, a_4, a_6 \in K$ and the discriminant of the curve $\Delta \neq 0$.

A Weierstrass equation can be simplified by applying change of coordinates. With $a, b \in K$, we consider the following cases:

- If $char(K) \neq 2$ and $char(K) \neq 3$, the equation 2.1 can be transformed to
$$y^2 = x^3 + ax + b \tag{2.2}$$
- If $char(K) = 2$, then 2.1 can be transformed to
$$y^2 + xy = x^3 + ax^2 + b \tag{2.3}$$
- If $char(K) = 3$, then 2.1 can be transformed to
$$y^2 = x^3 + ax^2 + b \tag{2.4}$$

A special point $\mathcal{O}$ named the "point at infinity", together with the set of points on an elliptic curve, forms an abelian group.

2.2. Point Addition on Elliptic Curve

We define $P + \mathcal{O} = \mathcal{O} + P = P$ for all P on the curve. In this work, we follow the recommended elliptic curve domain parameters (Brown, 2010) and focus to the case of $char(K) \neq 2, 3$.

Point Addition for $char(K) \neq 2, 3$

Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$, $P \neq \mathcal{O}$ and $Q \neq \mathcal{O}$, be two points on an elliptic curve. The inverse of P is $-P = (x_1, -y_1)$. With $Q \neq -P$, the sum of $P + Q = (x_3, y_3)$ is calculated as

$$x_3 = \lambda^2 - x_1 - x_2$$
$$y_3 = \lambda(x_1 - x_3) - y_1$$

where

$$\lambda = \begin{cases} \dfrac{y_2 - y_1}{x_2 - x_1} & \text{if } P \neq Q \\ \\ \dfrac{3x_1^2 + a}{2y_1} & \text{if } P = Q \quad \text{(Point Doubling)} \end{cases}$$

To calculate the subtraction $Q - P$, we add Q with the point $-P$.

2.3. Scalar Multiplication on Elliptic Curve

The operation of adding a point P to itself d times is called “scalar multiplication” by d and the result is denoted as dP. This is the basic operation of elliptic curve cryptosystems. Scalar multiplication on an elliptic curve is the analogy of exponentiation in the multiplicative group of integers modulo a fixed integer. The security of elliptic curve cryptosystems is based on the hardness of the elliptic curve discrete logarithm problem. Performing the scalar multiplication efficiently and securely is crucial to a cryptographic device.

Double-and-Add Method

A point multiplication can be computed with the straightforward approach using double-and-add method as shown in algorithm 1. This method is analogous to the square-and-multiply algorithm for exponentiation. Considering a balanced exponent d, this algorithm requires $1S + \frac{1}{2}M$ per bit of the exponent on average, where S is the cost of a doubling operation and M is the cost of a point addition.

Binary NAF Method

The “non-adjacent form” (NAF) uses digits from $\{-1, 0, 1\}$ to represent d.

$$d = \sum_{0 \leq i < l} d_i 2^i \text{ where } d_i \in \{-1, 0, 1\}$$

Algorithm 1: Left-to-Right Double-and-Add

Input: point P, secret $d = (d_{n-1}, ..., d_0)_2$
Output: $Q = dP$

```
1  Q ← P
2  for i = n − 2 down to 0 do
3  |  Q ← 2Q
4  |  if d_i = 1 then
5  |  |  Q ← Q + P
6  |  end
7  end
8  Return Q
```

In a scalar multiplication, this representation helps to speed-up the execution since the number of elementary point operations is reduced. We notice that subtraction in elliptic curve has the same cost as addition. Algorithm 2 shows the steps of this method. This algorithm requires $1S + \frac{1}{3}M$ per bit of the exponent on average.

Algorithm 2: Binary NAF Method

Input: point P, secret $d = (d_{n-1}, ..., d_0)_{NAF}$
Output: $Q = dP$

```
1  Q ← P
2  for i = n − 2 down to 0 do
3  |  Q ← 2Q
4  |  if d_i = 1 then
5  |  |  Q ← Q + P
6  |  end
7  |  if d_i = −1 then
8  |  |  Q ← Q − P
9  |  end
10 end
11 Return Q
```

Montgomery Ladder

Let $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$, $P_1 + P_2 = (x_3, y_3)$, $P_1 - P_2 = (x_4, y_4)$ and $2P_1 = (x_5, y_5)$ be points on the curve $y^2 = x^3 + ax + b$.

Montgomery observed that the x_3 and x_5 can be calculated from the x_1, x_2 and x_4 without using any y-coordinate.

$$x_3 = \frac{2(x_1 + x_2)(x_1x_2 + a) + 4b}{(x_1 - x_2)^2} - x_4 \tag{2.5}$$

$$x_5 = \frac{(x_1^2 - a)^2 - 8bx_1}{4(x_1^3 + ax_1 + b)} \tag{2.6}$$

Taking advantage of this observation, the multiplication can be computed by a sequence of pairs $(Q, H) = (sP, (s+1)P)$. This sequence always ensures the property $H - Q = P$. Algorithm 3 shows the details of this method. This algorithm always requires $1S + 1M$ per bit, regardless of the value of d. Since the number of operations executed in each iteration does not depend on the scalar d, Montgomery Ladder is a constant-time scalar multiplication algorithm.

Algorithm 3: Montgomery Ladder

Input: point P, secret $d = (d_{n-1}, ..., d_0)_2$
Output: $Q = dP$

```
1  Q ← P
2  H ← 2P
3  for i = n − 2 down to 0 do
4  |  if d_i = 0 then
5  |  |  H ← Q + H
6  |  |  Q ← 2Q
7  |  else
8  |  |  Q ← Q + H
9  |  |  H ← 2H
10 |  end
11 end
12 Return Q
```

3. Managing Keys and Addresses

In this chapter, we discuss in more detail how we generate and manage keys and addresses in the wallet application. At first, we describe the cases in which keys and addresses are manipulated in the section 3.1. Next, we present the derivation of addresses in Bitcoin and Ethereum in the section 3.2. Section 3.3 analyses the privacy problem in case we reuse addresses, which leads to the solution of the tree-like structure for keys presented in the section 3.4. This solution is based on Bitcoin Improvement Proposals (BIP) and widely used as standards in cryptocurrency wallet applications in practice.

3.1. Roles of Keys and Addresses

There are two main features which a cryptocurrency wallet must support. The first feature is to inform users how many coins they have. In other words, the wallet app has the ability to detect when there is someone sending coins to addresses managed by the app, and when users send coins from this wallet to others. The second feature is to perform transactions and broadcast them into the blockchain network. When users want to pay their bills or send coins to somebody, the wallet has to create new valid transactions and publish them to the decentralized network.

Keys in cryptocurrencies, as an asymmetric key encryption scheme, come in pairs where each pair consists of a private key and a public key. An address is a digital fingerprint of a public key. The combination of these three components (private key, public key, address) establishes the ownership of coins.

- Private key: it is used to generate a digital signature which is an essential part of transactions. Without having a valid signature, transactions cannot be included in the blockchain. Therefore, one has the control of coins if he/she has a copy of the private key.
- Public key: when a new transaction is published in the decentralized network, one of the miners have to check whether this transaction is valid. The miner uses the public key in this validation process.
- Address: it specifies the origin or the destination of coins in a transaction. An address has its balance which is the amount of coins controlled by its corresponding private key. The length of an address is shorter than the length of its public key, and hence, the size of a transaction is decreased.

3.2. Derivation of Address from Key

A cryptocurrency wallet contains a collection of key pairs. Each pair consists of a private key and a public key. We denote the private key as d, the public key as Q and the address as A. The private key is a number and it is usually picked at random. From a private key, we use elliptic curve multiplication to generate its public key. And from a public key, we use a one-way hash function to generate a bitcoin address. This procedure is depicted in the figure 3.1. This is the common way to derive an address from its key. However, there are some differences in the hash function used by each cryptocurrency.

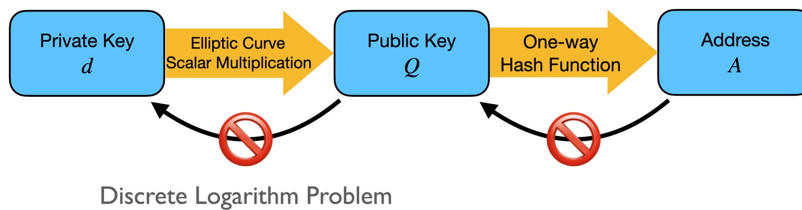


Figure 3.1.: Derive address from its key

Bitcoin Address

From a public key Q, Bitcoin uses SHA256 and RIPMD160 to compute the corresponding address. Note that the concatenation of x-coordinate and y-coordinate of Q is the input of this hash function.

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

The result of this hash function hash the length of 160 bits. Then, it is encoded by Base58Check. Base58 is a subset of Base64, using uppercase and lowercase letters and numbers, but omitting some characters that frequently cause confusions and can appear identical when displayed in some fonts.

$$Alphabet_{Base58} = Alphabet_{Base64} \setminus \{0, O, l, I, +, /\}$$

Base58Check encoding uses Base58 alphabet to encode and adds a checksum part at the end of the result. This checksum is generated by putting the concatenation of the version and the payload into double SHA256 hash. We then take the first 4 bytes as the checksum.

A version number is the prefix on each address which indicates the type of the address. Possible types include mainnet address, testnet address, extended public key, etc.

Ethereum Address

Unlike Bitcoin, Ethereum uses only one hash function Keccak256 to produce an address from its public key. Then, we take the last 20 bytes of the hash output as the address.

$$A = \text{last_20_bytes}(\text{Keccak256}(Q))$$

Ethereum uses hexadecimal format for its addresses. The checksum part is optional to include. This checksum is calculated by capitalisation defined in (Buterin and de Sande, 2020). In short, it puts the address into Keccak256 hash and take the first 20 bytes. Consider each letter in the hexadecimal format of the output, if it is greater than or equal to $0x8$, capitalize the alphabetic address character at the same position in the address.

3.3. Privacy Problem of Reusing Addresses and Solution

In reality, each person has a secret PIN to access the bank account and a public account number to receive money. We might think it is analogous that a person in a cryptocurrency network has one private key to control the asset and one public address to receive coins. In fact, it is fine if we interact with the blockchain network by only one address. However, reusing addresses is a bad idea in practice.

Once we spend coins, our public key is revealed since it is included in the transaction and hence in the ledger. The secret key is protected by the discrete log problem, which is unlikely to be broken in the near future. An idea widely used in the cryptocurrency community is to split the total balance into smaller amounts and control them by many secret keys. By this method, we have many addresses in our wallet. Ideally, we use each address only once and do not reuse it again in the future.

The other important reason why we should not reuse addresses is to protect the privacy of users. Consider the case when Alice sends coins to Bob, the receiver (Bob) will know exactly that the address of the sender in the transaction (which is clear for Bob and for everyone since it is included it the blockchain) is Alice. If Alice reuses a single address for all transactions, Bob can track and know everything about her transactions.

Furthermore, there are some websites which determine the richest addresses in the blockchain. Figure 3.2 shows an example of the recent richest address. It has totally 644 transactions in history. As a consequence, this address and its owner can easily become the target of attackers who wants to steal the private key and hence take control over the coins. The dangers not only come from

attacks in the computer world but also can be blackmails or crimes taking place in reality.

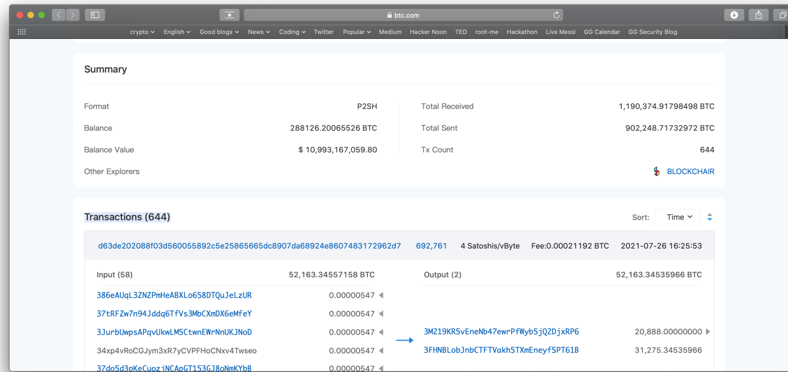


Figure 3.2.: The richest address in Bitcoin has 644 transactions

In can be concluded that it is very important to avoid reusing addresses in practice. Therefore, a wallet application has to manage many private keys and addresses in order to ensure that each address should be used only once. Depending on whether the keys contained in wallets are related to each other or not, the cryptocurrency wallets are categorised into two primary types.

- **Non-deterministic Wallet**: each key is independently generated from a random number. There is no relation between the keys in the wallet. The disadvantage of this approach is that we have to keep copies of all keys and it needs much effort to manage them. We notice that if the secret keys are lost, our asset is irrevocably inaccessible.
- **Deterministic Wallet**: keys are derived from a single master key, called the *seed*, by using a one-way hash function. Compared to non-deterministic wallet, it is sufficient to keep secure only the seed. From this seed, we can recover all keys that the wallet contains. This type of wallet is widely used in practice so that we can use a seed with different wallet applications. In this work, we also use this deterministic key generation which is discussed in the section 3.4.

3.4. Deterministic Wallet and Key Derivation

In this section, we present in detail the key generation which is used in our wallet. Keys in a deterministic wallet are derived from a single seed. There are some different key derivation methods as cryptocurrency wallet technology has matured. We follow the most popular and commonly used method in the industry, which uses a tree-like structure. It is known as *hierarchical deterministic* or HD wallet.

To create a new HD wallet, there are some standard methods we must follow:

- Mnemonic code for generating deterministic keys, based on BIP-39 (Palatinus et al., 2013). This defines how we generate the seed.
- Hierarchical Deterministic Wallets, based on BIP-32 (Wuille, 2013). This defines how we derive private keys and public keys in HD wallets.
- Multi-Account Hierarchy for Deterministic Wallets, based on BIP-44 (Palatinus and Rusnak, 2014). This defines how we derive keys for different cryptocurrency (e.g, Bitcoin, Ethereum) from a single seed.

We discuss each of these standards in the the following sections.

Mnemonic Words and Seed Generation (BIP-32)

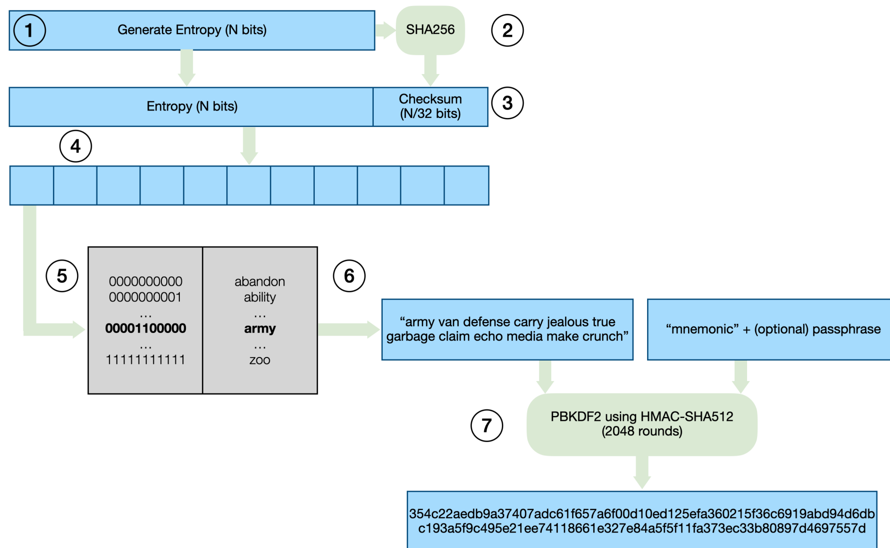


Figure 3.3.: Generate mnemonic words and seed

Mnemonic is a sequence of English words used to create seeds for HD wallets. This increases the convenience for users since mnemonic is easy to transcribe (compared to a hexadecimal string), export and import across wallets.

The steps of generating a new mnemonic code is as follows:

1. Create a random sequence (entropy) of N bits, $N \in \{128, 160, 192, 244, 256\}$.
2. Create a checksum of the entropy by taking the first $N/32$ bits of its SHA256 hash.
3. Add the checksum to the end of the entropy.
4. Divide the entropy into sections of 11 bits.
5. Map each 11-bit value to a word from the predefined dictionary of 2048 words.

6. The mnemonic code is the sequence of words

In our wallet, we choose $N = 128$ and the mnemonic code consists of 12 English words. We then put the string of these words, together with the string constant "mnemonic" concatenated with an optional passphrase string from user as the salt, into a PBKDF2 function using 2048 rounds of hashing with the algorithm HMAC-SHA512. The output of the PBKDF2 function is the seed of length 512 bits. This is the step 7 in the figure 3.3.

Key Derivation (BIP-39)

In the previous section, we discussed how to generate mnemonic words and then how to generate the seed. Each key in a HD wallet is deterministically derived from the seed. In this section, we discuss in detail these derivations.

Master Key from Seed

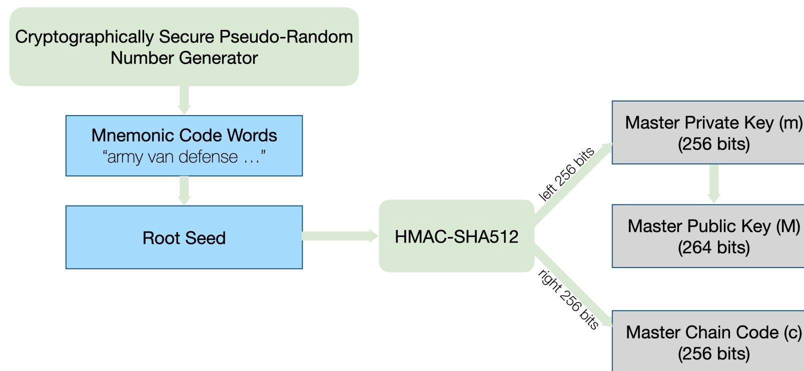


Figure 3.4.: Derive master key from seed

First, the master key pair is derived by putting the root seed into a HMAC algorithm with the hash function SHA512. The result is split into two halves: 256-bit left half and 256-bit right half. We take the left one as the master private key denoted by m, the right one as the master chain code c. Note that from m, we can easily calculate the master public key M by an elliptic curve multiplication. The derivation of master key is shown in the figure 3.4.

Child private key from parent private key

From a parent private key m_p, its public key M_p and chain code c_p, we can derive a child private key m_c, its public key M_c and chain code c_c. Using a HMAC-SHA512 function with (M_p, c_p, i) as the input, we obtain the tuple (l, c_c) as the output, where l and c_c are the 256-bit left half and 256-bit right half

of the output respectively (figure 3.5). i is a 32-bit index number, $i \in [0, 2^{32} - 1]$, and it allows us to build a tree-like structure.

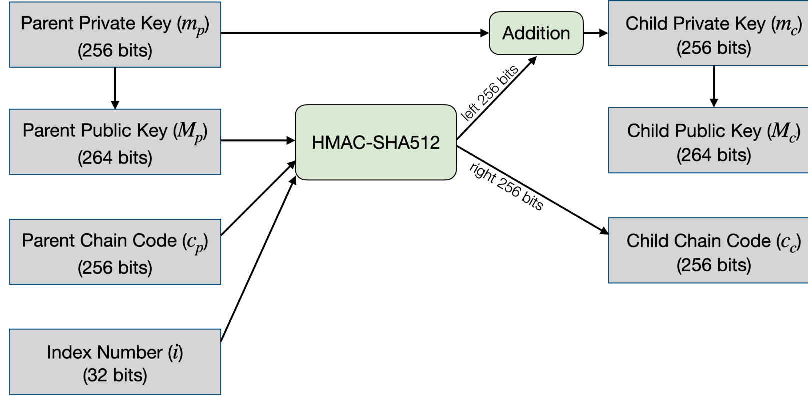


Figure 3.5.: Derive child private key from parent private key

$$(l, c_c) = \text{HMAC-SHA512}(M_p, c_p, i) \tag{3.1}$$

We take c_c as the chain code of the child. For the child private key m_c, we calculate it by adding the left part l and the parent private key m_p.

$$m_c = m_p + l \tag{3.2}$$

The child public key is easily derived by the scalar multiplication between the private key m_c and the base point G of the elliptic curve.

$$M_c = m_c \times G = (m_p + l) \times G \tag{3.3}$$

Extended Private Key: We notice that by this method, knowing a pair (m_p, c_p) is enough to derive its children. In practice, we call the concatenation of m_p and c_p is an "Extended Private Key", denoted by xprv. We have xprv $= m||c$.

Child public key from parent public key

An advantage of this derivation method is that a child public key can be derived from a parent public key without the need of any private key. Similar to the child private key derivation, we put (M_p, c_p, i) into a HMAC-SHA512 function and it produces the output (l, c_c), as shown in figure 3.6. The child public key is calculated by the equation 3.4 as follows.

$$M_c = M_p + l \times G \tag{3.4}$$

We can write the equation 3.4 in a different way as 3.5. Compared with the equation 3.3, we can see that it is not necessary for a private key to involve in the process of public key derivation.

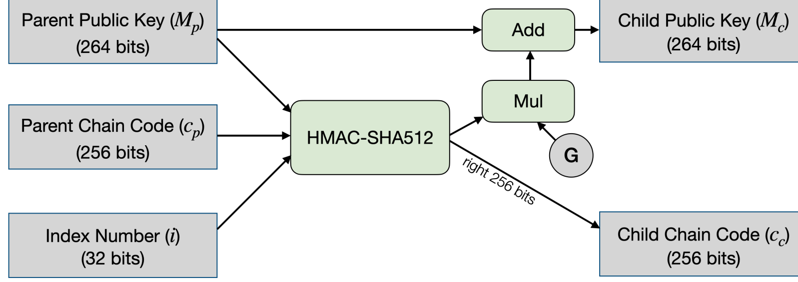


Figure 3.6.: Derive child public key from parent public key

$$M_c = m_p \times G + l \times G = (m_p + l) \times G \tag{3.5}$$

Extended Public Key: Analogous to xprv, knowing a pair (M_p, c_p) is enough to derive its children. We call the concatenation of M_p and c_p is an "Extended Public Key" and denote as xpub. We have xpub $= M||c$.

Hardened child key derivation

We note that xprv and xpub have the same chain code c. We consider a case when the private key of a node in the tree structure is somehow leaked as depicted in the figure 3.7. An attacker can use that private key, say m_2, and the chain code c in the xpub to deduce its parent private key (by 3.6, 3.7) and all of its children private keys.

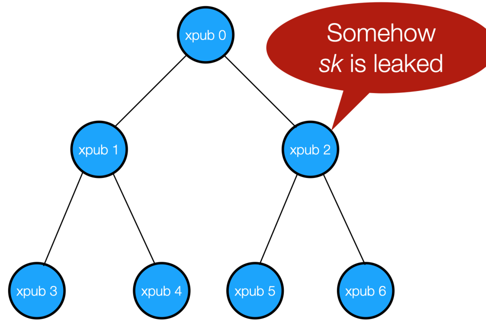


Figure 3.7.: Example of leaking private key at a node

$$(l, c_2) = \text{HMAC-SHA512}(xpub_0, i) \tag{3.6}$$

$$m_0 = m_2 - l \tag{3.7}$$

This comes to an alternative solution for private key derivation called *hardened child key derivation*. Hardened derivation aims to break the relationship between parent public key and the child chain code. It replaces the parent public key M_p by the parent secret key m_p in the input of the HMAC-SHA256 function. We can see in the figure 3.8 this hardened derivation.

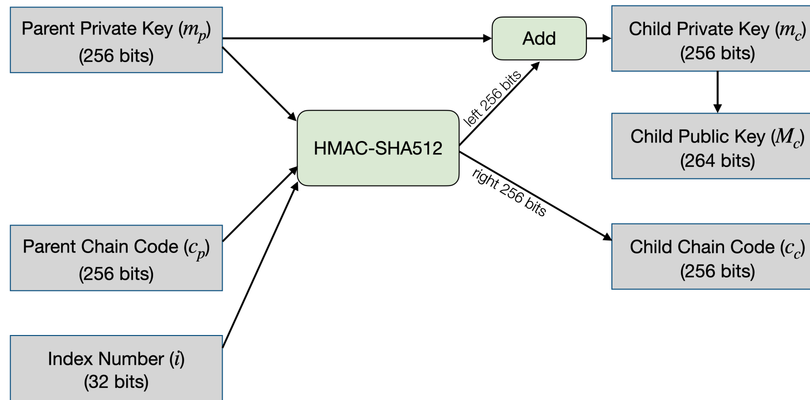


Figure 3.8.: Hardened private key derivation

It is important to know that when we use the hardened derivation, we cannot derive a child public key by its parent public key as before. In practice, besides normal derivation (figure 3.5), we use hardened derivation to increase the security of important parts in the tree structure. From a parent private key, we use the value of the index in the derivation to indicate that the derivation is normal version or hardened version. In particular,

- $i \in [0, 2^{31} - 1]$: normal derivation
- $i \in [2^{31}, 2^{32} - 1]$: hardened derivation. For convenience, i' is denoted for a hardened index, where $i' = 2^{31} + i$. For instance, $2'$ indicates the third child derived by harden version, and its actual index is $2^{31} + 2$.

Identifier Path (BIP-44)

In order to manage keys in a HD wallet effectively, each key in the tree structure is identified by a path. Levels of the tree are separated by a slash (/). The general path is specifies as follows:

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

where

- `m`: is the master private key. It can be `M` if we use the master public key to derive.

- `purpose`: is always set to 44 (standing for BIP-44). The symbol prime denotes the hardened derivation.
- `coint_type`: represents the coin. For instance, 0 is Bitcoin, 1 is Bitcoin Testnet, 60 is Ethereum.
- `account`: is for organisational purposes. One can have many accounts with one mnemonic code.
- `change`: it is either 0 (receiving addresses), or 1 (change addresses). An address used to show others and get coins from them is called a receiving address. In Bitcoin, when the amount of coins that a sender includes to a transaction is greater than the amount she wants to send, the change amount is not automatically sent back to her. To get this change amount, the sender has to specify one of her addresses as a destination of the transaction. This address is called a change address.
- `address_index`: it is i in the derivation.

Figure 3.9 shows an intuitive understanding of the BIP-44 specification.

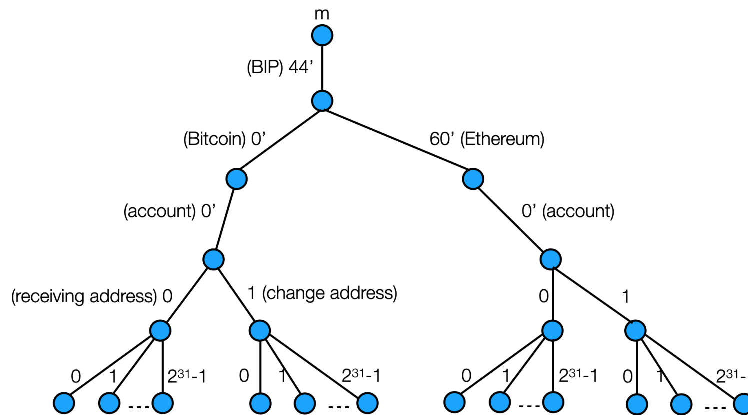


Figure 3.9.: Tree structure of BIP-44 specification

3.5. Address Balance and Account Balance

Remind that each address has a non-negative balance. It is easy to receive coins from others by showing them one of our addresses. As presented in the previous section, we follow BIP-44 to organise keys and addresses in the wallet application. It means that an account can have many addresses containing coins at a moment. Therefore, to obtain the total balance of an account, we have to retrieve the information about balance for each address, then calculate the sum of them.

However, it is impractical to traverse and get balance for all 2^{32} addresses. The BIP-44 specification instructs that we need to start from the address with

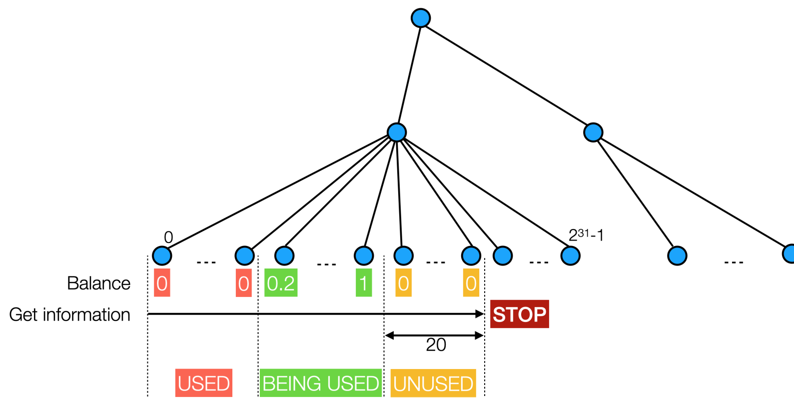


Figure 3.10.: Balance of an account

index 0, then $1, 2, 3, \ldots$ until we reach 20 consecutive unused addresses. An address never involved in any transaction is called an *unused address*. We also define an address with a positive balance which can be spent in the future as a *being used address*. An address with zero balance and already involved in one or more transactions is an *used address* and it should not be reused afterwards. If we choose the strategy which prioritises to spend the being used addresses with the smallest indices first, the tree structure of keys can be visualised as the figure 3.10.

4. Creating Bitcoin Transaction

In this chapter, we present how we create new Bitcoin transactions in our wallet application. Section 4.1 gives the context of a transaction. Then, we analyse required components in a transaction in the section 4.2 and introduce the steps to create a new transaction in the section 4.3. Additionally, we explain how Bitcoin prevents the double-spending problem and how a miner can validate a new transaction in this chapter.

4.1. Transaction at a first glance

A transaction is able to specify basic information such as the sender, the receiver, and the amount of coins. In addition, it also provides the proof which indicates that the sender has the control over her assets involved in the transaction. This proof is called the signature of the sender. Totally, a transaction must contain four essential components: sender, receiver, amount and signature.

However, as presented in the previous sections, the asset of a user is split into smaller amounts contained in many addresses. It can happen that using the coins of only one address is not affordable for the amount that the sender wants to transfer. In Bitcoin, we can combine multiple addresses in order to have enough coins for the sender in a transaction. In fact, a transaction can be included with one or more inputs and one or more outputs, where an input is a script referring to coins of an address under control of the sender, and an output is a script indicating a certain amount that an address of the receiver is planned to receive.

A transaction does not automatically send back the change coins (if any) to the sender. Hence, to take the change back to her wallet, she has to specify an output with one of her addresses and the change amount. Figure 4.1 shows an example in which Alice includes two inputs and two outputs in her transaction to send coins to Bob and get the change back to her wallet.

Transaction Fee

To be verified and appended into the blockchain ledger by miners, it is necessary for a transaction to include an amount of fee. Depending on how much fee we are willing to pay, our transaction will be processed early or late. This fee is automatically calculated by subtracting the total amount of the inputs by the total amount of the outputs (see figure 4.1 for an instance).

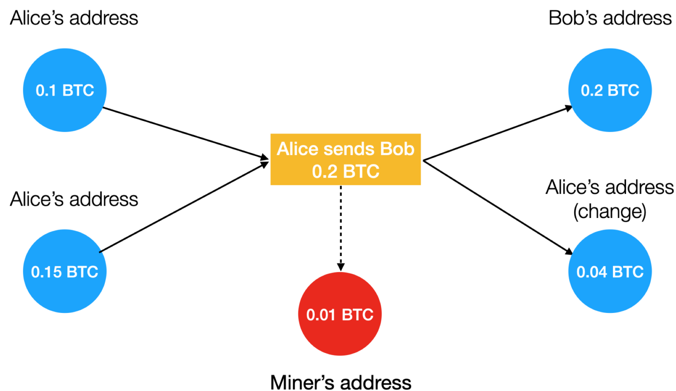


Figure 4.1.: Understanding basic components of a Bitcoin transaction

In practice, the fee for a transaction is not a fixed value. It is chosen by the wallet and usually based on two factors: the current resource of all miners in the network and the number of transactions waiting for validation. Additionally, fee also depends on the size of the raw transaction since a block in the ledger has a limited space.

In our implementation, we use an API to get a suitable amount of fee[1]. This offers the users three fee levels. The higher the level is, the faster the transaction will be verified.

4.2. Transaction Components

A transaction can have one or more inputs and one or more outputs. Figure 4.2 (left) shows the example of a transaction with two inputs and two outputs. The list of inputs and outputs are called `vin` and `vout` respectively. Letter "v" stands for the word "vector". To make things easier to understand, we explore the components of an output in the `vout` list first. An element of `vout` consists of the two following fields:

- `value`: is the amount of coins sending to the address of this output.
- `scriptPubKey`: is a script containing some opcodes (we discuss this script in the next sections) and the hash of the receiver's public key.

An input in the `vin` list consists of the three following fields:

- `txid`: is the identifier number of the confirmed transaction which this current transaction refers to. We notice that the `txid` of a confirmed transaction is generated and included by a miner. For the new transaction we create and broadcast to the decentralized network, it is unnecessary to have this field (see figure 4.2, right).

[1] https://bitcoinfees.earn.com

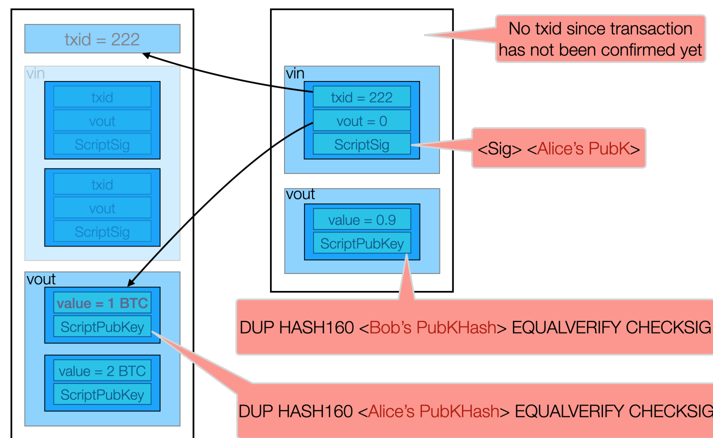


Figure 4.2.: Example of transaction components (left) and creating a new transaction (right)

- `vout`: is the position of the output in the `vout` list of the referred transaction.
- `scriptSig`: is a script containing the signature corresponding to this input (we discuss this script in the next sections) and the public key of the sender. Signature is the proof for the ownership of the sender on the coins she uses in a transaction. Since it is possible to have one or many inputs in a transaction, there can be one or many messages to be hash. We explain how to generate messages hash in the next sections.

If we notice carefully, it is not hard to realise that there are no addresses of the sender and the receiver in a transaction. Also, no information expresses that how many coins the sender includes in the transaction. Actually, we can derive these information easily from the fields described above. As presented in the previous chapter, an address is derived from its public key by hashing. Hence, based on the public key included in the `scriptSig` of an input, the address of the sender can be computed. Similarly, the hash of the public key in an output's `scriptPubKey` is used to derive the receiver's address. Regarding the amount of coins that the sender includes in a transaction, this information is determined through some references to confirmed transactions which is discussed in the next sections in this chapter.

4.3. Create a New Transaction

To create an input for a new transaction, we have to determine the output of a confirmed transaction that we will refer to. This reference indicates that the coins in the output belong to the sender as its `scriptPubKey` contains the hash of her public key. Then, we use the signature signed on this transaction by the private key corresponding to that public key to generate a `scriptSig`. For an

output creation in a new transaction, we declare a receiver by a `scriptPubKey` containing the hash of his public key along with the transferred value.

Let us consider the example shown in the figure 4.2, Alice has 1 BTC since there is an output with `value` 1 containing the hash of Alice's public key in its `scriptPubKey` of a confirmed transaction (`txid`= 222, left). Alice wants to send Bob 0.9 BTC and she creates a new transaction (right). The input of this new transaction refers to the transaction with `txid`= 222 and the position of the output in the `vout` list (`vout` = 0). Alice proves her ownership of these coins by the signature included in the `scriptSig`. She then specifies Bob as the receiver by the `scriptPubKey` carrying the the hash of Bob's public key in the output. Together with the `scriptPubKey`, she also specifies the value Bob will receive after this transaction is validated. In this example, the transaction fee a miner will get is $1 - 0.9 = 0.1$ BTC.

Unspent Transaction Output (UTXO) and Double Spending

When Alice creates a new transaction, she needs to refer to one or many outputs containing the hash of her public key in confirmed transactions. However, these outputs are referred at most once, otherwise, it causes double-spending problems. By this method, Nakamoto claimed that the Bitcoin network resists against double-spending (Nakamoto, 2009).

If an output has not been referred yet, it is call an unspent transaction output, or an UTXO for short. Balance of an address is the sum of values in all UTXOs containing the hash of the public key corresponding to that address in the `scriptPubKey`. Therefore, the rough idea to get balance for an address is to traverse all transactions in the blockchain, one by one, and find its UTXOs. Nevertheless, in practice, techniques related to database and queries are utilised in order to get information quickly.

Listing 4.1 shows an example of an UTXO information. The important information includes `vout`, `scriptPubKey` and `value`.

```
{
  "tx_hash": "92c07941b5df8486be63c212455daf59477a32167389fd47f14b8713fd",
  "tx_output_n": 1,          // vout
  "value": 10000,
  "spent": false,
  "confirmations": 1527,
  "confirmed": "2021-08-05T23:39:35Z",
  "script": "76a914f91aae82e7ae4ed5bb5ef68788b85bf7e04ce7e188ac"
}
```

Listing 4.1: Information of an UTXO

How miners validate a transaction

After being published to the decentralised network, a transaction is appended to a mempool (waiting list), then validated by miners. This validation includes checking its references to avoid double-spending problems, checking the ownership of the sender by cryptographic computations with signatures, and ensuring that the input amount is greater than or equal to the output amount.

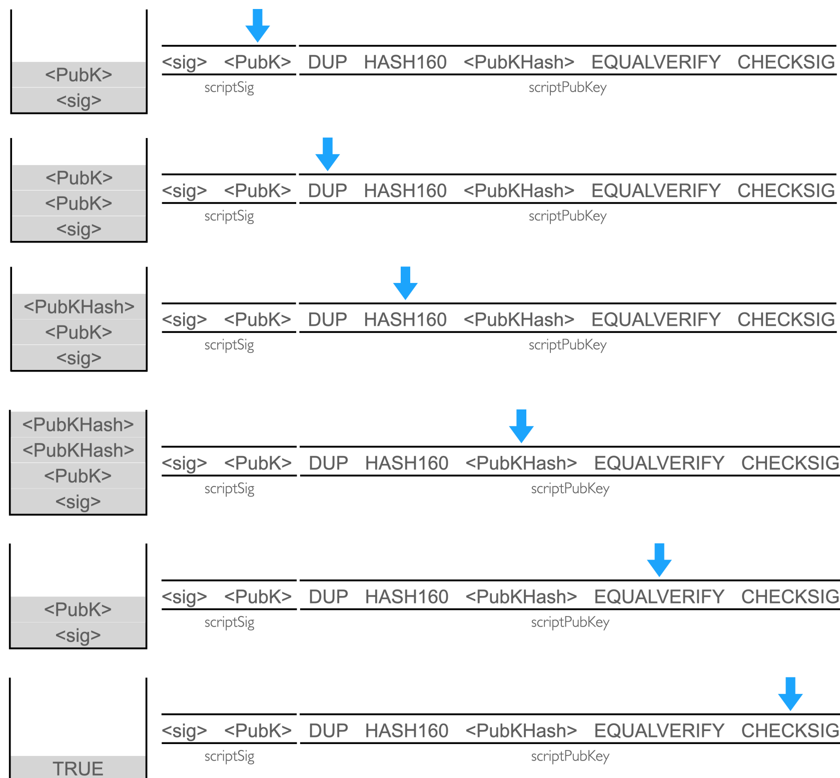


Figure 4.3.: Execution of `scriptPubKey` and `scriptSig`

To establish a valid reference from an input of a new transaction to an UTXO, miners use a program executed by a stack with the `scriptSig` and the referred `scriptPubKey` of the sender. The input of this stack is the concatenation of `scriptPubKey` and `scriptSig` as follows.

```
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG
```

This concatenation contains some opcodes denoting operations in a stack:

- `DUP`: this operation takes the variable on the top of the stack, then duplicates it and pushes those two back to the stack.
- `HASH160`: this operation takes the variable on the top of the stack (expectedly a public key, denoted by `PubK`), then computes the value `PubKHash` = RIPE160(SHA256(`PubK`)) and pushes this value into the stack.

- `EQUALVERIFY`: this operation takes the two variables on the top of the stack and compares them. If they match, both are removed and the execution continues.
- `CHECKSIG`: this operation takes the two variables on the top of the stack (expectedly a public key `PubK` and a signature `<sig>`), then validates the signature. If the signature is valid, it pushes `TRUE` on the top on the stack.

In a transaction, the receiver is locked to an output by a `scriptPubKey`. Thus, `scriptPubKey` is sometimes called the `lockingScript`. To be able to spend the coins held by a `lockingScript`, we use a suitable `scriptSig` which is also called the `unlockingScript`. The unlocking procedure makes use of a stack to execute the concatenated script above as depicted in the figure 4.3. If the stack is empty at the end and returns true, the value of UTXO is unlocked successfully and able to be spent. (A. M. Antonopoulos, 2017) presents this procedure in more details.

Message Hash in Signature

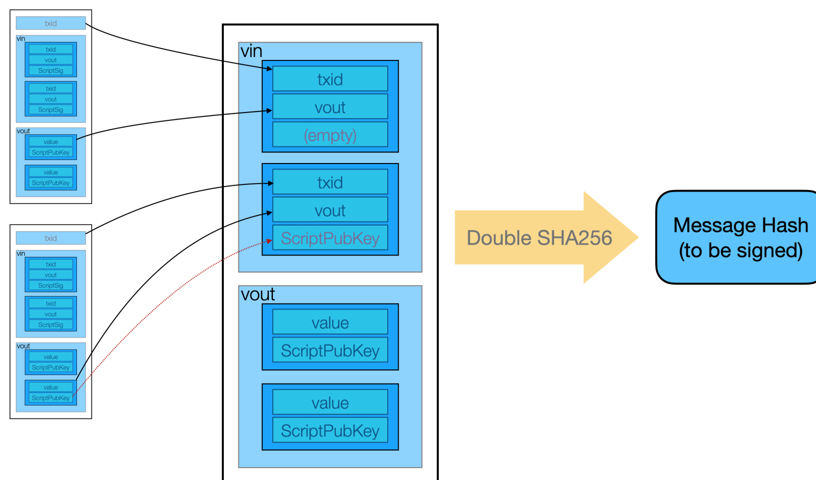


Figure 4.4.: Example of generating message hash to be signed (for the first input)

As it is possible to include several inputs in the `vin` list of a transaction, the messages to be signed are not the same for these inputs. At first, we create a transaction by filling all required fields for the inputs and outputs. However, regarding the field `scriptSig` of the inputs, these scripts are replaced by the corresponding `scriptPubKey` of the UTXOs they are referring to, since we do not have the signatures yet. Then, to generate a message hash for an input, which is an argument of the ECDSA signer, we remove the `scriptSig` of that input (it is replaced by `scriptPubKey` before), then serialise the transaction

and pass it through double SHA256 hash function. Figure 4.4 shows an example of generating message hash to be signed for the first input in the `vin` list.

After obtaining the signatures from ECDSA signer, we construct a `scriptSig` for each input with the signature and the public key corresponding to the private key used to sign as shown in the figure 4.2. The transaction is now ready to be broadcasted to the decentralized network.

5. Creating Ethereum Transaction

Ethereum is often described as the world computer. From a computer science's point of view, it is an unbounded state machine with a globally accessible singleton state and an Ethereum Virtual Machine (EVM) which maintains that state (A. Antonopoulos et al., 2018). Ethereum is also an open source, decentralised computing infrastructure and has the ability of executing programs called *smart contracts*. In Ethereum, blockchain is used to synchronise and store the changes of the state. Together with that, *ether* is the cryptocurrency used to meter and constrain resource costs of executions.

There are two types of account in Ethereum, including Externally Owned Account (EOA) and contract account. An EOA has a private key to control the access to funds or contracts. Unlike EOA, a contract account has smart contract code. In this work, we only consider the scenario of sending ether from an EOA to another EOA and ignore smart contracts. Section 5.1 gives an introduction to the blockchain and the world state in Ethereum. We analyse in detail the components of a transaction in section 5.2. Next, we show how to create a new Ethereum transaction in section 5.3. We also discuss the method of recovering a public key from a signature.

5.1. Blockchain and World State

Both Bitcoin and Ethereum have a blockchain recording full history of confirmed transactions. Nevertheless, there exists some drawbacks in the architecture of storage in Bitcoin so that it is inconvenient and costly to retrieve information from the ledger. Ethereum, on the other hand, proposed using a chain of states to improve these inconveniences.

Ethereum maintains a world state containing up-to-date information related to each address used in the network. This world state is updated right after a new block is mined and appended to the ledger. Figure 5.1 gives us a visualisation for the chain of blocks and the chain of world states.

As we can see in the figure 5.2, each address in the world state links to an account state. There are four fields in an account state of an address as follows:

- nonce: this is a counter indicating the number of transactions sent from this address. This prevents the double-spending problem in Ethereum by ensuring that each transaction is only processed once.

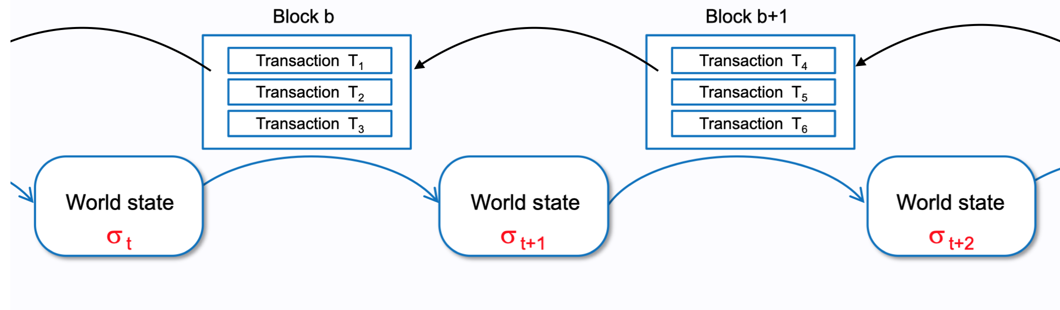


Figure 5.1.: Blockchain and World State in Ethereum

- balance: this is the number of ether owned by this address. In fact, the denomination *wei* is used to represent balance instead of ether. An ether equals to 10^{18} wei.
- code hash: this hash refers to the source code of the smart contract linked to this account. The smart contract is triggered and executed by calling to this hash. In the scope of this project, we only focus on externally owned accounts, and hence this field is empty.
- storage hash: this hash refers to the place storing the data of smart contract. An contract has to maintain its new state after being called by storing its local variables in the code. This field is set to be empty by default.

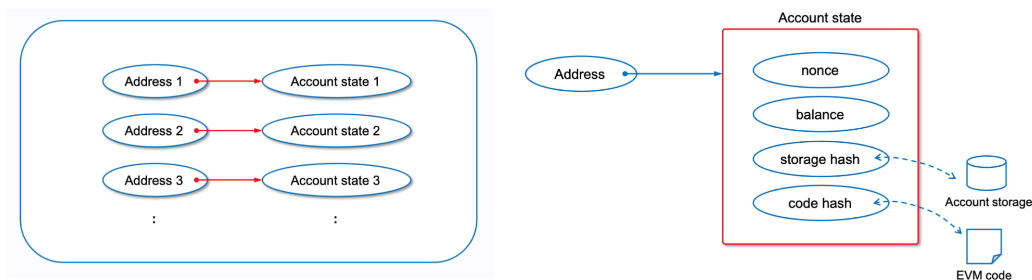


Figure 5.2.: Inside a world state (left) and an account state (right)

To know the balance of an address in Ethereum, we simply look up this address on the world state and obtain the information immediately. In the meantime, retrieving the balance of an address in Bitcoin roughly requires us to traverse all transactions stored in the blockchain and find the UTXOs of that address, then calculate the sum of these UTXOs. This is one of the reasons why there are many commercial APIs offering quick accesses to Bitcoin assets.

5.2. Transaction Components

Compared to Bitcoin, an Ethereum transaction does not allow to include many inputs and many outputs. It is also unnecessary to refer to other confirmed transaction to prove the ownership of the sender. Instead, it has exactly one address for the sender and one address for the receiver. Thanks to the world state, it is clear that how much coins the sender has at a certain moment, and hence more convenient to validate a transaction.

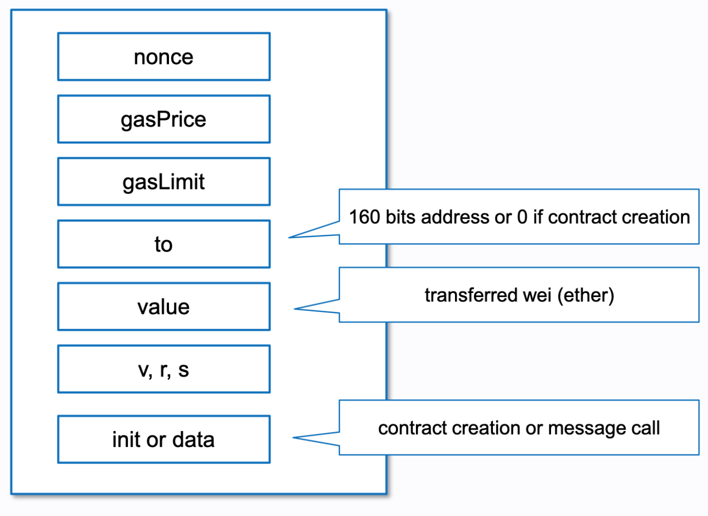


Figure 5.3.: Components of an Ethereum Transaction

An another difference in an Ethereum transaction compared to Bitcoin is the way of fee calculation. Regarding Bitcoin, size of the transaction is the main factor determining the fee which is based on an estimation of effort consumed by a miner. Ethereum, in a different way, uses gas as an unit to measure the computational effort required to execute operations on the network. Using gas as the fuel of Ethereum, on one hand, protects the system from the volatility which can be caused by the rapid change of ether's value. On the other hand, it manages the ratio between the costs of various resources used in the network such as memory, storage.

As depicted in the figure 5.3, an Ethereum transaction must include the following seven fields:

- `nonce`: this is one of the most important components defining the order of executing transactions. As briefly introduced in the account state, `nonce` is a counter and hence it is also the number of transactions sent from an address. Note that a transaction with `nonce` i is validated by a miner only when the transaction with `nonce` $i-1$ is already confirmed. Otherwise, it will be stuck and appended to the waiting list until the transaction $i-1$

is confirmed. If there are two transactions with the same `nonce` from an address, one of them will be confirmed and the another will be rejected depending on which one is validated by the miner first. Therefore, we should be careful with this field if we want to send some transactions in a row.

- `gasPrice`: this is the amount of wei per gas unit that an user is willing to pay for her transaction. The higher this amount is, the faster her transaction is likely to be confirmed.
- `gasLimit`: this is the maximum number of gas units that the sender is willing to buy in order to complete the transaction. If the receiver address is a smart contract, the `gasLimit` is needed to be estimated since in the program of the contract, there might exist conditions which lead to different execution paths. However, in the scope of this project, our target is just to send ether from one address to another. In this case, the `gasLimit` needed is fixed at 21,000 gas units.
- `to`: the destination's 160-bit address as presented in the previous chapter. This destination is set to be 160 bits of 0s if this is a smart contract creation transaction.
- `value`: number of wei that the sender wants to transfer to the receiver in this transaction.
- `signature`: besides the two parts r and s of an ECDSA signature, an Ethereum transaction has to include one more value, namely v, to indicate the the sign of y-coordinate of the elliptic curve point in which r is the x-coordinate.
- `data`: this is the place where we can put the code of a smart contract. In case a transaction does not relate to any smart contract, this field is set to be empty.

We notice that there is no address of the sender in a transaction. In fact, the sender's address is derived from the signature included in the transaction. This is the reason why an Ethereum transaction requires the value of v apart from r and s in the signature. The detail of this derivation is presented in the section 5.4.

5.3. Create a new transaction

We have to provide all required fields presented in the previous section to create a new transaction (see Appendix B for the serialisation of an Ethereum transaction). Regarding the `nonce`, it is necessary to get the up-to-date value of the counter corresponding to the address of the sender. This is done by requesting the nonce value stored on the world state. Moreover, we avoid the case of creating many transactions in a row in our application in order to prevent the problems of nonce duplication.

For the field of `gasPrice`, we use a third-party API[1] to get the recommended price at the moment of transaction creation. Since our wallet application only supports the normal transaction in which coins are transferred from one address to an another, the `gasLimit` is always fixed to be $21,000$ gas units. The transaction fee equals to the `gasPrice` multiplied with the `gasLimit` in this case. Note that in cases a transaction contains a smart contract, we have to set another value for the `gasLimit` and the transaction fee is equivalent to the product of the actual consumed gas units and the `gasPrice`. If the actual used gas is greater than the value of `gasLimit`, the miners will stop execute the transaction and throw an error. Otherwise, only the actual used gas units are charged in the fee.

The sender also provides the information about destination's address and the amount of wei she wants to transfer. The field of `data` is set to be empty since we do not use smart contracts. We then serialise this transaction and pass it through the Keccak256 hash function to get the message hash. This hash, together with the corresponding private key, are the two arguments of the ECDSA signer. Finally, the transaction now is ready to be broadcasted with the output v, r and s of the signer included.

5.4. Public Key Recovery

As mentioned earlier, an Ethereum transaction does not contain the address as well as the public key of the sender. It is because the public key can be computed directly from the signature, and then the address can be derived easily. Given r and s, we can compute two possible public keys since the x-coordinate gives us two possible points on the elliptic curve. To exactly determine which point is used, the transaction includes a prefix value v. Let us denote:

$$R = \begin{cases} (r, y) & \text{if } v \text{ is even} \\ (r, -y) & \text{if } v \text{ is odd} \end{cases}$$

Finally, the public key of the sender is recovered by the following equation:

$$K = r^{-1}(sR - zG)$$

where z is the message hash and G is the base point of the elliptic curve.

EIP55 (Buterin, 2016) describes in detail how to set the value of v properly. This value depends on the chain identifier of the network we are working on. For example, 1 is the main network, 3 is the ropsten netwok for testing purpose, etc. In particular,

$$v = \{0, 1\} + \text{CHAIN_ID} * 2 + 35$$

[1] See recommended gas price at https://ethgasstation.info/

where $\{0, 1\}$ is the parity of the y-coordinate of the curve point for which r is the x-coordinate. 35 stands for we hash nine Recursive Length Prefix (RLP) encoded elements as show in table B.5 of the Appendix B.

6. Wallet Implementation

In this chapter, we present in details the implementation of the wallet application. At first, we present the design pattern used to build the architecture of our application in the section 6.1. We then show the table in the local database storing information of addresses and tokens in the section 6.2. After that, we present the interactions with third-party APIs to get information about addresses from the decentralised network in the section 6.3. Next, we analyse the global workflow of our wallet application in the section 6.4. Finally, we showcase a typical and important usecase in the wallet, namely, Bitcoin transaction creation in the section 6.5.

6.1. Design Pattern: MVVM

Model-view-viewmodel (MVVM) (Anderson, 2012) is a software architectural pattern that facilitates the development by separating the graphic user interface (view) from the business logic (model). The viewmodel of MVVM is a value converter responsible for exposing data objects from model in such a way that objects are easily managed and presented. A viewmodel also handles all view's display logic and organises accesses to business logic layer. In our implementation, we use this design pattern to develop the wallet.

In the MVVM architecture, view and viewmodel communicate with each other to exchange data. A viewmodel also needs to interact with some repositories which provide data from database or from web services. We use RxJava (Davis, 2019) to handle these communications by events. By RxJava, it is easier to define a threading model operated with multi-threads. It also helps us manage tasks running on background threads and main thread. Briefly, the main components of RxJava include:

- Observable: is a class that emits a stream of data or events.
- Observer: it is a class which receives the events or data, then acts upon them.
- Scheduler: tells an observable and an observer which thread should be started. In short, it manages concurrency.

6.2. Database

A wallet must provide its users information about their accounts such as balance for each type of cryptocurrencies, exchange prices, etc. In background, our wallet manages many addresses and public keys in such a way that it is convenient to create and broadcast new transactions. To do those, we design a local database where store essential information about addresses, public keys and tokens of the wallet. Each table in the database is described as below.

Account Table

This table manages different accounts in the same wallet (see table 6.1). We use Google service to handle login functionality.

_id	xpub	account_id	coin
1	tpubDC3ELhn74GWbX152iNVn...	11740199...	0
...	...	...	...

Table 6.1.: Account table in database

- _id: a counter represents identifier for each record.
- xpub: account extended public keys of Bitcoin and Ethereum received from offline server. The Android app uses those xpubs to generate addresses.
- account_id: each Google account has an identifier. A Google account used in this app is mapped to a Bitcoin's xpub and an Ethereum's xpub. These two xpubs are derived from a mnemonic code.
- coin: 0 is Bitcoin. 1 is Ethereum.

Address Table

This table stores information related to each address of an account (see table 6.2).

address	idx	pubkey	status	category	balance	unconfirmed	ntx	haskey
ms6b...	0	03995...	0	0	10000	0	3	1
...	...	...	...	...	...	...	...	...

Table 6.2.: Address table in database

- address: string of an address in the tree (see figure 3.9).
- idx: index of the address in the tree structure of BIP-44.
- pubkey: public key of this address.

- status: 0 indicates an used address, 1 means a being used address, and 2 means an unused address (see figure 3.10).
- category: 0 means this is a receiving address, 1 stands for a change address (see figure 3.10).
- balance: number of satoshi (1 BTC = 10^8 satoshi) for a Bitcoin address, and number of wei (1 ETH = 10^{18} wei) for an Ethereum address.
- unconfirmed: this is the coming coins in transactions which are waiting for a miner to confirm.
- ntx: number of confirmed transactions involving this address
- haskey: 0 means this address does not have its token stored in the wallet application. Otherwise, it is 1 and it is taken into account in the spendable amount.

Token Table

This table stores tokens loaded from the server (see table 6.3).

address	prvkey
ms6bM4UR4G...	a58672b185fe9c...
...	...

Table 6.3.: Token table in database

- address: string of an address in the tree (see figure 3.9).
- prvkey: the token corresponding to this address and loaded from the server.

6.3. API Usages

In our wallet, we use third-party APIs to get information about addresses from decentralised networks. To connect to those APIs, we create HTTP requests and send calls to the third-party servers, then receive responses and extract information from them. There are two types of a HTTP request: GET and POST.

- GET: to request data from the server. In this project, we use GET requests to get information about balance and UTXOs of addresses. A request of this type is represented clearly in the URL. For example, listing 6.1 shows the implementation of GET request used to retrieve information about an address. We concatenate the address to the URL (line 4) and execute it (line 11). The server then returns a JSON object containing information about the required address. We need to parse this information (line 13) to extract information necessary for our process.

- POST: to submit data to be processed to the server. In this project, we use POST requests to submit raw transactions. Listing 6.2 shows an example of submitting a raw transaction by a POST request. We need to attach the transaction with a suitable format (lines 6, 10) to the request. Once receiving this request, the server checks if the transaction is in a valid format, then broadcasts it into the decentralised network.

In our implementation, we use some APIs from different providers. However, the usages of these API are done by GET and POST requests similar to listing 6.1 and listing 6.2. For more details about these APIs, see the appendix ??.

```
public static Address requestBtcAddressInfo(String addressBase58Check){
    StringBuilder url = new StringBuilder();
    url.append(Constants.BTC_ADDRESS_INFO_BLOCKCYPHER);
    url.append(addressBase58Check);
    url.append("?includeScript=true");

    OkHttpClient client = new OkHttpClient();
    Request request = new Request.Builder()
            .url(url.toString()).get().build();

    Response response = client.newCall(request).execute();
    JSONObject json = new JSONObject(response.body().string());
    return Mapper.fromJsonBlockCypherToAddress(json);
}
```

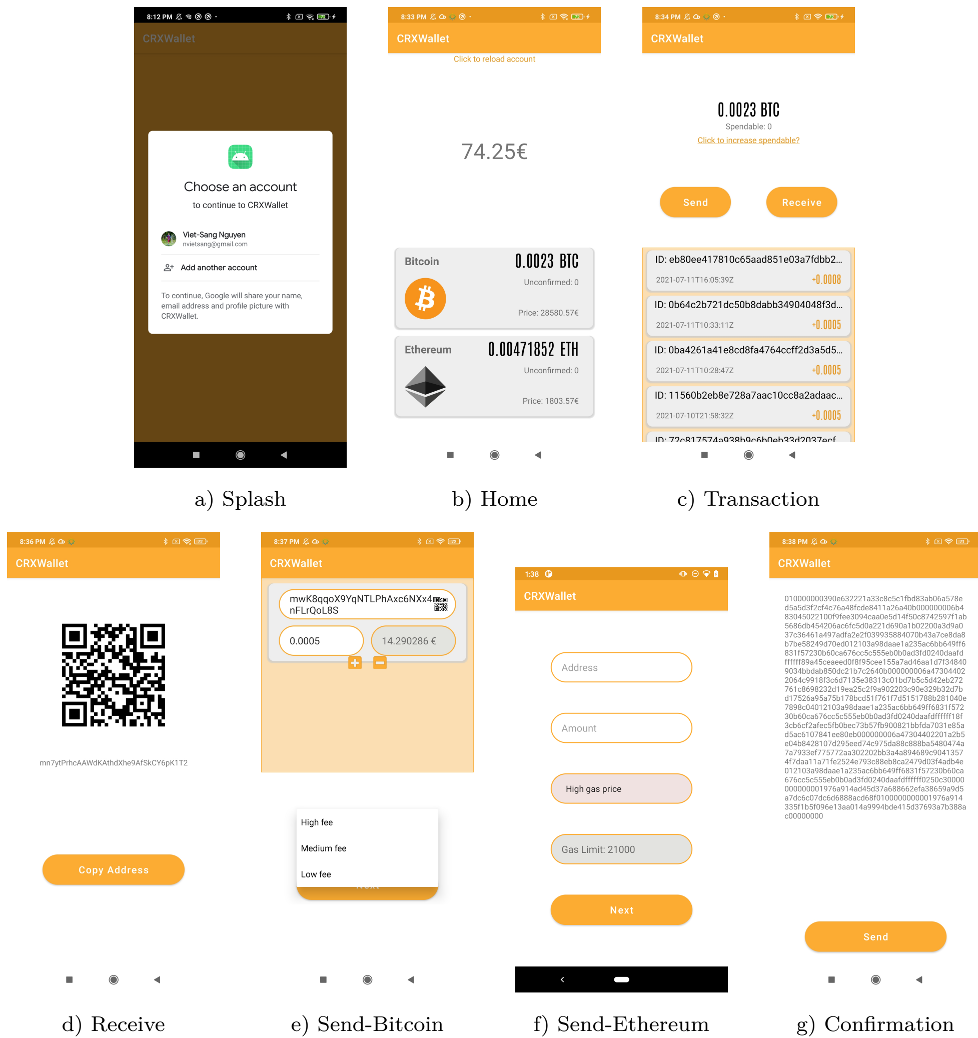
Listing 6.1: Example of GET request

```
public static String requestBtcToBroadcastTransaction(String rawTx){
    StringBuilder url = new StringBuilder();
    url.append(Constants.BTC_BROADCAST_CHAINSO);

    JSONObject data = new JSONObject();
    data.put("tx_hex", rawTx);

    OkHttpClient client = new OkHttpClient();
    MediaType type = MediaType.parse("application/json");
    RequestBody body = RequestBody.create(type, data.toString());

    Request request = new Request.Builder()
            .url(url.toString()).post(body).build();

    Response response = client.newCall(request).execute();
    JSONObject json = new JSONObject(response.body().string());

    if (json.has("status")) {
        if (json.getString("status").equals("fail")){
            return null;
        }
    }

    return json.getJSONObject("data").getString("txid");
```

```
25 }
```

Listing 6.2: Example of POST request

6.4. Workflow of Wallet Application



a) Splash b) Home c) Transaction

d) Receive e) Send-Bitcoin f) Send-Ethereum g) Confirmation

Figure 6.1.: Screenshots of the wallet application

Splash Activity

1. Right after the user opens the wallet application, it shows this activity with the login button.
2. If the user logins successfully, it will check in the account table of the database if the Google identifier of this user exists.

- If yes, it will load the information about this account from the database and move to home activity.
- If no, the application will as the user to connect to the token server in order to create a new account and move to home activity.

Home Activity

1. Get the real-time exchange price for Bitcoin and Ethereum by periodically (2 minutes) requesting with APIs.
2. Show the information about Bitcoin and Ethereum including balance, unconfirmed amount, logo, etc.
3. Check incoming coins for Bitcoin periodically (2 minutes) for some types of addresses as below with the APIs. Ideally, we have to observe incoming coins on all addresses belonging to this wallet. However, it consumes many requests and leads to reach the limitation of the API providers. Therefore, we only observe on the addresses which has high probabilities to receive or sent out coins:
    - a receiving address
    - a change address
    - addresses with tokens available on the wallet.
4. Filter new transactions in the Ethereum network to take and process transactions related to this wallet. Unlike Bitcoin, we can use a library for fee to observe real-time new transactions. Then, we have to check if one of our address relates to any transaction. If yes, we take that transaction and parse information from it.
5. If the user clicks on a coin (Bitcoin or Ethereum), it moves to the transaction activity.

Transaction Activity

1. Load transaction history.
2. If the user clicks on “click to increase spendable?”, the application will show a list of addresses with positive balance for the user to choose. Then, if the user connects the application with the token server and clicks the button “Update”, it will load tokens for chosen addresses and store in the database. If the token server is close, it will ask the users to establish the connection.
3. If the user clicks on the button “Receive”, it moves to the receiving activity.
4. If the user clicks on the button “Send”, it moves to the corresponding sending activity.

Receive Activity

1. Get the lowest-index address in the receiving branch (see figure 3.9).
2. Show the QR code and the address string on the screen.
3. If the user clicks on the button "Copy Address", it will copy the address to clipboard.

Send-Bitcoin Activity

1. Show placeholders for the user to input information of receivers, the user can specify more than one output address and the sending amount by the buttons "plus" and "minus".
2. The user can choose the level of fee.
3. If the user clicks on the button "Next", it will check if the total amount that the user wants to send is smaller than the spendable amount.
    - If yes, it will go to the database and select sufficient UTXOs, then create and sign a new transaction, then move to the confirmation activity.
    - If no, it shows an error notification on the screen.

Send-Ethereum Activity

1. Show placeholder for the user to input information of a receiver. Note that unlike Bitcoin, an Ethereum transaction can only have one input address and one output address.
2. The user can choose the level of fee.
3. If the user clicks on the button "Next", it will check if the total amount that the user wants to send is smaller than the spendable amount.
    - If yes, it will go to the database and select an address as the input, then create and sign a new transaction, then move to the confirmation activity.
    - If no, it shows an error notification on the screen.

Confirmation Activity

1. Show the raw transaction to be broadcasted.
2. This activity is planned to show information decoded from the raw transaction for the user to verify before actually broadcasting it.
3. If the user clicks on the button "Send", it will create a POST request to broadcast the raw transaction.

6.5. Usecase: Bitcoin Transaction Creation

In this section, we consider a typical usecase in the wallet application, namely, creating a new Bitcoin transaction. After filling necessary information about recipient(s), the user clicks the button “Next” on the send-bitcoin activity (see figure 6.1e). The clicking action triggers the function `onClickSendBitcoin()` in the viewmodel corresponding to this activity. Listing 6.3 presents the execution after the application receives the signal of creating a new transaction from the user. It first checks if any address is invalid (lines 6-10), then checks if the spendable amount is affordable (lines 13-17). If those conditions are satisfied, it formalises the outputs of the transaction (lines 19-21) and finally triggers the event of transaction creation (lines 24-31).

```
public void onClickSendBitcoin(List<BTCReceiver> listBTCReceivers){
    listInputs.clear();
    listOutputs.clear();

    listOutputs = new ArrayList<>();

    for (BTCReceiver BTCReceiver : listBTCReceivers){
        if (!BTCReceiver.isValidReceiver()){
            Toast.makeText(context, "Invalid address or amount").show();
            return;
        }

        totalSentAmount = totalSentAmount.add(BTCReceiver.getAmount());
        if (!coin.isEnoughToPay(totalSentAmount)){
            Toast.makeText(context, "Not enough spendable coin").show();
            return;
        }

        listOutputs.add(
            new Vout(BTCReceiver.getAddress(), BTCReceiver.getAmount())
        );
    }

    transactionUsecase.execute(
        new SelectBtcSenderObserver(),
        Event.SELECT_BTC_SENDER_ADDRESS,
        coin.getId(),
        totalSentAmount,
        listOutputs.size()+1,
        coin.getListFees().get(feeTarget)
    );
}
```

Listing 6.3: Trigger to create a new Bitcoin transaction

Once the event is triggered, it selects some UTXOs of addresses from database for the inputs of the transaction (see listing 6.4). Since the Bitcoin transaction fee depends on the size of the transaction and we do not know exactly this

size until the transaction is completely created. It means that when we add one UTXO to the inputs, the transaction fee will be increased. However, it is insufficient to allow the fee uncontrolled increases as we have to consider the our spendable amount as an upper bound (lines 13, 25-31). If the spendable amount allows, the selection function returns a list of UTXOs that we can use as the inputs of the transaction (lines 17-24, 30).

```
1  public Single<List<Vin>> selectBtcSender(String tableId, BigInteger
       sentAmount, int nOutputs, int fee) {
2      return Single.create(emitter -> {
3          List<Vin> listInputs = new ArrayList<>();
4          BigInteger spendingAmount = BigInteger.ZERO;
5          int nInputs = 0;
6          int sizeInputs;
7          int sizeOutputs = nOutputs* SIZE_VOUT;
8          BigInteger estimatedFee = BigInteger.ZERO;
9
10         List<Address> listUsingAddress = database.loadUsingHasKeyAddress(
       tableId);
11         for (Address addr: listUsingAddress){
12             List<UTXO> listUTXO =  Wrapper.requestUtxoBtc(addr.
       getAddressString());
13             spendingAmount = spendingAmount.add(addr.getBalance());
14
15             String token =
16                 database.loadToken(tableId, addr.getAddressString());
17             for (int i=0; i<listUTXO.size(); i++){
18                 listInputs.add(new Vin(
19                         listUTXO.get(i),
20                         addr.getPubkey(),
21                         addr.getAddressString(),
22                         token
23                 ));
24             }
25             nInputs = listUTXO.size();
26             sizeInputs = nInputs * SIZE_VIN;
27             estimatedFee = estimatedFee.add(BigInteger.valueOf((sizeInputs+
       sizeOutputs+EXTRA_SIZE) * fee));
28
29             if (spendingAmount.compareTo(sendingAmount.add(estimatedFee)) >=
        0){
30                 emitter.onSuccess(listInputs);
31             }
32             emitter.onError(new Throwable());
33         }
34
35     });
36 }
```

Listing 6.4: Select UTXOs for a new transaction

We notice that in the listing 6.3, we also specify the class `SelectBtcSenderObserver`

which processes the data returned by the input selection. As of this moment, we know exactly the size of the transaction, hence the exact transaction fee can be calculated (see listing 6.5, lines 10-12). Then, we create a raw transaction by serialising the fields in each input and each output (see line 17 and appendix B). From the raw transaction, we derive the corresponding messages used to sign by ECDSA (line 18). We then sign each message for each input and serialise them again to have the final raw transaction which can be broadcasted (lines 20-28).

```
1  class SelectBtcSenderObserver extends DisposableSingleObserver<List<Vin>> {
2      @Override
3      public void onSuccess(@NonNull List<Vin> vins) {
4          listInputs = (List<Vin>) e.getData()[0];
5          BigInteger sumInputs = BigInteger.ZERO;
6          for (Vin vin: listInputs){
7              sumInputs = sumInputs.add(vin.getUtxo().getValue());
8          }
9  
10         BigInteger actualFee = (listInputs.size()*SIZE_VIN
11             + (listOutputs.size()+1)*SIZE_VOUT) * feeTarget;
12 
13         BigInteger changeAmount = sumInputs.subtract(totalSentAmount).
    subtract(actualFee);
14 
15         listOutputs.add(new Vout(coin.getChangeAddress(), changeAmount));
16 
17         TransactionBitcoin transactionBitcoin = new TransactionBitcoin(
    listInputs, listOutputs);
18         List<String> listMsgHash = transactionBitcoin.getListMsgHash();
19 
20         List<String> listSig = new ArrayList<>();
21         for (int i=0; i<listMsgHash.size(); i++){
22             String msgHash = listMsgHash[i];
23             Signature sig = Signer.sign(msgHash, listInputs[i].getToken());
24             String encodedSig = Signer.encodeSigDER(sig);
25             listSig.add(encodedSig);
26         }
27 
28         String rawTx = transactionBitcoin.getRawTransaction(listSig);
29 
30         // broadcast rawTx...
31     }
32 
33     @Override
34     public void onError(@NonNull Throwable e) {
35         Toast.makeText(context, "Error").show();
36     }
37 }
```

Listing 6.5: Create and sign a raw transaction

7. Secure Wallet Architecture

Figure 7.1 shows the overview of the secure cryptocurrency wallet architecture which we design and build in this project. In this architecture, the white-box ECDSA signature and the token generator are user-dependent and they are linked with each other in such a way that generated tokens can only be operated by the corresponding white-box implementation. To satisfy the user-dependent property, the generators require a password from the user to operate. Moreover, environmental fingerprint is also used to lock the tokens. This aims to prevent attackers from extracting the white-box ECDSA signature and reuse it. In short, without any of the three factors (associated white-box part, user's password and environmental fingerprint), it should be impossible to get information about the private key from a token.

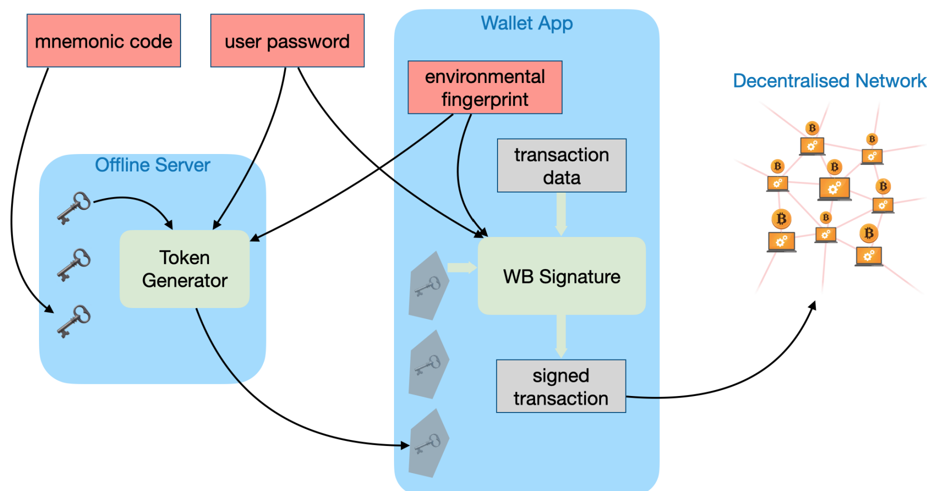


Figure 7.1.: Overview of the secure cryptocurrency wallet architecture

To establish the association between the token generator on the server and the white-box signature generator on the wallet application, we build a service which is responsible for generating a pair of them. To have a link between these two ones, the master private keys msk are the same and randomised in this establishment (see figures 7.2 and 7.4). Therefore, a token generated by a server can only be operated by its associated signature generator. This establishment is taken place at the initial phase when a user creates a new account on the wallet application.

7.1. Offline Server

An offline server is supposed to run on a trusted computer without connection to the internet since this server works with sensitive information such as the mnemonic code of the master seed. There are two types of requests from the wallet application that the server can receive: creating a new wallet account and updating tokens.

Request of creating a new wallet account

When receiving a request of this type, the server generates a completely new mnemonic code by the BIP-39 presented in the previous chapter. It then asks the user to securely keep this mnemonic code and sends a response to the wallet application with an attachment of account's extended public keys for coin types supported by the wallet. Specifically, the paths of an account's extended public keys for Bitcoin and Ethereum are as follows:

Bitcoin:	`M/44'/0'/0'`
Ethereum:	`M/44'/60'/0'`

From an account's extended public key, the wallet application is able to generate itself the addresses and public keys belonging to this account by the derivations discussed in the previous chapter.

Besides, this is also the initial phase where we need to generate the two associated white-box parts for the token generator and the signature. The association between these two parts is represented by the embedded key msk as shown in the figures 7.2 and 7.4 as mentioned earlier. This msk is randomized at the initial phase, and hence it is unique for each pair of token generator and white-box ECDSA signature.

Request of updating tokens

Token generator must guarantee that a generated token is a secure container for an ECDSA private key. Figure 7.2 shows a detailed design of the token generator part lying on a server. According to this design, a token is a ciphertext of an AES encryption with an ECDSA private key d as the plaintext. To establish an association with the white-box signature on the wallet application, we use a white-box implementation with the embedded key msk for this AES encryption. Note that the key d is derived from the tree-like structure from a mnemonic code and its BIP-44 path attached in the request from the wallet application. In our wallet, we use a message with the following format for the wallet to send to the server:

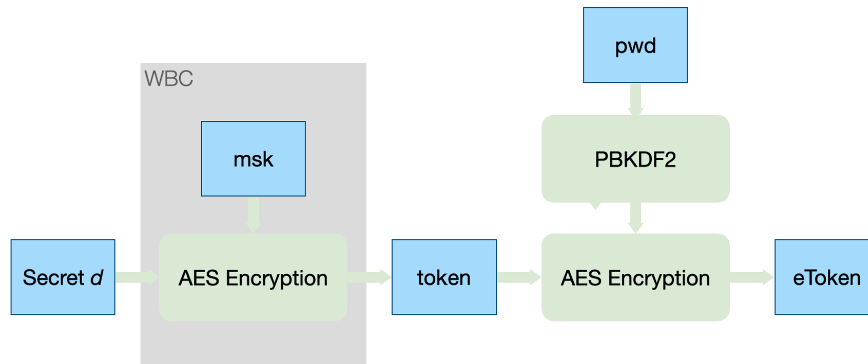


Figure 7.2.: Token generation with white-box cryptography on server

`<xpub>-<address_type>-<index>-<coin_type>`

When the server receives a request containing this message, it then extracts information and constructs a path in the tree structure to the target key of the request.

- `xpub`: the user is required to input his mnemonic code at the beginning to use this functionality. To ensure that the mnemonic code is correct, we compare the `xpub` derived from it with the one from the wallet application.
- `address_type`: indicates that we have to generate the key in the receiving branch or the change branch (see figure 3.9).
- `index`: indicates the index of the key in the tree structure.
- `coin_type`: indicates that the generated key belongs to Bitcoin or Ethereum.

To ensure the user-dependent property, we use the user's password to encrypt the token. This password is stretched by a password-based key derivation function PBKDF2 before being used as the key for AES encryption. The output of this encryption is an eToken which is used to transfer and store in the wallet application.

This server does not store any sensitive information about keys. It requires the user to input the mnemonic code of his account at the beginning of a working session. From this mnemonic code and the request sent by the wallet application with a path in the tree of BIP-44, it is able to generate the private key d corresponding to that path. In addition to the mnemonic code, the password pwd is also required when the server is started. Once the server finishes responding requests about tokens of the wallet application, it is closed without storing any information. The operation of this server is supposed to take place on a trusted computer, thus there is no malware which is able to capture the inputs from an user.

7.2. Secure Wallet Application

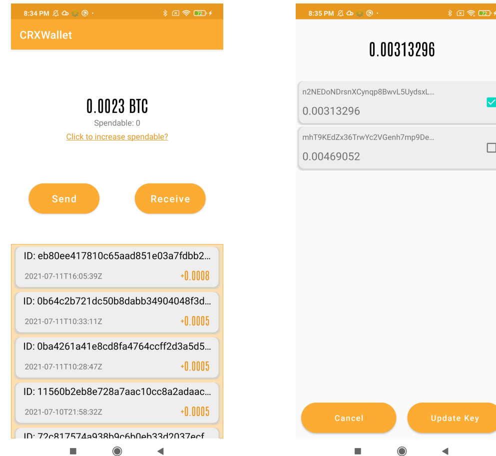


Figure 7.3.: Update tokens to increase spendable amount

Private key generation does not take place on the wallet application. Instead, the wallet stores some tokens loaded from its secure server. A user has the right to choose the number of tokens to be stored on the wallet application. The amount of coins contained in addresses whose private keys are available on the wallet application under the forms of tokens is called *spendable amount*, as we can generate the signatures by these tokens and spend the coins. The coins in addresses with positive balance without having corresponding tokens available on the wallet application are called *unspendable amount*. In order to spend coins in these addresses, we have to connect the application with the server and load the corresponding tokens. Figure 7.3 shows the screenshots of our wallet application where we can see the spendable amount (left) and choose some addresses with positive balance to load their tokens from the server (right).

The procedure of signing a transaction by its token is depicted in the figure 7.4. An eToken is a secure container of the private key provided by the server and stored on the wallet. From this eToken, we decrypt it to get the ECDSA private key d by the reversed flow of token generation presented in the previous section. It is first put into an AES decryption which takes the stretched pwd by the PBKDF2 as the key. This decryption gives us the token which is only encrypted by the white-box AES with the embedded key msk. Decrypting this token results us the ECDSA secret key d used to sign our transaction. The signer of ECDSA takes this secret key and the hash of the transaction as the inputs, then produces the expected signature. We notice that the white-box implementation covers both the AES decryption with the embedded key msk and the ECDSA signer. Doing so, we ensure that no information about the secret key d controlling the coins is leaked.

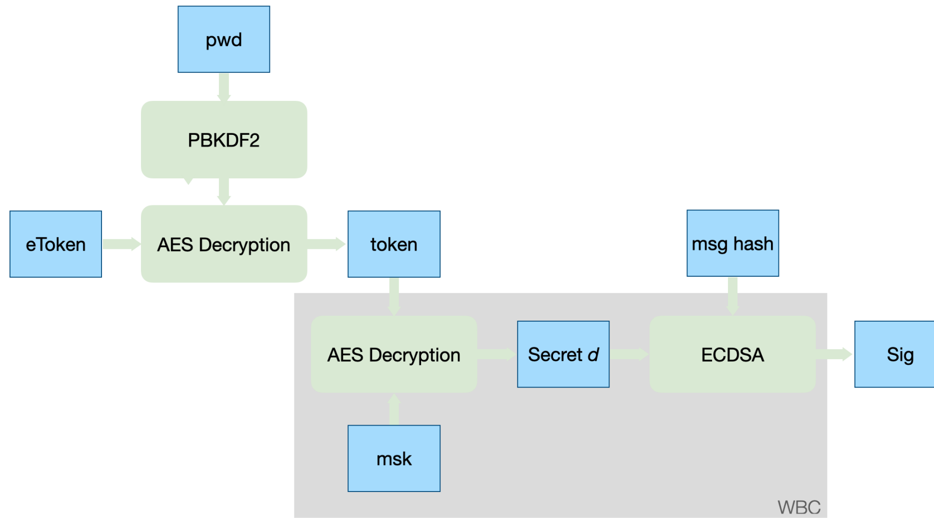


Figure 7.4.: Token decryption and signature with white-box cryptography on wallet app

7.3. Security Analysis

The wallet is deployed on smartphones in which a malicious attacker might control the entire execution environment. The main goal of our design is to provide a high protection for private keys used to sign transactions in the wallet application. As we can see in the figure 7.4, the ECDSA key is used only inside the white-box implementation. An adversary in a white-box attack context can see not only the inputs and the outputs of the implementation, but also intermediate computations happening along the way. She can also collect the addresses and values of accessed memory, or tamper with the implementation (e.g., injecting faults and altering the control flows). However, it is infeasible for her to extract the key from the implementation. Thus, white-box cryptography protects the key extraction from the ECDSA signing process.

We consider the case in which an attacker takes control of the smartphone where our wallet application is installed and tampers with the wallet implementation. In the storage of the wallet, there are only eTokens securely containing ECDSA keys. The attacker could not learn anything from an eToken without knowing pwd and msk. The pwd is user-dependent and environment-dependent as it contains the password from the user and the environmental fingerprint. Suppose that somehow the attacker knows this pwd, she can only decrypt an eToken to obtain a token still encrypted by msk in the white-box implementation. Unless she is able to break the white-box part, it is infeasible for her to extract msk from the implementation.

In case there exists a malware trying to steal useful information from our wallet, an attacker can only get tokens. Without the white-box implementation, she cannot achieve the goal of having the private keys controlling coins. In

this design, there is no useful information about the private keys outside the white-box part. The only way for an attacker to steal the private keys is to break the white-box implementation. However, a white-box implementation is supposed to be resistant against any possible attacks that an adversary can perform.

In case an attacker can take control over the wallet application and use it as the owner, she cannot steal all coins since there are a few tokens stored in the wallet. The number of tokens available in the wallet is chosen by the user (see figure 7.3, right). The user still keeps himself the mnemonic code and hence the coins controlled by other private keys (except the stolen ones) are still secure. There is a trade-off between how frequently the user wants to connect to the server to update tokens and how many tokens he wants to update in a connection. The more tokens the user updates in a connection with the server, the less number of connections it requires. However, he has to face more risks if an attacker can control his wallet.

Considering the server, it must be deployed on a trusted environment. Despite the fact that we try to eliminate storing sensitive information such as the user password and mnemonic code in the server, an attacker can get this information easily if she is able to attack or install malware on this server. To generate a token, the server requires the user to input his mnemonic code. In our design, we always avoid connecting this server to the internet.

8. Attacks and Countermeasures on ECDSA

An adversary in the white-box attack model can be seen as the owner of the device running the implementation. Hence, she is assumed to have full control over the execution environment. This enables her to perform physical attacks in order to extract the key. It implies that a white-box implementation should resist all existing and future side-channel attacks.

In this chapter, we first present the pseudo-code of the ECDSA as the algorithm 4. In both Bitcoin and Ethereum, they use the same elliptic curve with the same parameters defined in secp256k1 (Brown, 2010). We then analyse some vulnerable computations to side-channel attacks in this algorithm and discuss some typical attacks exploiting those weakness. For each attack presented in this chapter, we also discuss its countermeasures and apply to our implementation in the wallet application.

Algorithm 4: ECDSA signature generation

Input: message hash z, private key d, base point G, order of curve n
Output: signature pair (r, s)

1. Select a cryptographically secure random integer $k \in [1, n-1]$
2. Compute the curve point $(x_1, y_1) = k \times G$
3. Compute $r = x_1 \mod n$. If $r = 0$, go back to step 3.
4. Compute $s = k^{-1}(z + rd) \mod n$. If $s = 0$, go back to step 3.
5. **Return** (r, s)

8.1. Reusing nonce

It is crucial avoid reusing the nonce k for different signatures because it is easy to recover this nonce, and hence the private key d from step 4 in the algorithm 4. Suppose that two signatures (r_1, s_1) and (r_2, s_2) generated for the messages z_1 and z_2 are computed with the same unknown nonce k. The signal of reusing k is that the first halves of these two signatures are the same, $r_1 = r_2$. An attacker can recover the nonce by as follows:

$$k = \frac{z_1 - z_2}{s_1 - s_2}$$

Then, an attacker can easily calculate the private key:

$$d = \frac{s_1 k - z_1}{r_1}$$

Moreover, signing with biased nonces can cause some vulnerabilities. Slightly smaller nonces allow us to recover the private key easily (Moghimi et al., 2019). This bias is detected by measuring execution time.

Countermeasure

In the past, some failure implementations reusing k caused users of Bitcoin wallet on Android to lose their assets[1]. Nowadays, a nonce is deterministically generated in the signatures of cryptocurrency wallets from the private key and the message hash to avoid this vulnerability. This deterministic generation is defined by the standard algorithm in RFC 6979 (Pornin, 2013). Note that using alternative nonce generation should not be detectable. In case the nonce is deterministically generated, it should be indistinguishable from what a random and uniform generator can give. In the white-box context, it is impractical to use an external random number generator (RNG) since such an external RNG can be easily detected and disabled by the adversary.

8.2. Simple Power Analysis

This targets to exploit the vulnerability in the scalar point multiplication. A Simple Power Analysis (SPA) attack (P. Kocher et al., 1998) is based on the observation that power consumed by doubling operation is distinguishable to adding operation. In the case of using the double-and-add algorithm to perform an elliptic curve point multiplication, the bits of the nonce k is revealed by this method. Only one side-channel trace is enough to perform this attack.

Countermeasure

In order to be resistant against SPA, we prevent attackers from drawing a conclusion about the bit sequence of key based on power consumption by avoid branch instructions. Double-and-Add-Always method described in algorithm 5 or Montgomery Ladder method described in algorithm 3 can be used as a countermeasure against this attack.

[1] Android Security Vulnerability: https://bitcoin.org/en/alert/2013-08-11-android

Algorithm 5: Left-to-Right Double-and-Add-Always

Input: point P, secret $d = (d_{n-1}, ..., d_0)_2$
Output: $Q = dP$

```
1 Q[0] ← P
2 for i = n − 2 down to 0 do
3     Q[0] ← 2Q
4     Q[1] ← Q[0] + P
5     Q[0] ← Q[d_i]
6 end
7 Return Q[0]
```

8.3. Correlation Power Analysis

Correlation Power Analysis (CPA) was introduced by Brier, Clavier and Olivier (Brier et al., 2004) based on the well known Differential Power Analysis (P. C. Kocher et al., 1999). This attack uses the known plaintext and the guessed subkey to calculate the hypothesis power consumption. Then it correlates the actual power consumption and the hypothesis one by Pearson correlation coefficient and chooses the subkey giving the highest correlation.

Considering the manipulated data D and a reference state R, the power consumption is supposed to be linear with the Hamming distance $H(D \oplus R)$. Let W be the hypothesis power consumption, we have a model as follows:

$$W = \alpha H(D \oplus R) + \beta$$

Denote $\mathcal{C}$ as the set of power consumption curves. At the first step of the CPA attack, we perform t executions with the input data $m_1, ..., m_t$ and collect the corresponding curves $\mathcal{C} = \{C_1, ..., C_t\}$. Secondly, we calculate the intermediate value D_i by the target function with m_i and key hypothesis g. Then, we produce the set of t Hamming distances: $\mathcal{H} = \{H_1, ..., H_t\}$ where $H_i = H(D_i \oplus R)$. Finally, we calculate the estimated correlation factor:

$$\hat{\rho}_{\mathcal{C},\mathcal{H}} = \frac{cov(\mathcal{C},\mathcal{H})}{\sigma_{\mathcal{C}}\sigma_{\mathcal{H}}} = \frac{t\sum(C_iH_i) - \sum C_i \sum H_i}{\sqrt{t\sum C_i^2 - (\sum C_i)^2}\sqrt{t\sum H_i^2 - (\sum H_i)^2}} \tag{8.1}$$

The correlation factor ρ is maximum when the right guesses are computed. By this method, an attacker can gradually recover the secret.

Applying CPA attack on ECDSA was proposed by Amiel, Feix and Villegas (Amiel et al., 2007). It targets to exploit the vulnerability of the multiplication $r \times d \pmod n$ in the step 4 of the ECDSA signature generation algorithm 4. This attack aims to recover the secret d bit by bit. Algorithm 6 describes this attack in more details. Note that f in step 7 indicates the transformation of a number into Montgomery space if needed.

Algorithm 6: CPA on ECDSA signature

Input: t signatures $(r_1, s_1), ..., (r_t, s_t)$, set of power curves $\mathcal{C}$
Output: secret $d = (d_{n-1}d_{n-2}...d_1d_0)_2$

```
1  d ← 0
2  for j from n − 1 to 0 do
3  |   d̂ ← d + b^j
4  |   g_max = 0
5  |   ρ_max = 0
6  |   for g from 0 to 1 do
7  |   |   H = {H_1, ..., H_t} where H_i = H(f(d̂ × r_i) ⊕ R)
8  |   |   ρ_g = ρ_{Ĉ,H}
9  |   |   if |ρ_g| > |ρ_max| then
10 |   |   |   g_max = g
11 |   |   |   ρ_max = ρ_g
12 |   |   end
13 |   end
14 end
15 Return d
```

Countermeasure

In order to be resistant against CPA, we can use a randomisation of the secret proposed by Coron (Coron, 1999). At first, we select a random number l and compute $d' = d + l \times n$, where n is the order of the elliptic curve. In practice, l can be a 20-bit number. Then, d' is used to multiply with r instead of d as before. This works since $r \times d' = r \times d \pmod{n}$. This countermeasure can prevent this attack since d' changes in each execution of the algorithm.

Since an external RNG can be easily detected and disabled by the powerful adversary in the white-box context, the randomness in a white-box implementation must be pseudorandomly generated from a single input plaintext. In other words, the pseudorandom number generator (PRNG) is seeded by the input plaintext. As stated in (Wang, 2020), there are some security properties that a PRNG should fulfill in the white-box setting:

- Pseudorandomness: the output of the PRNG should be difficult to distinguish from a true randomness.
- Obscurity: we must keep secret the design of the PRNG.
- Obfuscation: the PRNG should be mixed with other instructions of the white-box implementation so that it is difficult to distinguish the output of the PRNG from other intermediate values.

8.4. Differential Computation Analysis

Differential Computation Analysis (DCA) (Bos et al., 2016) is a counterpart of the DPA attack (P. C. Kocher et al., 1999). Instead of using the power consumption traces, DCA performs attack based on software execution traces which contain information about the memory addresses being accessed. We can consider DCA as a variant of DPA specialised for implementations in the white-box context. The steps of this attack are outlined as follows:

- **Step 1**: Trace a single execution of the program with an arbitrary plaintext and record all accessed address and data during the execution.
- **Step 2**: Visualise the trace obtained by step 1 to understand where the block cipher is being used and determine which cryptographic algorithm is implemented by counting the number of repetitive patterns.
- **Step 3**: Record multiple traces with random plaintexts, optionally limit the scope of the recording activity to the first round or last round.
- **Step 4**: Serialise values (usually bytes) in the traces recorded in step 3 into vectors of ones and zeros. This step is similar to the classical hardware setup in DPA, named memory transfers.
- **Step 5**: Use regular DPA tools to extract the key.

In WhibOx 2019 workshop, Vinet introduced the attack on ECDSA by applying DCA (Vinet, 2019). The objective is to recover the secret d based on the target operation $r \times d$ on the step 4 of the algorithm 4. In this attack, we guess byte by byte starting from the least significant one. As an example, the figure 8.1 shows the steps of DCA attack on ECDSA. This example supposes $r = (r_3 r_2 r_1 r_0)_8$ and $d = (d_3 d_2 d_1 d_0)_8$. At the first step (figure 8.1a), we guess d_0 and calculate the intermediate value $c_0 = r_0 \times d_0 \pmod{2^8}$. Then, we correlate c_0 with the traces to find the best candidates for d_0. In the next step (figure 8.1b), we guess d_1 and combine with the best candidates of d_0 to calculate the intermediate value $c_1 c_0 = r_1 r_0 \times d_1 d_0 \pmod{2^{16}}$, and so on.

$$
\begin{array}{cccccccc}
 & & & & r_3 & r_2 & r_1 & r_0 \\
 & & & \times & d_3 & d_2 & d_1 & d_0 \\
\hline
c_7 & c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0
\end{array}
\qquad
\begin{array}{cccccccc}
 & & & & r_3 & r_2 & r_1 & r_0 \\
 & & & \times & d_1 & d_2 & d_1 & d_0 \\
\hline
c_7 & c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0
\end{array}
$$

a) Guess d_0 and correlate 8 bits b) Guess d_1 and correlate 16 bits

Figure 8.1.: Recover key of ECDSA by DCA attack

Countermeasure

We notice that in attack targets on the multiplication $r \times d$ similar to the previous attack. Therefore, in order to be resistant against DCA, we can bind

the actual key d. As the countermeasure we have proposed for CPA, we can use a randomisation of the secret (Coron, 1999). By this method, d' is computed in the multiplication with r instead of d, and hence the vulnerability of this multiplication is avoided.

9. Conclusion and Future Work

In this internship, we have designed and developed a cryptocurrency wallet application on Android platform with the considerations of privacy and security. Our wallet supports two main functionalities with Bitcoin and Ethereum: (1) sending coins from an user's account to another account by creating new transactions and broadcasting them to the decentralised network, and (2) receiving coins sent by others to an user's account. This wallet also adapts the standards of Bitcoin and Ethereum communities to generate and manage the key structure. Moreover, we concentrated on solving the security problem in the most vulnerable parts of a cryptocurrency wallet: key storage and ECDSA signature. Our solution is to utilise white-box implementations for AES and ECDSA to protect secret keys on an open environment such as smartphones. The white-box implementations include the token generator running on a trusted server and the ECDSA signature used to sign transactions. A token securely containing an ECDSA secret key and generated by a server can only be operated by the corresponding white-box ECDSA signer. Besides, we also analysed some popular attacks on ECDSA and their countermeasures.

Future Work: Regarding the functionalities of our wallet application, it currently enables spending coins in the testing networks of Bitcoin and Ethereum, namely, Bitcoin Testnet3 and Ethereum Ropsten, respectively. To make it work in the main networks, we have to modify the fields indicating the type of the targeted network in the address and key constructions. Additionally, the wallet is relying on third-party APIs to interact with the decentralised networks. This limits the number of requests per hour and sometimes causes unexpected errors. Ideally, we can set up our application to become a node in the decentralised network. Doing so requires many complicated configurations.

Regarding the white-box cryptography, there does not exist an ideal white-box implementation in reality so far. In industry, people try to protect their white-box implementations by mixing different countermeasures and obfuscation techniques, hiding their designs and changing them frequently. We have studied some countermeasures to this aim, however, achieving good white-box implementations to integrate to this project is still a challenge.

Bibliography

Amiel, F., Feix, B., & Villegas, K. (2007). Power analysis for secret recovering and reverse engineering of public key algorithms. In C. Adams, A. Miri, & M. Wiener (Eds.), *Selected areas in cryptography* (pp. 110–125). Springer Berlin Heidelberg. (Cit. on p. 52).

Anderson, C. (2012). The model-view-viewmodel (mvvm) design pattern. *Pro business applications with silverlight 5* (pp. 461–499). Apress. https://doi.org/10.1007/978-1-4302-3501-9_13. (Cit. on p. 34)

Antonopoulos, A., Wood, G., & Wood, G. (2018). *Mastering ethereum: Building smart contracts and dapps*. O'Reilly Media, Incorporated. https://books.google.fr/books?id=SedSMQAACAAJ. (Cit. on p. 28)

Antonopoulos, A. M. (2017). *Mastering bitcoin: Programming the open blockchain* (2nd). O'Reilly Media, Inc. (Cit. on p. 26).

Bos, J. W., Hubain, C., Michiels, W., & Teuwen, P. (2016). Differential computation analysis: Hiding your white-box designs is not enough. In B. Gierlichs & A. Y. Poschmann (Eds.), *Cryptographic hardware and embedded systems – ches 2016* (pp. 215–236). Springer Berlin Heidelberg. (Cit. on p. 54).

Brier, E., Clavier, C., & Olivier, F. (2004). Correlation power analysis with a leakage model. In M. Joye & J.-J. Quisquater (Eds.), *Cryptographic hardware and embedded systems - ches 2004* (pp. 16–29). Springer Berlin Heidelberg. (Cit. on p. 52).

Brown, D. R. L. (2010). Sec 2. standards for efficient cryptography group: Recommended elliptic curve domain parameters. https://www.secg.org/sec2-v2.pdf. (Cit. on pp. 6, 50)

Buterin, V. (2016). Simple replay attack protection [https://github.com/ethereum/EIPs/blob/master/EIPS/eip-155.md]. (Cit. on p. 32).

Buterin, V., & de Sande, A. V. (2020). Mixed-case checksum address encoding [https://github.com/ethereum/EIPs/blob/master/EIPS/eip-55.md]. (Cit. on p. 12).

Coron, J. (1999). Resistance against differential power analysis for elliptic curve cryptosystems. *CHES* (cit. on pp. 53, 55).

Davis, A. L. (2019). Rxjava. *Reactive streams in java: Concurrency with rxjava, reactor, and akka streams* (pp. 25–39). Apress. https://doi.org/10.1007/978-1-4842-4176-9_4. (Cit. on p. 34)

Hur, J. B., & Shamsi, J. A. (2017). A survey on security issues, vulnerabilities and attacks in android based smartphone. *2017 International Conference*

on Information and Communication Technologies (ICICT), 40–46. https://doi.org/10.1109/ICICT.2017.8320163 (cit. on p. 2)

Johnson, D., Menezes, A., & Vanstone, S. (2001). The elliptic curve digital signature algorithm (ecdsa). *Int. J. Inf. Secur.*, *1*(1), 36–63. https://doi.org/10.1007/s102070100002 (cit. on p. 1)

Kocher, P., Jaffe, J., & Jun, B. (1998). Introduction to differential power analysis and related attacks [http://www.cryptography.com/dpa/technical] (cit. on pp. 2, 51).

Kocher, P. C., Jaffe, J., & Jun, B. (1999). Differential power analysis. *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, 388–397 (cit. on pp. 2, 52, 54).

Menezes, A. (1992). *Elliptic curve cryptosystems* (Doctoral dissertation) [UMI Order No. GAXNN-75771]. CAN, University of Waterloo. (Cit. on p. 6).

Moghimi, D., Sunar, B., Eisenbarth, T., & Heninger, N. (2019). Tpm-fail: Tpm meets timing and lattice attacks. (Cit. on p. 51).

Nakamoto, S. (2009). Bitcoin: A peer-to-peer electronic cash system. http://www.bitcoin.org/bitcoin.pdf. (Cit. on pp. 2, 24)

Palatinus, M., & Rusnak, P. (2014). Multi-account hierarchy for deterministic wallets [https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki]. (Cit. on p. 14).

Palatinus, M., Rusnak, P., Voisine, A., & Bowe, S. (2013). Mnemonic code for generating deterministic keys [https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki]. (Cit. on p. 14).

Pornin, T. (2013). Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). https://doi.org/10.17487/RFC6979. (Cit. on p. 51)

Vinet, G. (2019). Optimize binary tracing: Example with ecdsa implemenation. https://www.cryptoexperts.com/whibox2019/slides-whibox2019/Guillaume_Vinet-optimize_trace.pdf. (Cit. on p. 54)

Wang, J. (2020). *On the practical security of white-box cryptography* (Theses). University of Luxembourg ; Université Paris 8. https://tel.archives-ouvertes.fr/tel-02953586. (Cit. on p. 53)

Wuille, P. (2013). Hierarchical deterministic wallets [https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki]. (Cit. on p. 14).

Appendix A.

APIs used in the project

??

A.1. Bitcoin

- Transaction fee (GET):

```
https://bitcoinfees.earn.com/api/v1/fees/recommended
```

- Address information (GET):

```
https://api.blockcypher.com/v1/btc/test3/addrs/<address>
https://api.blockchair.com/bitcoin/testnet/dashboards/
    address/<address>
```

- Address UTXO (GET):

```
https://api.blockcypher.com/v1/btc/test3/addrs/<address>?
    includeScript=true&unspentOnly=true
https://chain.so/api/v2/get_tx_unspent/BTCTEST/<address>
```

- Broadcast transaction (POST):

```
https://api.blockcypher.com/v1/btc/test3/txs/push
{
    "tx": "<transaction>"
}
https://chain.so/api/v2/send_tx/BTCTEST/
{
    "tx_hex": "<transaction>"
}
```

A.2. Ethereum

- Transaction fee (GET):

```
https://ethgasstation.info/api/ethgasAPI.json?api-key=<key
    token>
```

- Address Information (GET):

```
https://api-ropsten.etherscan.io/api?module=account&action=txlist&address=<address>&startblock=0&endblock=99999999&sort=asc&apikey=<apikey>
```

- Broadcast transaction (POST):

```
https://ropsten.infura.io/v3/<key token>
{
    "jsonrpc": "2.0",
    "method": "eth_sendRawTransaction",
    "params": [<transaction>]
    "id": 1
}
```

A.3. Exchange Price

```
https://min-api.cryptocompare.com/data/price?fsym=BTC&tsyms=EUR
https://min-api.cryptocompare.com/data/price?fsym=ETH&tsyms=EUR
```

Appendix B. Serialisation

B.1. Extended Key

The serialisation of an extended key has to consist of the fields shown in table B.1.

No.	Field	Length (byte)	Description
1	version	4	This field indicates the type of networks. For instance, `0x0488b21e` is for public keys on Bitcoin mainnet.
2	depth	1	This field indicates the depth of the current node in the tree structure BIP-44
3	fingerprint	4	This is four bytes checksum taken from the hash of the parent key.
4	chaincode	32	This is 32 bytes of chaincode
5	key	33	The first byte is `0x00` if this is an extended private key. For an extended public key, it can either `0x02` if its y-coordinate is even or `0x03` if its y-coordinate is odd.
6	checksum	4	The byte array of five fields above is put into a double-hash SHA256 function. Then we check the first four bytes of the result as the checksum.

Table B.1.: Serialisation of an extended key

B.2. Bitcoin Transaction

The serialisation of a Bitcoin transaction has to include the fields shown in table B.2. Note that a variable length is calculated by the implementation (Java) in the listing B.1.

No.	Field	Length (byte)	Description
1	version	4	Version 1 (constant)
2	`vin` size	variable	Calculated by listing B.1
3	`vin` list	`vin` size	See table B.3
4	`vout` size	variable	Calculated by listing B.1
5	`vout` list	`vout` size	See table B.4
6	locktime	4	Set to be 0 (deprecated)

Table B.2.: Serialisation of a Bitcoin transaction

No.	Field	Length (byte)	Description
1	txid	32	Transaction identifier
2	vout	4	Position of the UTXO in the `vout` list
3	`scriptSig` size	variable	Calculated by listing B.1
4	`scriptSig`	`scriptSig` size	This consists of the signature and the public key of a sender
5	sequence	4	`fdffffff` (constant, deprecated)

Table B.3.: Serialisation of an input of `vin` list

No.	Field	Length (byte)	Description
1	value	8	Number of satoshi
2	`scriptPubKey` size	variable	Calculated by listing B.1
3	`scriptPubKey`	`scriptPubKey` size	This consists of some opcodes and the public key hash of a receiver

Table B.4.: Serialization of an output of `vout` list

```java
public static byte[] fromBigIntToBytesReverse(BigInteger num, int size){
    byte[] bytes = new byte[size];
    for (int i=0; i<size; i++){
        bytes[i] = (byte) num.mod(B256).intValue();
        num = num.divide(B256);
    }
    return bytes;
}

public byte[] numToVarInt(BigInteger num){
    if (num.compareTo(new BigInteger("253")) < 0){
        byte[] res = new byte[1];
        res[0] = (byte) num.intValue();
        return res;
    }
```

```
16      if (num.compareTo(new BigInteger("65536")) < 0) {
17          byte[] res = new byte[3];
18          res[0] = (byte) 253;
19          System.arraycopy(fromBigIntToBytesReverse(num, 2), 0, res, 1, 2);
20          return res;
21      }
22      if (num.compareTo(new BigInteger("4294967296")) < 0){
23          byte[] res = new byte[5];
24          res[0] = (byte) 254;
25          System.arraycopy(fromBigIntToBytesReverse(num, 4), 0, res, 1, 4);
26          return res;
27      }
28      else {
29          byte[] res = new byte[9];
30          res[0] = (byte) 255;
31          System.arraycopy(fromBigIntToBytesReverse(num, 8), 0, res, 1, 8);
32          return res;
33      }
34  }
```

Listing B.1: Convert a number to a variable-length byte array

B.3. Ethereum Transaction

The serialisation of a Bitcoin transaction has to include the fields shown in table B.5. Note that Ethereum uses Recursive Length Prefix (RLP) to encode data. The implementation (Java) of this encoding algorithm is showns in listing B.2.

No.	Field	Length (byte)	Description
1	nonce	RLP	Calculated by listing B.2
2	gasPrice	RLP	Calculated by listing B.2
3	gasLimit	RLP	Calculated by listing B.2
4	to	RLP	Calculated by listing B.2
5	value	RLP	Calculated by listing B.2
6	data	RLP	Calculated by listing B.2
7	v	RLP	Calculated by listing B.2
8	r	RLP	Calculated by listing B.2
9	s	RLP	Calculated by listing B.2

Table B.5.: Serialisation of a Ethereum transaction

```
1 private static final int EMPTY_MARK = 128;
2 private static final int TINY_SIZE = 55;
3 private static final int BASE_PREFIX = EMPTY_MARK + TINY_SIZE + 1;
4
5 public static byte[] rlp_encode(byte[] bytes) {
```

```
6       if (bytes == null || bytes.length == 0)
7           return emptyEncoding;
8
9       if (bytes.length == 1 && (bytes[0] & 0x80) == 0)
10          return bytes;
11
12      if (bytes.length <= TINY_SIZE) {
13          byte[] encoded = new byte[1 + bytes.length];
14          encoded[0] = (byte)(EMPTY_MARK + bytes.length);
15          System.arraycopy(bytes, 0, encoded, 1, bytes.length);
16          return encoded;
17      }
18
19      byte[] blength = lengthToBytes(bytes.length);
20      byte[] encoded = new byte[1 + blength.length + bytes.length];
21      encoded[0] = (byte) (BASE_PREFIX + blength.length - 1);
22      System.arraycopy(blength, 0, encoded, 1, blength.length);
23      System.arraycopy(bytes, 0, encoded, 1 + blength.length, bytes.length);
24
25      return encoded;
26  }
```

Listing B.2: RLP encoding implementation