Bitcoin & Ethereum Cross-chain Atomic Swap

A Trustless Method of Exchanging Bitcoin For Ether Between Two Peers

Bachelor's Thesis submitted to the

Computer Science Engineering, Software and Multimedia developement orientation of the *Haute Ecole Arc Ingénierie (HES-SO), Switzerland*

in partial fulfillment of the requirements for the degree of

Bachelor of Applied Science in Computer Science

presented by

Luca Srdjenovic

under the supervision of

Prof. Ninoslav Marina

co-supervised by

Thomas Shababi

July 2019

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Luca Srdjenovic
Neuchâtel, 26 July 2019

Abstract

Atomic swaps are practical for exchanging different cryptocurrencies in avoiding any trusted third-parties. This project shows a swap between Bitcoin and Ethereum blockchain using payment channels tools like hashlock or timelock. When the protocol is followed by both participants, it guarantees that the swap will either go ahead without any risk or be aborted. Furthermore, there is no scenario where someone can control both parties coins, therefore the entire swap process is non-custodial in nature.

Keywords : `Bitcoin, Ethereum, Atomic Swap`

Acknowledgements

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The main problem when two parties want to exchange some goods is the problem of trust. For dealing with an exchange that works, we need a certain level of trust must be applied particularly when the transaction depends on no one. To illustrate this, we cannot simply assure that a party A will get a good after paying some money to a party B and vice versa. A trusted third party or escrow may resolve this problem to arbitrate a transaction without no one try to swindle the other. But in compensation for this, an extra cost may be added to the transaction and there is still no guarantee that the escrow is a trusted source. Secondly, most of the crypto-based exchanges are still fully centralized even though the main goal of cryptocurrencies is to create a decentralized system of finance. When two parties would like to exchange two different coins, they must pass by crypto-fund commerce. However these platforms remain in generally exclusively of the main centralized exchanges. Anyone who uses these exchanges may potentially risk various additional problems like an inherent risk of being hacked or having a breach of user confidentiality, some delays, and many other problems. To figure these problems out, the necessity of implementation of cross-chain payments or atomic swap has been made for assuring a trustless and decentralized system that exists, authorizing the blockchain application to interact with another blockchain.

1.2 Purpose of the thesis

Our objective is to understand the conception of the cross-chain swap, to know how it works and to implement an example of an atomic swap with two actors exchanging two difference cryptocurrencies. The cryptocurrencies chosen for this project are Bitcoin and Ethereum both the most common blockchains. For making the success of the project, we need to master how both cryptographic currencies work if we want to program and develop software that implements them. The main goal of this thesis is not to have a product made for the production but instead create more educational work from scratch allowing the understanding of all the important concepts from the blockchain world.

1.3 Challenge

Programming on a blockchain for developing programs is not an easy task. Programming on two different blockchains at the same time is worse especially when we do not know about it. Every blockchain has its lot of features to take into account even they have also several similarities, which they deviate from the standard way of thinking in software development. The main goal here is to explain the atomic swap method that must be generic and therefore can be easily understandable and used by a large number of people. For doing this, we need to care about all the constraints and limit coming from each blockchain. The best approach to handle this challenge is to gather all the knowledge we can get and just start to code. Practices make perfect.

Chapter 2

Bitcoin, A Peer-to-Peer Payment System

Invented in 2008, Bitcoin is a digital currency that follows the ideas of mysterious pseudonymous developer software, Satoshi Nakamoto. Bitcoin is one of the first digital currency to use the peer-to-peer technology for executing payments with his currency called Bitcoin for goods. Bitcoin deviates from the traditional online payment system (e.g. bank system) and centralized to emerge a decentralized authority that operates it. Today bitcoin takes any such breadth to the level of economic, political and security point that we can define it like a distributed computing innovation.

2.1 Functioning of Bitcoin

The blockchain relies on a shared public ledger which includes all the confirmations of the transactions. The bitcoin wallets use these data for computing their spendable balance, thus new transactions may be verified and in this way ensuring that the spender owns them. All the integrity and the order from the beginning of the blockchain are secure with cryptography.

Another part of bitcoin is the distributed consensus system called mining that allows confirming pending transactions by including them into the blockchain. It enforces the chronological order of the blockchain protecting the neutrality of the Bitcoin network and enables different computers to agree and change the state of all the system. The modification of previous blocks is prevented by theses rules for any modifications and will conclude by the invalidation of all the subsequent blocks. It also prevents the fact to add easily new blocks successively to the blockchain. Therefore, no one can control what is included in the blockchain or modify parties of the blockchain to get back theirs spends. The transactions in the blockchain are irreversible.

2.2 Fees

Each time there is a transaction in bitcoin, the payer must include transaction fees. The fees are the reward for the miners and they are given to them after each confirmation of a

new block containing transactions. Transactions pay fees that are based on the total byte size also called `virtual size` of the signed transaction. That means for each byte of the transaction, we calculate the fees. The bytes is based on the demand for space in mined blocks they consume. In addition, we can choose a ratio that influences the transaction fee. If we give no transaction fees, our transaction will never be added to the blockchain and if we put to low fees, we have a chance that our transaction will never be mined or it will take an infinite time. This is because miners prefer to mine transactions that have higher fees.

2.3 Transaction

A transaction in bitcoin is the process to transfer coins between Bitcoin wallets all include in the blockchain. For each wallet, a secret data called private key is used to sign transactions providing a mathematical proof they are the owner of their wallet. A signature can also prevent the alteration of the transaction by another party. All transactions are broadcast to the Bitcoin network and their confirmations begin only within 10-20 minutes on average through a process called mining.

Figure 2.1. Illustrations of transactions in Bitcoin where each output wait as an Unspent TX Output (UTXO) until a later input spends it.
Source:`https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch02.asciidoc`

The figure 2.1 above shows the main part of a Bitcoin transaction. Each transaction must possess one input and one output. Each input spends a payment in satoshi[1]for a previous output. Each output then waits as an Unspent Transaction Output (UTXO) until an input spends it. A simple example, when our Bitcoin wallet tells us that we have a 10 bitcoins in our balance, that means in reality that we have 10 bitcoins waiting in one or

more UTXO.

To summarize, a transaction is composed of a set of inputs and outputs. Input is based on a `UTXO` at the address from the one who sends it. Output refers to an address where the funds are sent.

2.4 Bitcoin Script

As we have seen in chapter 2.3, a transaction has one input and one output. Input refers to a transaction identifier called `txid` and an output index number called `vout` for identifying a specific output to be spent. The output also has an amount in satoshi which it sends it.

A script is a list of instructions included in the transaction to explain how a person wants to spend the amount of satoshi. In a typical Bitcoin transaction, a person who satisfies the conditions of the script can gain access to it. The spender must provide a public key that then hashed can provide the target destination address. He also must give a signature to prove ownership of the private key corresponding to the public key just provided. The two next figures below are an example of how bitcoin script resembles with its system of the stack:

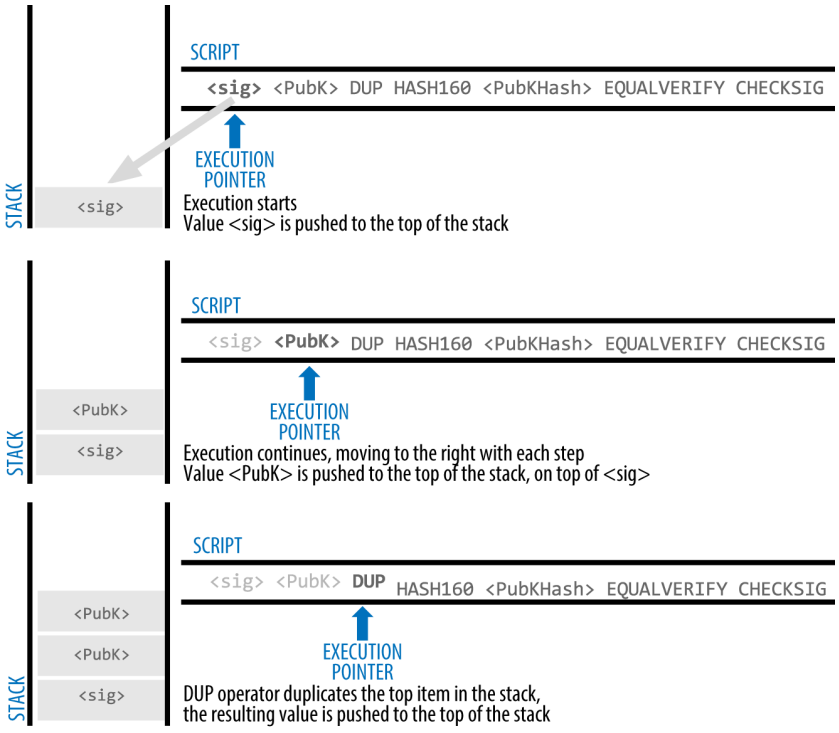

Figure 2.2. Example of a script for P2PKH transaction.
Source:`https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch06.asciidoc`

[1]Smallest unit in Bitcoin where 1 `satoshi` $= 10^{-8}$ `bitcoin`

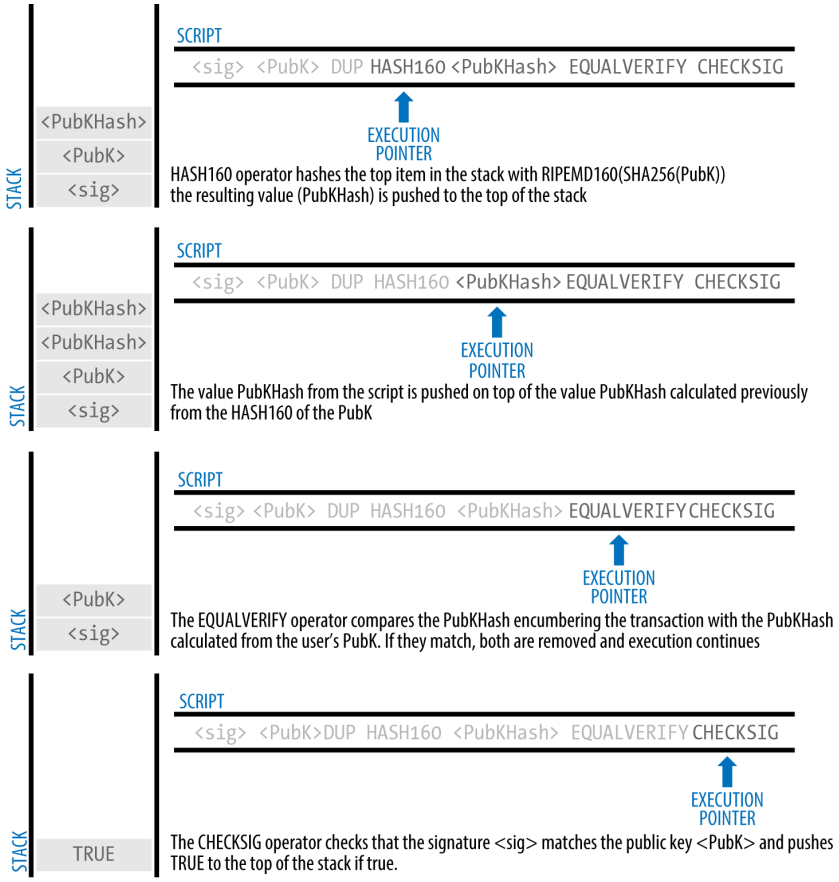

Figure 2.3. Next part of a script for P2PKH transaction.
Source:`https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch06.asciidoc`

2.5 P2PKH Script

P2PKH is the basic form of making a transaction and is the most common form of transaction on the Bitcoin network. Transaction paying Bitcoin to an address which it has P2PKH scripts are resolved only on sending the public key and a digital signature providing by the private key that corresponds. Theses data are concatenated into an unlocking script that needs to be evaluated for the script :

```
1       <Sig> <PubKey> OP_DUP
2       OP_HASH160 <PubkeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

Listing 2.1. An unlocking script for P2PKH script.

For checking if a transaction is valid, a signature script and public key script are executed

at the same time. The figure 2.2 and 2.3 shows the evaluation of a standard P2PKH pubkey script :

1. The signature is pushed to an empty stack. The public key is pushed on top of the signature.
2. The `OP_DUP` operation executes the script pubkey. `OP_DUP` pushes onto a copy of the data from the top of the stack.
3. `OP_HASH160`, pushes onto the stack a hash of the data currently on top of and creates a hash of the public key.
4. The pubkey hash is then pushed onto the top of the stack where now there are two copies.
5. `OP_EQUALVERIFY` is the equivalent of `OP_EQUAL` followed by `OP_VERIFY`. `OP_EQUAL` check the two values at the top of the stack and return true on the stack whether they are equal. `OP_VERIFY` check the boolean value on the top of the stack and terminates if the value is `FALSE`.
6. Finally, `OP_CHECKSIG` checks the signature provided against the now-authenticated public key provided. If the signature matches the public key and was generated using all of the data required to be signed, `OP_CHECKSIG` pushes the value `TRUE` onto the top of the stack.

A transaction to be valid must have `TRUE` value onto the top of the stack.

2.6 P2SH Script

Pay To Script Hash (P2SH) was standardized in Bitcoin Improvement Proposal (BIP) 16 and is the second common script the most used in the blockchain. It allows a transaction to be sent into a script hash instead of using the P2PKH script providing a public key hash. The receiver of the transaction must give a script that matches the script hash and the data inside for making the evaluation script to `true`. When this process is valid, the receiver can spend the amount of the output sent via `P2SH`. `P2SH` has the advantage to create various unusual ways for securing a transaction, e.g. adding a required password in additional of the signature into the script to unlock the transaction and spend it.

2.7 Segregated Witness

Segregated Witness (SegWit) is a BIP 141 that allows the modification of the transaction format of the cryptocurrency bitcoin. Segregated Witness means `consensus layer`. Its purpose is to prevent problems came from standard transaction structure as such bitcoin transaction malleability and fix them. The protocol allows optional data transmission and goes trough certain protocol restrictions. The protocol is described as an intention for reducing the blockchain size limitation problem that slows bitcoin transaction speed. It does this by separating the transaction into two segments, remove the unlocking signature data called `witness` from the original portion and adds it to a split structure in the end. The first section continues to keep the sender and receiver data and the new structure `witness` contains only scripts and signatures. The advantage of this technique is the size of the data segment doesn't change, however, witness data segment will have its size divided by four instead of keeping its real size in the standard protocol.

Chapter 3

Ethereum, A Decentralized Computing Platform

Launched in 2015, Ethereum is a decentralized software platform that authorizes the developers to build their smart contracts and Decentralized Application (DApp) and allows them to avoid any fraud, control or interference from a third party. The basis of Ethereum is to make a platform system using a Turing-complete programming language running in the blockchain instead of using the script language from Bitcoin. Developers can create distributed applications and publish them into the blockchain that runs on the Ethereum Virtual Machine (EVM). Ethereum work on the system of user accounts and balances in a manner called state transitions and doesn't use the UTXO from Bitcoin. We can define Ethereum as a programmable blockchain.

3.1 Ether

The currency refereed for making operation and transactions in Ethereum is the `ether`, which is the fundamental token of the blockchain. The smallest unit in Ethereum is the `wei` where 1 `wei` $= 10^{-18}$ `ethers` but it exists a multiple of units. The utility of wei comes from the representation of the data for the users who use it for paying the gas in transactions and smart contracts.

3.2 Gaz and Fees

Gas is a unit that measures the amount of computational effort, more precisely Gaz is used to calculate the number of fees that need to be paid into the network to execute an operation. Ethereum Gas is the mechanism of the Ethereum system, we cannot avoid that. A simple transaction or even smart contract that creates operations in the Blockchain cost Gas. The fees allow preventing any fissures in the system e.g. in introducing infinite loop into the code. The amount of fee to pay is defined by $GasLimit \cdot GasPrice$. The main goal of the fees is for rewarding the miners who mine transactions and then put them into blocks for securing the blockchain.

`Gas Limit` is the limit because there is a maximum amount of gas that we agree to spend on a transaction. This is here for avoiding e.g. if there is an error in the code and produce to overpay accidentally to much amount without our will.

`Gas Price`, is the amount that will increase or decrease the speed of the confirmation of the transaction by the miners in the blockchain. On spending fewer fees on a transaction, this won't be interesting for the miners, thus your transaction should take more time to be mined. On the opposite, higher is the number of fees, more the transaction will be mined quickly because it is more attractive for them.

3.3 Account

In Ethereum, it exists two types of account. The Regular account which represents the owned account and smart contract account. An account is often confused with an `address` but both terms have the same meaning. A regular account can be controlled by an external part of the blockchain (e.g. a user). An account to function needs to hold a private key for signing transactions that allows sending ether and a public key that allows receiving ether.

An account introduces the concept of `account state` that is defined by :

- **Nonce** : Represent simply the transaction count of an account.
- **Balance** : The ether balance owned by an account
- **Storage Root** : The storage contents of the account by default empty.
- **Code** : For contract accounts, this represents the code of the contract account stored.

3.4 Transaction

In Ethereum there are two distinct terms for defining a transaction, `Transaction` and `Message`. A `transaction` is a piece of data, signed by a regular account. It represents either a Message or a new Autonomous Object. Transactions are stored in each block of the blockchain. That's means that a transaction can be either a message or a new contract. A `message` is a piece of data and an amount of Ether that is transferred between two accounts. A message is created by contracts interacting with each other, or by a transaction from it. The main difference is that a regular account sends transactions and a contract account sends a message.

3.5 Ethereum Virtual Machine

EVM is the heart of the Ethereum mechanism. This machine is a Turing complete system provided by Ethereum and capable to execute codes into the Ethereum blockchain with an environment runtime for compiling smart contracts and execute them, altering the state of the blockchain. The code of the smart contract is compiled into Bytecode by the EVM compiler, executed and then managed in transactions initiated by accounts in the blockchain for integrating the new state. Transactions that have been executed are chained and immutable and represent the state of the system that is mined and stored on the blockchain. As soon as a new transaction is executed and mined, this produces a transition of a new state.

The EVM is implemented on a stack-based architecture. To compute code on the EVM, the code must be written in low-level stack-based bytecode language which we can define it as an intersection between Bitcoin Script and Assembly language. The size of the stack item is 256-bits (32-bytes) which is also the size of the word size of the machine that is ordered in a set of bytes which information can be stored or operate with the machine. This facilitates the Keccak-256 cryptographic hash scheme and also allows the use of the Elliptic Curves Cryptography (ECC) to sign a scheme for validating the origin and integrity of transactions.

Every node running the EVM in Ethereum networks can execute all same instructions from other nodes for achieving and maintaining the consensus about the state of the system. The computations are slow and have a cost but this supplies some advantages like greater data integrity and censorship-resistance. It's why Ethereum is called a `World Computer`.

3.6 Smart Contract

A contract is a collection of code and data that is possessed by a specific address on the Ethereum blockchain. Contract accounts can pass messages between themselves with Turing complete computation. For living on the blockchain, the contract must be implemented in the binary format called EVM bytecode. We said a contract is smart when its conditions of execution are fulfilled and are automatically executed on the blockchain taking all the specifications, limitations coded into the contract. Smart contracts are typically written in some high-level language like `Solidity` and then compiled into bytecode to be uploaded on the blockchain.

3.7 Solidity

Solidity is a language that looks like JavaScript which allows the development of contracts and compiles them to the EVM bytecode format. It is the most popular and the easiest language to learn from the Ethereum Community and it is developed by the Ethereum foundation. Solidity uses a huge number of programming perceptions that are implemented in other languages. Solidity supports also the high typing, variables, string manipulations, classes, functions, arithmetic inheritance, complex structure user, library and a lot of other features. Solidity was influenced by C++, Python, and JavaScript.

We use the features like `Modifier` or `Mapping` of the languages for implementing the smart contracts in this project. In the following chapters, we show a few examples of how using these different features that Solidity offers us. Unfortunately, but we cannot explain all of them.

3.7.1 Structs

Solidity provides a way to define new types in the form of structs. A struct in solidity is just a custom type. A struct is defined with a name and associated properties inside as variables. Implementation of a struct is shown in the following listing **??**:

```
struct Voter { // Struct
    uint weight;
    bool voted;
    address delegate;
    uint vote;
}
```

Listing 3.1. Example of a struct type Voter which has several variables inside.

3.7.2 Mapping

In Solidity, a mapping is represented as such hash tables which consist of having a key type and a value type pairs. The following listing **??** define how a mapping looks like :

```
mapping(address => uint) public balances;
```

Listing 3.2. Creation of a mapping, which accepts first the key type as an address, and the value type will be uint, the mapping is referenced as 'balances' for the name.

3.7.3 Modifier

Function modifiers are a special function that can't be called directly but allows them to modify behavior or automatically check a condition before executing the function. The modifiers are always called before the call of the function itself.

```
modifier minimumFund() {
        require(msg.value > 0, "No founds sent !");
        _;
}
```

Listing 3.3. Implementation of modifier that checks if the user has a minimum of Ether to send.

3.7.4 Globally variables

Solidity provides us a list of special global variables and functions that we can use. One of them is the variable `msg` and `block` we describe below :

- **block.blockhash** : hash of the given block - only works for 256 most recent blocks excluding current
- **block.number** : current block number
- **msg.sender** : sender address the message currently call
- **msg.value** : uint containing the number of wei sent with the message

3.7.5 Require

`Require` is a function of control allowing to verify a condition. If the condition is false, require will call the REVERT EVM opcode which stops the execution of the transaction. The advantage of this function is when the condition is not fulfilled, it doesn't consume all of the gas and revert the state changes. The character _ is used in modifiers. It returns the flow of execution to the original function that is annotated.

3.7.6 Event

Events are dispatched signals that the smart contracts can execute. DApp connected to Ethereum JSON-RPC API, can listen to these events and receipt them taking data inside. The advantage of the events is they can be indexed, so that means the event history is searchable later.

Chapter 4

Cryptography

In this chapter, we explain all the security-based only in the two cryptocurrencies used for the atomic-swap, Bitcoin and Ethereum. We explain what type of function they use in a general way instead of going into the details. We will see also how to create Bitcoin and Ethereum addresses to understand their incentives that they provide.

Bitcoin, Ethereum and many other cryptocurrencies are based on cryptography. Cryptography is a branch of mathematics used mainly in computer security. It allows to digital currencies the use of digital keys, addresses, and digital signatures. Most people think that a wallet contains the cryptocurrencies. This information is false, a wallet is a database containing the digitals keys only. The keys allow ownership attestation and the cryptographic-proof security model.

`Digitals signatures` are used to spends the funds from transactions. Most of the cryptocurrency transactions need a valid digital signature to be added into the blockchain. The digital signature can be generated only by the way of a secret called the private key. Anyone who has a copy of this private key can control the coins.

We talk also about digital `fingerprints` called more commonly addresses corresponding to the public key of the recipient. An address can be represented in real life like a bank account where the private key is the code to access the account. This address is generated from and target to a public key and the public key is generated from the private key. Addresses are defined as the destination of the transaction, more precisely the recipient of the funds.

The main cryptographic functions used by Bitcoin and Ethereum are ECC and cryptographic hash function all described below :

Elliptic Curve Cryptographic :

Elliptic curve cryptography is a function that implements an approach of public-key cryptography based on the discrete logarithm problem from the algebraic structure of Elliptic Curves (EC). It is expressed by the addition and multiplication on the points of a EC function. It exists multiples kind of Elliptic Curve, but Bitcoin and Ethereum are based on the same, `secp256k1`.

Cryptographic Hash Function :

A cryptographic hash function is a `one-way` hash function that takes an input data of any size and returns an output of a fixed size. The input to a hash function is called a `pre-image` or `message`. The output from the hash function is called the `hash value` or `digest`. The hash function is used to produce a `fingerprint`. Bitcoin and Ethereum don't use the same hash functions. Bitcoin uses `SHA-2` known by `SHA256` and Ethereum `SHA-3` also known by `Keccak-256`.

4.1 Private and Public Key

As we have seen in the chapter above, each cryptocurrency works with a wallet using a pair of keys that consists of a private key and a public key. The private key that we called k is a number randomly generated between 1 and 2^{256}. With the private key, we can get the public key using ECC which generates K, the public key. With the public key K, we can generate the address A by using a cryptographic hash function.

These functions have the advantages that it is easy to do in one direction but impossible in the inversed direction. To recap (see figure 4.1), It is impossible with the address to reverse the hash functions and calculate the public key and from the public key, no one can reverse the ECC function and calculate the private key. These mathematical functions are the mechanism of the secure digital signatures that prove ownership of funds.

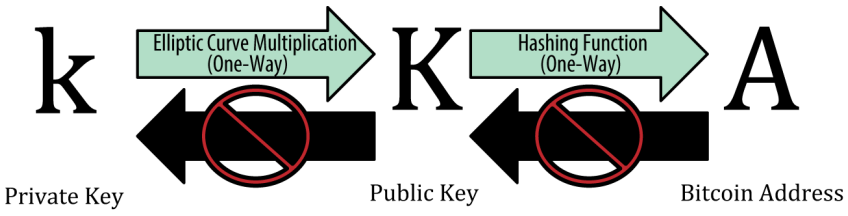

Figure 4.1. Illustration of a private key generating the public also generating the address.
Source:`https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch04.asciidoc`

4.2 Generation of a Bitcoin address

A bitcoin address looks like `1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy`. It is a string of digits and characters that are shared for anyone who wants to send money. An address is the recipient funds. It is derivated from the public key using the cryptographic hash function that produces a fingerprint or "hash". Starting with the public key K derivated from private key k, we compute the hash function $\mathscr{H}_{160}(K)$, which consists of a `SHA256` hash and then compute the `RIPEMD160` hash of the result, producing a 160-bit (20-byte) number. This algorithm is also called `Double Hashed` or `HASH160`. Then Bitcoin address is encoded as `Base58Check`. `Base58Check` is used for better human readability and for avoiding ambiguity. It protects against errors in address transcription and entry. The process is illustrated in

the equation below :

$$\begin{aligned} k &: \text{the private key; where } k \in [1, 2^{256}] \\ K &: \text{the public key; where } K = k \cdot G \\ h &: \text{the hash value of K; where } h = \mathscr{H}_{160}(K) \\ A &: \text{the address; where } A = Base58Check(h) \end{aligned} \tag{4.1}$$

4.3 Generation of an Ethereum address

An ethereum address looks like `0x95FCA54fDA1bA3c2aF5BA54F3ec250B8Fe3Ae697`. Generate an Ethereum address is simpler than generate a Bitcoin address that we have described above. Instead of computing the algorithm `HASH160` and then make a `Base58Check`, we compute only the algorithm `Keccak-256` to calculate the hash of the public key. Starting with the public key K derivated from private key k, we compute the the hash function $Keccak(K)$ Then we take only the last 20 bytes (least significant bytes)making our Ethereum address: The process is illustrated in the equation below :

$$\begin{aligned} k &: \text{the private key; where } k \in [1, 2^{256}] \\ K &: \text{the public key; where } K = k \cdot G \\ h &: \text{the hash value; where } h = Keccak256(K) \\ A &: \text{the Address; where } A \text{ is the last 20 bytes } \in h \end{aligned} \tag{4.2}$$

Chapter 5

Atomic Swap, A Method of Exchanging Different Cryptocurrencies

Definition : `Atomic Swap is the process of peer-to-peer exchange of two cryptocurrencies between two parties, without using any third-party service like a crypto exchange.`

In a few explications, an atomic cross-chain swap is a smart contract distributed where two parties or more exchange two cryptocurrencies across different blockchains. It is called cross-chain because you are no longer dependant on the blockchain. An atomic swap protocol guarantees if both parties follow the protocol, then all swaps take place. But if one of the two parties deviates from the protocol, then no conforming party and the no coalition produce automatically the cancel of the swap. At any moment, no one can control both coins, hence no coalition has an incentive to deviate from the protocol.

5.1 Atomicity

`Atom` comes from Greek and means 'a' -not/un, 'tom' -cut, in another word, no divisible or cuttable. It means that atomic transactions cannot be splittable into parts. We use the familiar expression `all or nothing` in atomic where it is the same applied concept in bitcoin. For example, Alice pays Bob in one transaction, they all know that either Bob will be paid or either bob won't. There are only two ways, the transaction is confirmed or not but there is no way for having an half-confirmation. That's the reason why the atomicity is fundamental in an atomic swap, to protect both parties, there must be no scenario in which one part can control both coins at the same time.

Another example, no atomic transaction for illustrating is when Alice wants to buy something in a web store. First, she needs to transfer the money to the site and then wait for the store to send her the object back. Here there always is a chance that Alice doesn't get her purchase.

5.2 Difference with Payment Channels

In Bitcoin, Payment Channel is a class of techniques designed to allow users to make multiple Bitcoin transactions without committing all of the transactions to the Bitcoin blockchain. In a typical payment channel, only two transactions are added to the blockchain but an unlimited or nearly unlimited number of payments can be made between the participants.[1] Actual transaction speeds on a blockchain like Bitcoin or Ethereum does not allow to perform instantaneous or fast transactions. Typically, a transaction can take up to several dozen minutes until it is included in a block. Payments channels are one way to address this issue by allowing participants of a channel to perform transactions off-chain which are performed faster, without compromising on the safety properties of cryptocurrencies. To recap, it is faster cheaper transaction between two parties because each transaction doesn't need to be written to the blockchain.

An atomic swap is not a payment channel, however atomic swap implement the same off-chain system from payments channel and its tools of it as such Hashed Timelock Contracts (HTLC) (HTLC), a technique that can allow payments to be securely routed across multiple payment channels that we describe below (see chapter 5.3). It is a concept from the Bitcoin community that is used in the Lightning Network.

5.3 Security by Hashed Timelock Contracts

HTLC is a kind of smart contracts that allows eliminating counterparty risk using tools like hashlock and timelock. It enables time-bound transactions between the two parties. A time-bound means when a recipient at the other end of the transaction is required to acknowledge the transaction, the person needs to provide cryptographic proof. This cryptographic proof is required until a specific deadline. When the exchanges mights fail, that automatically makes the transaction null and void. In practical terms, this means that recipients of a transaction have to acknowledge payment by generating cryptographic proof within a certain timestamp. Otherwise, the transaction isn't valid. The cryptographic proof of payment that the receiver generates can then be used to trigger other actions in other payments, making HTLC a powerful technique for producing transactions puzzle.

There are many benefits for HTLC :

1. It prevents the person who is making the payment from having to wait indefinitely to find out whether or not his or her payment is confirmed.
2. The person who makes the payment will not to be worried about losing his or her money if the payment is not accepted. It will simply be returned.
3. The recipient helps to validate the payment on the blockchain because cryptographic proof of payment is required for the recipient to accept the payment.
4. The hashes that are created for the HTLC can be easily added to blockchains.
5. Each party is protected from counterparty risk. This means that the structure of the method allows the people sending and receiving the payments do not have to trust each other or even know each other to make sure that the contract will be executed properly.

[1] Micropayment channel: Bitcoin.org Developer Guide

To work, A Hashed Timelock Contract implements several elements from existing cryptocurrency transactions. The concept of signatures, HTLC uses multiple signatures that consist of using a private key and public key to verify and validate transactions. The main elements that make HTLC a powerful method are the concept of `hash lock` and `timelock`.

5.3.1 Hashlock

A hashlock is a type of primitive that restricts the spending of an output until a specified piece of data is publicly revealed. When a person reveal the hashlock, he allows to any other hashlock secured using the same key can to be open. The hashlock is a hash version of a cryptographic value generated by the originator of a transaction.

5.3.2 Locktime

A Timelock is a type of smart contract primitive that restricts the spending of some bitcoins until a specified future time or block height. Timelocks are implemented in many Bitcoin smart contracts, payment channels and hashed timelock contracts.

Absolute Timelock

The locktime will be set in an absolute way. It can be interpreted as two ways. First way, it is defined by block height or by the time, which will be`Unix Epoch timestamp`. If it is interpreted as a block height, it will be recorded as at block e.g. number 455488. If it is interpreted as the time, it will be set as 1564003242 secondes which means e.g. 07/24/2019 @ 9:20pm (UTC). Relative timelock is represented by an absolute UTXO-level timelock and has been was added to Bitcoin by the BIP 65. CLTV included a `nLocktime` is implemented in the Bitcoin's script are defined in the Bitcoin's script as `OP_CHECKLOCKTIMEVERIFY` (see the listing 5.1).

```
1 IF
2     <provider pubkey> CHECKSIGVERIFY
3 ELSE
4     <expiry time> CHECKLOCKTIMEVERIFY DROP
5 ENDIF
6 <client pubkey> CHECKSIG
```

Listing 5.1. Example of locking script with CheckLockTimeVerify.

Relative Timelock

The relative timelock are similar to absolute timelock and can also be presented by either block height or by the time. For example, if we want set a timelock to 1 day, this values is corresponds to 144 blocks in Bitcoin for being achieved. If it is interpreted as the time, it will be set as after 86400 seconds. The relative timelock includes the Check Sequence Verify (CSV) and a nSequence based on the specification BIP 68. It shows up in the Bitcoin script as the opcode `OP_CHECKSEQUENCEVERIFY` (see the listing 5.2).

```
1 IF
2     <provider pubkey> CHECKSIGVERIFY
3 ELSE
4     <expiry time> CHECKSEQUENCEVERIFY DROP
5 ENDIF
6 <client pubkey> CHECKSIG
```

Listing 5.2. Example of locking script with CheckSequenceVerify.

Chapter 6

Protocol

We describe an analysis of the problematic and the implementation of the cross-chain atomic swap protocol between Bitcoin and Ethereum. The protocol can be generalized for Bitcoin and any other cryptocurrencies that fulfill the same requirements as Bitcoin (e.g. LiteCoin), see the chapter 6.3. This protocol is heavily based on the `BIP-199` (BIP) from [Bowe and Hopwood, 2017] for the Bitcoin part. For Ethereum the concept is roughly the same but with fewer prerequisites than Bitcoin. For sending funds, each participant must generate a specific address called HTLC on each blockchain, lock funds onto these addresses where the other party can take control from the other blockchain.

6.1 Limitations

The most important process of the protocol is `liveness`. Liveness means that participants must be online for respecting the protocol (at least one participant is still online). In the worst scenario where someone doesn't follow the protocol, it can happen the coalition end up and lose the funds. This happens only if a party does not remain online during the swap or it has not claimed the funds in time.

Secondly, there is another factor to take on board which is the `Fees`. Each blockchain has different fees because there are built with different internal parameters and transaction complexity. It is also due to a factor, the block space that depends on the demand. In this project, we use the Bitcoin Blockchain as a tool, more precisely, we use some advanced features that increase the cost of the transaction for bitcoin side as `P2SH`. In general, the transaction is more expensive on Bitcoin than Ethereum, because the transaction cost in Ethereum doesn't depend on the user but by the operation in the contract.

The difficult problem with cross-chain swaps is off-chain coordination. In a few words, it's to find an agreement between the two parties on specific conditions. This agreement can depend on the speed of the protocol (i.e. to considerate that confirmation is confirmed) but the speed is influenced by the slowness and several confirmations required for validating a confirmation in each blockchain side. The protocol is slow but it can be extended by way of setups. The only thing we can change from the setups is the ranges of fees that can be consumed in a transaction.

6.2 Scenario

Alice and Bob want to exchange 1 Alice tokens for 10 Bob tokens. The problem is that they are not in the same blockchain, Alice token is defined in Bitcoin blockchain, whereas Bob token is only present Ethereum blockchain.

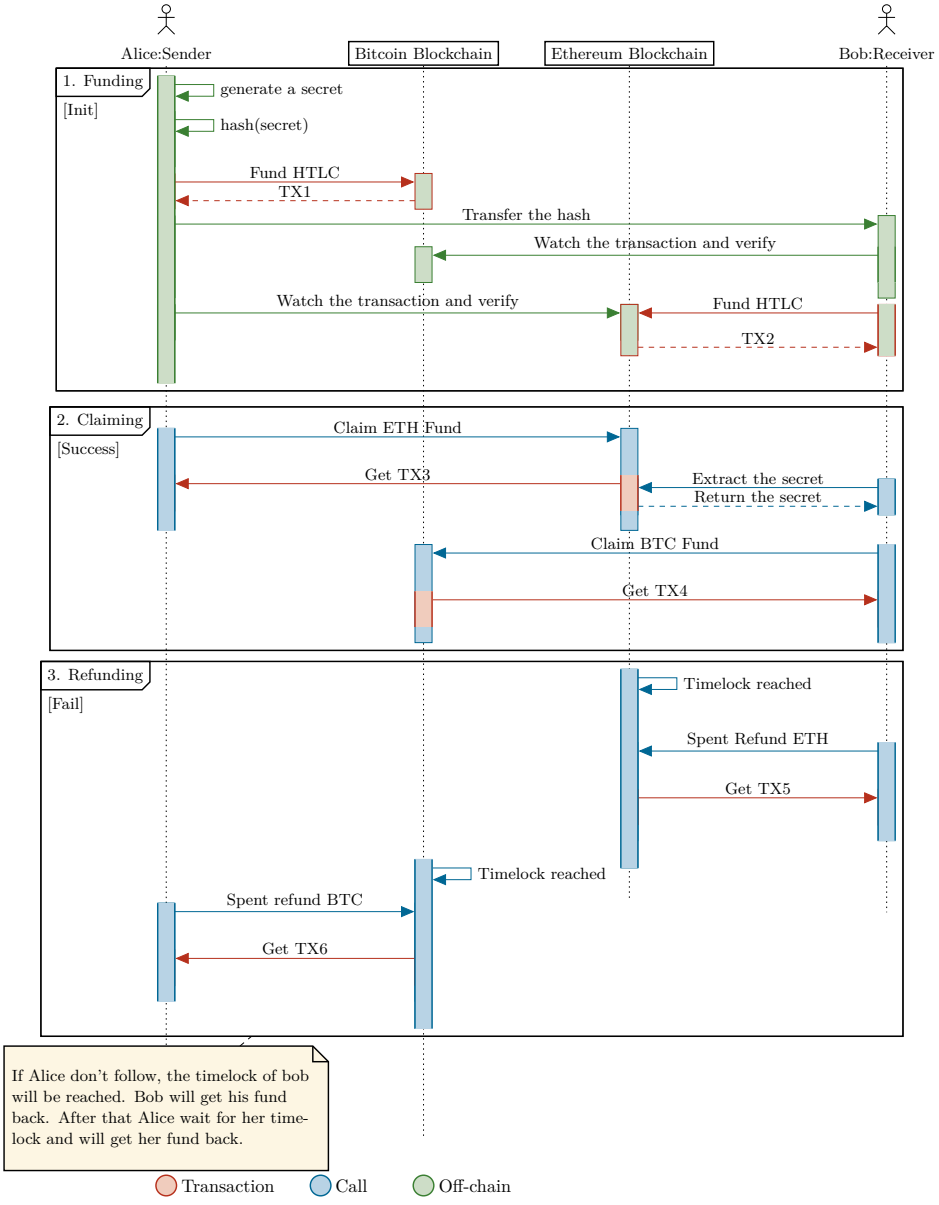

Figure 6.1. Sequence of atomic swap protocol.

Let's see the process in figure 6.1 :

1. Alice generates a random set of bytes called value or preimage. The proof should

have a size of 32 bytes.
2. Alice hashes the obtained proof to generate the secret.
3. Alice is the instigator of the swap, she starts by initiating the locking script, TX1 in the Bitcoin chain.
4. Upon doing this, Alice broadcasts TX1 to the Bitcoin chain and transfer the secret to Bob.
5. Bob defines locking script transaction TX2 to Ethereum using the hash.
6. Only Alice can unlock the ETH in this address because she has the value which generates that particular hash. It's the claiming transaction.
7. Alice can get her ETH by signing a transaction for Bob's contract address and Bob can retrieve the BTC by signing a transaction TX3 for Alice's contract address.
8. When Alice signs Bob's contract address with the value, she unlocks the address and reveals the value to Bob as well.
9. Bob, now knowing the value, signs off the transaction TX4 for Alice's address and retrieves his BTC.

To summarize the process, the scenario describes the participants and their incentives. `Alice` the sender owns Bitcoin (BTC) and Bob the receiver owns ether (ETH), they want to swap funds. Alice and Bob have already negotiated the price in advance and found an agreement (i.e. an amount of bitcoin for an amount of ether to swap). They are only two possible ways of execution path for both parties :

- `The protocol succeeds` - Alice gets her ETH and Bob his BTC.
- `The protocol failed` - both parties keep their fund (they will lose some amounts because they need to pay some fees for each transaction).

6.2.1 Successful swap

For having a successful swap, both parties must follow the protocol. They will be four transactions in total, 2 transactions for Bitcoin blockchain and 2 transactions for Ethereum blockchain :

1. Lock the funds in Bitcoin and make it ready for the swap.
2. Lock the funds in Ethereum and make it ready for the swap.
3. Unlock the funds in Ethereum.
4. Unlock the funds in Bitcoin.

When participants unlock the funds, they take control of the output of the contract in the other chain. here is the most simple and optimal way to perform the protocol. Here, no timelock is required but both participants must care about the minimum number of transactions and for the minimal transaction, the funding transaction (locking fund). Confirmations vary between each chain, so it needs to be considered if both parties are expecting the funding transaction to be considered final and are sure to keep going the protocol.

6.2.2 Swap aborted

The swap is aborted only if one party wants not to continue the process. To get the refund of the locked funds, Alice or Bob must wait for the timelock is reached. When Alice

starts, there is no ETH but only BTC locked into a contract. If bob doesn't follow, so Alice waits for her timelock and after that, she can spend the refund. The length of time on each lock is important to ensure that the game can only be played fairly. Alice's time lock should be longer and Bob's lock should be much shorter. This is because Alice knows the hash lock secret and therefore has a major advantage. It is very important because if Alice's timelock had the shorter refund time, Alice could wait until that time expires, refund herself the Bitcoin and after that then enter the secret preimage into Ethereum to claim the ETH that Bob sent. Alice would have both coins and Bob would lose his ETH.

6.2.3 Worst scenario

There is a possibility that the protocol can be broken again if a party doesn't follow the rules from the Bitcoin part. If the swap process succeeds with Alice claiming ETH funds and Bob doesn't claim his BTC fund before Alice's timelock, then Alice can spend her refund as soon as her timelock is reached. It will conclude that Bob would lose his funds and Alice would get both coins. In Ethereum this can't happen because when the timelock is reached, claim funds are automatically blocked and Alice cannot claim the fund, only Bob spent the refund to avoid that situation. To resolve this problem, we must implement a protocol that forces Bob to be not offline or compensate Bob if Alice doesn't follow correctly the protocol.

6.3 Prerequisites

In chapter 6.2, we describe the conditional process that must be followed to guarantee a swap with atomicity. Bitcoin has a small stack-based script language that allows for conditional execution and timelocks. Whereas Ethereum uses the programming language that allows hashing and timelocks too. The challenge is then to implement the BIP-199 in Ethereum.

6.3.1 Bitcoin

The bitcoin transactions in this protocol use Segregated Witness structure that allows reducing the fees. For any other cryptocurrencies with a bitcoin style UTXO model as such e.g. Litecoin, these requirements must be fulfilled for having the same compatibility with this protocol. E.g. Bitcoin Cash isn't compatible.

Pre-image

Generation of a valid pre-image $\alpha \in \mathbb{Z}_{2^{256}}$ of 32 bytes size to a given $h = \mathscr{H}_{256}(\alpha)$ where $\mathscr{H}_{256}$ is the `SHA256` algorithm.

Public key hash

For a public key Q to a given $h_Q = \mathscr{H}_{160}(Q)$ where $\mathscr{H}_{160}$ is the `SHA256` followed by the `RIPEMD-160` algorithm. h_Q is the version of Q that is given to another participant so that they can send it bitcoins. It's shorter than the original public key, and it may provide an extra layer of security for the bitcoins compared to giving the public key directly.

Hashlock

Hashlock is for revealing the secret to the other participant. It is a primitive that includes a value to reveal some data (pre-image) that is associated with a given hash where $h_s = \mathscr{H}_{256}(s)$ and handle the spent of the HTLC.

Timelock

The timelock is to enable an execution path that is predefined by an amount of time. This amount of time is expressed in a number of block `nSequence` where $t = nSequence$. We use the number of the block instead of the amount of time in second for avoiding a problem called `leap second`.

Multi signatures

The signatures of both participants are required for creating HTLC only accessible by them if they agree.

6.3.2 Ethereum

Ethereum doesn't use the same Model as Bitcoin (UTXO) but is based on `Account Model`. Every cryptocurrency with this style model that must fulfill these requirements for having the same compatibility with this protocol. In comparison to Bitcoin, Ethereum uses a smart contract that handles also timelock and hash lock. However, we don't need the `Public key hash` for the verification. We use instead, the `msg.sender` from Smart contract that allows verification.

6.3.3 Elliptic Curve

Bitcoin and Ethereum do use the same elliptic curves. They use the `secp256k1` curve from Standards for Efficient Cryptography (SEC) with the Elliptic Curve Digital Signature Algorithm (ECDSA) algorithm. The curve is described as follow :

$$
\begin{aligned}
&p : \text{a prime number; } p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1 \\
&a : \text{an element of } \mathbb{F}_p;\ a = 0 \\
&b : \text{an element of } \mathbb{F}_p;\ b = 7 \\
&E : \text{an elliptic curve equation; } y^2 = x^3 + bx + a \\
&G : \text{a base point; } G = \\
&(\texttt{0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798}, \\
&\ \texttt{0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8})
\end{aligned}
\tag{6.1}
$$

6.4 Hashed Timelock Contract

The description of the protocol is as follows: Alice moves her bitcoin into a `P2WSH` address where each participant controls a type of transaction using Bitcoin scripting language. Bob does the same into an Ethereum `Smart contract` address that is then used to reveal the

secret depending on Alice who claims the ether. Bitcoin and Ethereum transactions are designed in such a way that if a participant follows the protocol, there is no way for losing his coin. If the deal goes through, Alice spends the ether by revealing the secret, thus allowing Bob to spend the locked bitcoin. If the deal is aborted, Bob spends the ether after the second timelock, thus allowing Alice to spend the bitcoin after the first timelock. In both cases, the participants must add transaction fees. The full protocol is described in Table 6.1.

Alice (BTC→ETH)		Bob (ETH→BTC)
	Initialization phase	
$k_A \xleftarrow{R} [1, p-1]$		$k_B \xleftarrow{R} [1, p-1]$
$Q_A \leftarrow k_A G$		$Q_B \leftarrow k_B G$
$h_A \leftarrow \mathscr{H}_{160}(Q_A)$		$h_B \leftarrow \mathscr{H}_{160}(Q_B)$
$s \xleftarrow{R} \mathbb{Z}_{2^{256}}$		
$h_s \leftarrow \mathscr{H}_{256}(s)$		
	$\xleftarrow{\langle \text{send } h_B \rangle}$	
	Lock phase	
create BTX_{lock} $\langle h_A, h_B, h_s, t_0 \rangle$		
sign BTX_{lock}		
broadcast $\text{BTX}_{lock}^{signed}$		
	$\xrightarrow{\langle \text{send } h_s, \text{ watch } \text{BTX}_{lock} \rangle}$	
		verify $\text{BTX}_{lock}^{signed}$
		create ETX_{lock} $\langle h_s, t_1 \rangle$
		sign ETX_{lock}
		broadcast $\text{ETX}_{lock}^{signed}$
	$\xleftarrow{(\text{watch } \text{ETX}_{lock})}$	
verify ETX_{lock}		
	Swap phase	
unlock ETX_{lock} $\langle s \rangle$		
broadcast ETX_{unlock}		
	$\xrightarrow{\text{watch } \text{ETX}_{unlock}, \text{ extract } s}$	
		spent BTX_{lock} $\langle s \rangle$
		broadcast BTX_{unlock}

Table 6.1. Full protocol of cross-chain atomic swap between Bitcoin and Ethereum with Alice and Bob with initialization, lock, and swap phases.

6.4.1 Time parameters

$$\begin{aligned} t_0 &: \text{ Alice's timelock} \\ t_1 &: \text{ Bob's timelock} \end{aligned} \tag{6.2}$$

We use two timelocks t0 and t1 that are defined during lock swap. t0 sets the time for Alice where it is safe to execute the exchange. When t0 is passed, the refund may start. t1 sets the response time during which Alice is required for claiming the coin from Bob and reveals her preimage. When t1 is passed, Bob can get his ether back and allows Alice to redeem her bitcoin. Note that all timelocks are defined in Relative Timelock, see 5.3.2.

6.4.2 Bitcoin Script

Swaplock

P2SH is used to lock funds and defines the two base execution paths :

1. swap execution [success].
2. refund execution [fail].

The script is defined with Bob's h_B public key hash, Alice's h_A public key hash and the preimage h_s in the Listing 6.1:

```
1 OP_IF
2     OP_SHA256 <h_s> OP_EQUALVERIFY OP_DUP
3     OP_HASH160 <h_B>
4 OP_ELSE
5     <t_0> OP_CHECKSEQUENCEVERIFY OP_DROP OP_DUP
6     OP_HASH160 <h_A>
7 OP_ENDIF
8 OP_EQUALVERIFY
9 OP_CHECKSIG
```

Listing 6.1. Swaplock script.

The Swaplock is executed when `OP_IF` reads a `TRUE` value from the stack. It expects a secret value, an ECDSA signature and the Public Key Hash (PKH). It hashes the secret and checks that it matches a given hash, then it checks PKH followed by the signature against the given public key. When the value `FALSE` from the stack is read, it executes the `OP_ELSE` branch.

Claim Fund

Bob takes control of bitcoin in using the pre-image s and his public key hash h_B from Alice to redeem the `Swaplock P2SH`. To redeem the HTLC from this way, Bob uses the following script in the input of a transaction:

```
1    <sig_B> <h_B> <s> OP_TRUE
```

Listing 6.2. Bob's script signature

Spend Refund

With this contract Alice can spend this output with her public key hash h_A after the timelock t_0 with the script signature :

```
1    <sig_A> <h_A> OP_FALSE
```

Listing 6.3. Alice's script signature

6.4.3 Ethereum Smart Contract

Ethereum doesn't use script language Bitcoin but the programming language for the smart contract. The smart contract allows to create functions from the UML diagram in figure 6.2 :

Function lock()

A function that will create a contract with all the prerequisites and lock it with the address of the sender `Bob` and the address of the receiver `Alice`.

Function unlock()

Function when it is called, check if the pre-image s is correct and then transfer the fund to `Alice`.

Function refund()

Function when it is called, check if the timelock t_1 is reached and then transfer the fund to `Bob`.

6.4.4 Transactions

All the transactions are described in figure **??**.

Funding transaction

The funding transaction is the transaction sending fund to the contract address. BTX_{lock}, bitcoin transaction with 1 or more inputs from Alice and the output (vout) to the `Swaplock P2SH`. ETX_{lock}, ethereum transaction from Bob that sends funds to the Smart Contract address.

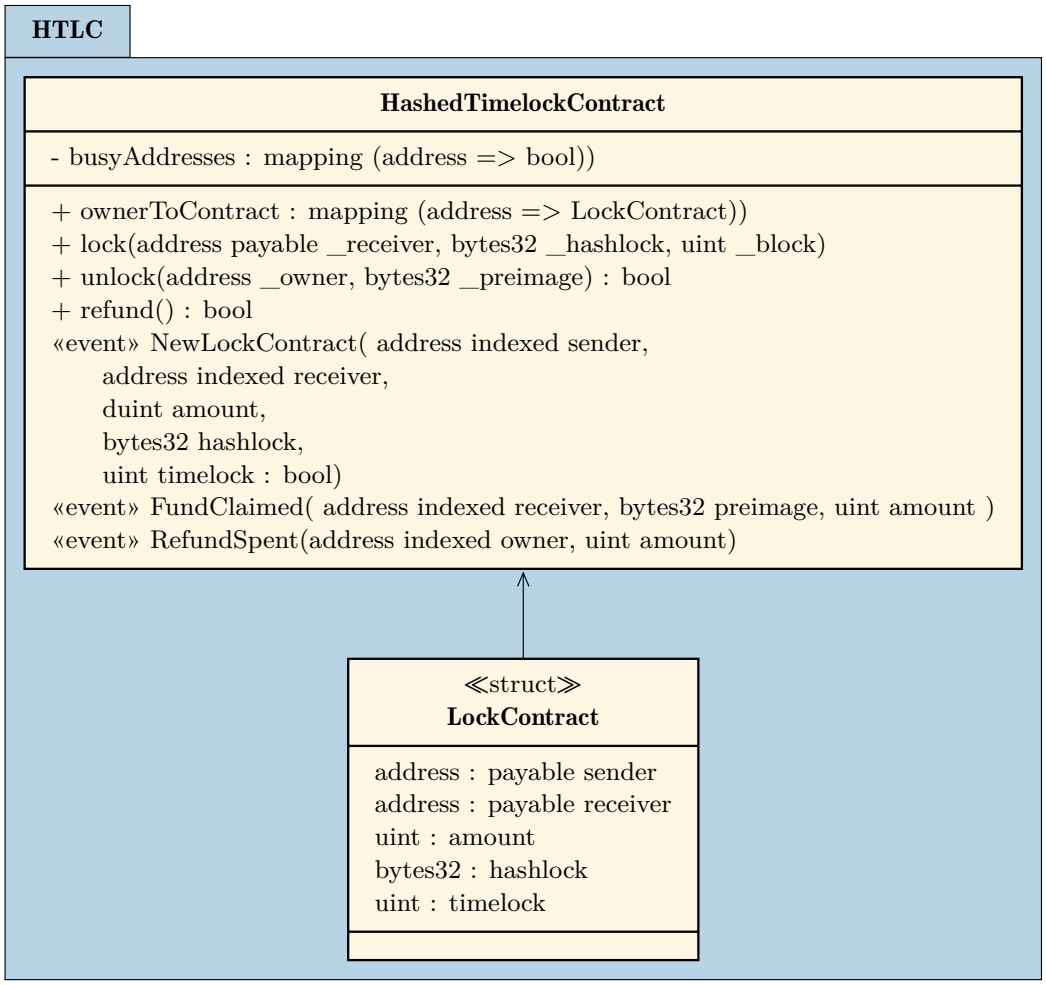

Figure 6.2. UML class diagram of the `Smart Contract` reference Implementation in Ethereum.

Claim Transaction

The claim transaction is a transaction that allows the sender to spend the funds. BTX_{unlock}, bitcoin transaction with 1 inputs consuming `Swaplock P2SH` (BTX_{lock}) and 1 output vout to Bob. ETX_{unlock}, ethereum transaction from the `Smart Contract` that call the function `unlock` to send the funds to Alice.

Refund transaction

The refund transaction is a transaction that allows the sender to abort the swap and get his funds back. BTX_{refund}, bitcoin transaction with 1 inputs consuming `Swaplock P2SH` (BTX_{lock}) and 1 output vout to Alice. ETX_{refund}, ethereum transaction from `Smart Contract` that call the function `refund` to send back the funds to Bob.

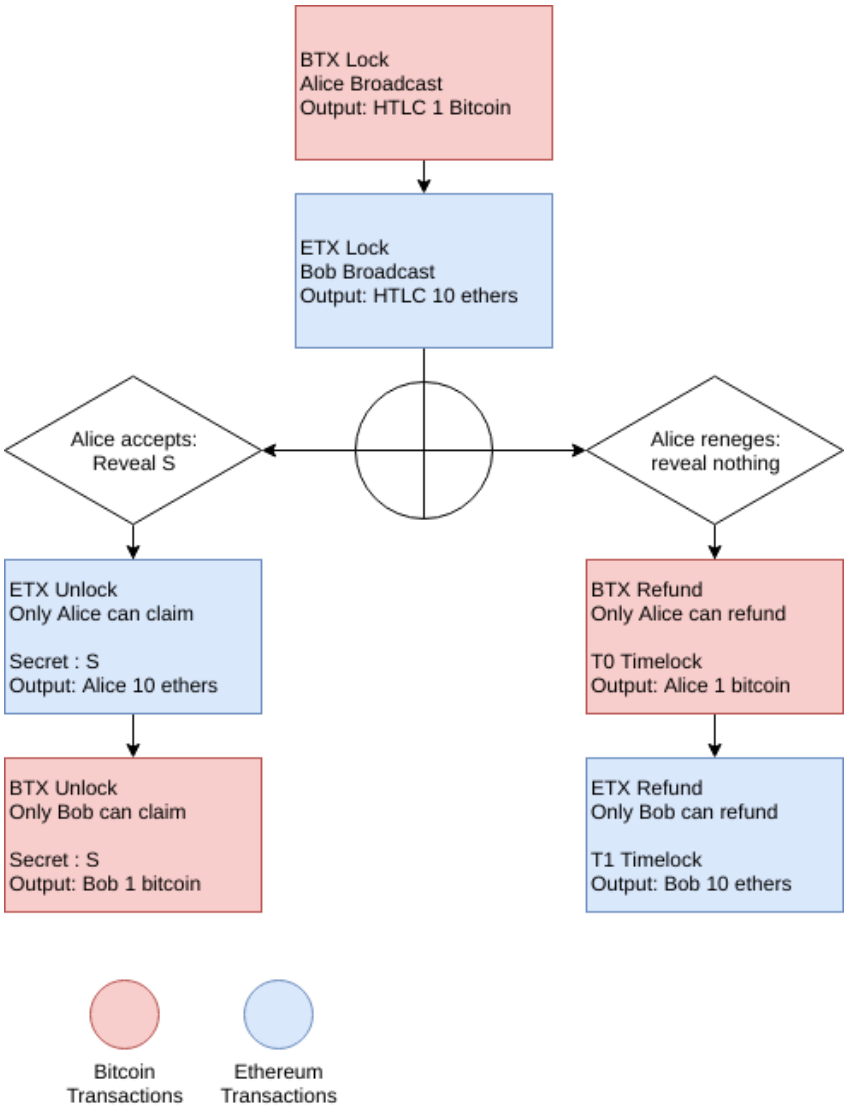

Figure 6.3. List of transactions.

Chapter 7

Implementation in Bitcoin with bitcoinjs-lib

For this project, we describe a very simple method of constructing and executing bitcoin smart contracts that sacrifice some privacy and potentially some security. The implementation is spread into five main components (i) a `generation of a wallet`, (ii) `how estimate the fees in Bitcoin`, (iii) `implementation of the funding transaction`, (iv) `implementation of the claiming transaction`, and (v) `implementation of the refunding transaction`. Note that the current implementation is not ready for production. This implementation is more an educational work than an implementation for production and needs to be reviewed and tested more deeply before being used in production.

This chapter refers to the implementation available on GitLab at `https://gitlab.com/Skogarmadr/atomic-swap/tree/master` of the writing of this report. Note that the sources may change or evolve from the last version of the code. So, the last version is updated in the GitLab.

7.1 Configuration

The project employs NodeJS with modern Typescript, for the simplicity of use and the installation of dependance modules. The library used is `bitcoinjs-lib` to manage all parts of Bitcoin. The incentive of this library is, (i) compatible with Typescript, (ii) still active, (iii) has a large number of contributors and releases and (iv) has a folder full of clear examples. The project employs NodeJS with modern Javascript, for the simplicity of use and the installation of dependance modules. The library used is `bitcoinjs-lib` to manage all parts of Bitcoin. The incentive of this library is, (i) compatible with Typescript, (ii) still active, (iii) has a large number of contributors and releases and (iv) has a folder full of clear examples.

There is `config.json` file that contents all the data needed for the environment works (i.e. Alice and Bob entropy for getting back the key pair). It is important to note this config file in production must be saved only on the client-side because it is private data. Note also that the project works with the `Tesnet Network` because he is still in development and not testing for the mainnet Network.

```
{
    "alice": {
        "entropy": "d399c2cbabdc9fa8790ec5111bb05da6",
        "phrase": "squeeze sock real fish size stage tomorrow suffer
            baby talk blast erupt",
        "seed": "4973ee8e81e0047a633f3b0d3cec61ec3b028846ffb648f509d5
            5be97e72a2e502e35148b463770cea5394bb7fe53a72344f7b5ffe08f
            3b208c27adb33f95d92",
        "prvWIF": "cPmUdWJYPESULvX3s2tFBfwJ5FWiWHm2wcx8pmiPiK6a2
            yKsrrb6",
        "xprv": "tprv8ZgxMBicQKsPd4RZif2NZzSMcFYdtEbCL8nSovoBYzTobGy9
            Pyjx6kK1BhZjYGRyvoMgyshBSNBrAkaaax6fGF7YAoq2i74ZEdjR2
            NfVKcQ",
        "xpub": "tpubD6NzVbkrYhZ4WXTMcJgxyQ6UBH4a3Zn6uSPE6SqUyGGCRmDv
            2NZYHEvsMqqNbbzKJYh2Lacgh37J2EMbDrmxtBSZfP5hbQBRMmDfshtPH
            3G"
    },
    "use_testnet": true,
    "APINETWORK" : "BTCTEST/"
}
```

Listing 7.1. Example of a config file for Bitcoin.

7.2 Generation of an HD wallet

In this project, we use always the same set of addresses, but in production, the practice says that the user shouldn't reuse the same addresses. For that, we need a wallet. We decide to create our wallet to see how a wallet is built from the beginning to the end. To do anything with Bitcoin, you need a private and a public key pair. The public key can be used by other people to send you Bitcoin, and the private key can be used by you to send Bitcoin to someone else by verifying who created the transaction.

We generate our mnemonic and with this we create two different BIP32 Hierarchical Deterministic (HD) wallets for Alice and Bob, each containing one distinct ECDSA key pairs. From each public key is derived one set of Bitcoin address for each type of PKH output. The code for generating an HD wallet is in Listing **??**

```
1
2  const bip32 = require('bip32');
3  const bip39 = require('bip39');
4  const crypto = require('crypto');
5
6  // 128 bit entropy => 12 words mnemonics
7  // Generate random entropy
8  const alice_randomBytes = crypto.randomBytes(16);
9  const bob_randomBytes = crypto.randomBytes(16);
10
11 const alice_entropy = alice_randomBytes.toString('hex');
12 const bob_entropy = bob_randomBytes.toString('hex');
13
14 const funding_path = "m/44'/0'/0'/0/0";
15
16  // Get mnemonic from entropy
17  var mnemonic = bip39.entropyToMnemonic(alice_entropy);
18
19  // Get seed from mnemonic
20  var seed = bip39.mnemonicToSeedSync(mnemonic);
21
22  // Get BIP32 master from seed
23  var master = bip32.fromSeed(seed, NETWORK);
24
25  // Get child node
26  var child = master.derivePath(funding_path);
27
28  // Get child wif private key
29  var wif = child.toWIF();
30
31   // Get child extended private keys
32   var childXprv = child.toBase58();
33
34   // Get child EC public key
35   var ECPubKey = child.publicKey.toString('hex');
```

Listing 7.2. Generation of a HD wallet.

```
[
  {
    "username": "alice",
    "wallet": [{
      "wif": "cPg4ssEMrxSELzpD6hK52y1tfPcYfJvU1R153HhP2yWoyUNihitK",
      "pubKey": "0307560f56d2652c309b732394fa1c460fae7231f12adb271532
          9241028cca888a",
      "pubKeyHash": "ea2da066b9fdadea872af0d4ac138b8c1d4181ae",
      "p2pkh": "n2sB58biJzTCpbtzPgdDvCPCSYbt9oRuCs",
      "p2sh-p2wpkh": "2MwfFe3fmjoe7AMEMTxcVZChMvKSJC9ocoB",
      "p2wpkh": "tb1qagk6qe4elkk74pe27r22cyut3sw5rqdwyq5dk6"
    }]
  },
  {
    "username": "bob",
    "wallet": [{
      "wif": "cUpa4EZsjN4YL25ZfgqT9LR3jRzwPrD2Hc7E8HnPizX2A83PhMno",
      "pubKey": "039d04be8039c20e3799af14a61c5cdc86d2103c4d45b80a7ac8
          f25b4834722475",
      "pubKeyHash": "a007223750ccd2cead48848a731f062839b1cc7a",
      "p2pkh": "mv76xQGhSj2CChqkAg9tcCuMNVDdqCRuhA",
      "p2sh-p2wpkh": "2N7PRqTkPDgtj2XQL3tXMZVQtHwLTs1KfnP",
      "p2wpkh": "tb1q5qrjyd6senfvat2gsj98x8cx9qumrnr6qdska7"
    }]
  }
]
```

Listing 7.3. Example of a wallet.

In the Listing 7.3, Wallet Import Format (WIF) is the version of the compressed private key. With that, we can get our EC key pair. With the key pair, we can derivate all data that we need (e.g. the address of each person).

7.3 Get free Testnet bitcoins

For making new transactions, we need bitcoin. Testnet Network is a useful tool that offers Testnet Bitcoin, a valueless coin for testing our application without spending any money. For that, we need to get theses free Testnet Bitcoins. We use `faucet` to get theses bitcoin Testnet, they can be hard to found sometimes. The one we use is `https://tbtc.bitaps.com/` for the two reasons: (i) it gives us 0.01 Bitcoin every 5 minutes and (ii) it is compatible with the SegWit addresses.

7.4 Get data from the Testnet blockchain

To read data from a public blockchain, we use a block explorer that is a third-party REST API instead of running our node in local and explore it ourselves. Here, for this

project, we use (https://chain.so/api because they have a solid API, an excellent documentation site, and you don't need an API key. This allows us to get e.g. the Unspent Transaction Output (UTXOs) for having got the balance of an address.d

```
async function getUnspentTx(address: string) {
    try {
      return $.get('https://chain.so/api/v2/get_tx_unspent/' +
          APINETWORK + address);
    } catch (error) {
    }
  }
```

Listing 7.4. Get Unspent Transactions Output from API.

7.5 Estimate fees

For building a valid transaction, there is an important part to not forget in Bitcoin transaction: the fee. The fee is important because if you set it too low then the transaction won't be attractive to miners and might take a long time to appear in the blockchain or it might even never get accepted. This is not such a problem on the Network Testnet because there is less traffic, but that can be a big issue for the Bitcoin main network. We set the fee with a REST API that supplies a list of fee recommended per byte of transaction data. We use always the fastest fee rates because we don't need to be precious with the satoshis of Testnet Bitcoin. We need only test it and we don't want to wait one day for our transactions must be accepted. In production, the fee rate will depend on the need of the user, if he wants his transactions must be confirmed quickly or slowly.

```
const feeRate = await getFeeRateByTarget(0.5);
const estimatedTxSize = inputCount * 104 + outputCount * 32;
const estimatedTxSizeReceiver = 104 + 32;
const estimatedFeeAmount = Math.floor(feeRate * estimatedTxSize);
const estimatedFeeAmountReceiver = Math.floor(feeRate *
    estimatedTxSizeReceiver);
```

Listing 7.5. Get the estimation of fee amount.

To calculate the fee, we base us on the SegWit size of a transaction. We compute the number of `inputs` minimum equal to 1 and his `outputs` maximum equal to 2, more precisely if the sender has a rest. For the receiver, the size of the transaction is simpler, the HTLC has only 1 input and 1 output.

7.6 Funding transaction

For creating a funding transaction we base us on the Bitcoin Script from the protocol 6.4.2 in the chapter 6. First, we need to construct a `csvCheckSigOutput` for the building of

our P2SH Script. Here, we use SegWit transaction, thus we will create instead a P2WSH. In a P2WSH context, a redeem script is called a witness script.

```
function csvCheckSigOutput(senderPubkeyHash: bitcoin.StackElement,
    receiverPubkeyHash: bitcoin.StackElement, hash: bitcoin.
    StackElement, sequence: number) {
    return bitcoin.script.compile([
      bitcoin.opcodes.OP_IF,
      bitcoin.opcodes.OP_SHA256,
      hash,
      bitcoin.opcodes.OP_EQUALVERIFY,
      bitcoin.opcodes.OP_DUP,
      bitcoin.opcodes.OP_HASH160,
      receiverPubkeyHash,
      bitcoin.opcodes.OP_ELSE,
      bitcoin.script.number.encode(sequence),
      bitcoin.opcodes.OP_CHECKSEQUENCEVERIFY,
      bitcoin.opcodes.OP_DROP,
      bitcoin.opcodes.OP_DUP,
      bitcoin.opcodes.OP_HASH160,
      senderPubkeyHash,
      bitcoin.opcodes.OP_ENDIF,
      bitcoin.opcodes.OP_EQUALVERIFY,
      bitcoin.opcodes.OP_CHECKSIG,
    ]);
  }
```

Listing 7.6. csvCheckSigOutput function that returns the redeemScript.

We need to initiate and create the witness script for the P2WSH address that will be the HTLC address with the parameters required.

```
const senderKeyPair = bitcoin.ECPair.fromWIF(senderWif, NETWORK);
const senderP2wpkh = bitcoin.payments.p2wpkh({ pubkey: senderKeyPair.
    publicKey, network: NETWORK });
const senderAddress = senderP2wpkh.address as string;
const sequence = bip68.encode({ blocks: nbBlock });ddd
const witnessScript = csvCheckSigOutput(senderKeyPair.publicKey,
    pubKeyReceiver, revHash, sequence);
const p2wsh = bitcoin.payments.p2wsh({ redeem: { output:
    witnessScript }, network: NETWORK });
const p2wshAddress = p2wsh.address as string;
```

Listing 7.7. Initialization of the P2WSH address.

Now we must start building the transaction that will send bitcoins into the P2WSH address. We must add the unspent output as input from the sender address. Then we add outputs sending the desired amount to the receiver address, and the change back to our address. We calculate how much change there is by subtracting the amount we're sending

and the fee that we have estimated from the unspent amount. Then we need to sign that input using the key pair of the sender.

```
const txb = new bitcoin.TransactionBuilder(NETWORK);

let totalBalance = 0;
let inputCount = 0;
const values = [utxos.length];


for (const utxo in utxos) {
    if (utxo) {
    totalBalance += Math.floor(parseFloat(utxos[utxo].value) * 1e8);
    values[inputCount] = Math.floor(parseFloat(utxos[utxo].value) * 1
        e8);
    txb.addInput(utxos[utxo].txid, utxos[utxo].output_no, undefined,
        senderP2wpkh.output);
    console.log("adding input: " + utxos[utxo].txid + " to funding
        transaction");
    inputCount++;
    }

}

const estimatedTxSize = inputCount * 104 + 2 * 32;
const feeRate = await getFeeRateByTarget(0.5);

const estimatedFeeAmount = Math.floor(100 * estimatedTxSize);
const sendAmount = amountSatoshis + estimatedFeeAmount;

if (totalBalance - sendAmount > 0) {
    if (totalBalance - sendAmount !== 0) {
    txb.addOutput(senderAddress, (totalBalance - sendAmount));
    }

    txb.addOutput(p2wshAddress, sendAmount);

    for (let i = 0; i < inputCount; i++) {
    txb.sign(i, senderKeyPair, undefined, undefined, values[i]);
    }

    const fundingTransaction = txb.buildIncomplete();
    console.log("");
    console.log("===== Funding Transaction to " + p2wshAddress + " )
        =======");
    console.log("Transaction ID: " + fundingTransaction.getId());
    console.log("Transaction (needed for broadcasting TX): " +
        fundingTransaction.toHex());

    await broadcastTx(fundingTransaction.toHex());
}
```

Listing 7.8. Creation of the funding transaction.

7.7 Claiming transaction

Once funds are sent to the P2SH addresses, we can spend them using the unlocking input scripts. We have already generated the witness script so we just need to get it and to create a transaction with it and sign it. First, we must prepare a transaction with the input and output. We generate the hash that will be used to produce the signatures. Then we will add the Witness Stack. For running the claiming branch of the script we must end the unlocking script by the boolean value `TRUE`.

```
1  const signatureHash = txRaw.hashForWitnessV0(0, p2wsh.redeem!.output
       !, totalBalance, hashType);
2
3  const witnessStack = bitcoin.payments.p2wsh({
4    redeem: {
5      input: bitcoin.script.compile([
6        bitcoin.script.signature.encode(receiverKeyPair.sign(
             signatureHash), hashType),
7        receiverKeyPair.publicKey,
8        Buffer.from(preimage, 'hex'),
9        bitcoin.opcodes.OP_TRUE
10     ]),
11     output: witnessScript
12   },
13 }).witness;
```

Listing 7.9. Preparation of the transaction to claim the funds.

7.8 Refunding transaction

For having a valid refunding transaction, the timelock must be reached. To start, initialize the unlocking script with the witness script from the swaplock. Then we must prepare a transaction with the input and output and the timelock used (nSequence).

```
const signatureHash = txRaw.hashForWitnessV0(0, p2wsh.redeem.output,
    totalBalance, hashType);

const witnessStack = bitcoin.payments.p2wsh({
redeem: {
    input: bitcoin.script.compile([
    bitcoin.script.signature.encode(senderKeyPair.sign(signatureHash)
        , hashType),
    senderKeyPair.publicKey,
    bitcoin.opcodes.OP_FALSE
    ]),
    output: witnessScript
},
}).witness;

txRaw.setWitness(0, witnessStack);
```

Listing 7.10. Preparation of the transaction to spend refund.

7.9 Broadcast a transaction into the network

Once a transaction is built, we can get her in the `Hexadecimal form` (the only way to push a transaction). But this transaction for the moment is only known by us, we need to share it into the network for the miners confirm it and include it into a block. With the block explorer `chain.so`, its REST API offers us a POST request to push a transaction in the Testnet. See listing 7.11.

```
async function broadcastTx(txHex: string) {
    try {
        $.post('https://chain.so/api/v2/send_tx/' + APINETWORK, "
            tx_hex=" + txHex).done(() => {
            swal.fire("Transaction bitcoin broadcasted")
        });
    } catch (error) {
        console.log(error);
    }
}
```

Listing 7.11. Function to broadcast a raw transaction.

Chapter 8

Implementation in Ethereum with Solidity

For this project, we describe a very simple method of constructing the protocol in the Ethereum smart contract. The implementation is spread into three main components (i) a `description of the function lock`, (ii) `description of the function unlock`, (iii) `description of the function refund`. The current implementation is ready for a demonstration, however, for the production, we need to add a coverage test phase for assuring the complete deployment. This chapter refers to the implementation available on GitLab at `https://gitlab.com/Skogarmadr/atomic-swap/tree/master` of the writing of this report. Note that the sources may change or evolve from the last version of the code. So, the last version is updated in the GitLab.

8.1 Configuration

For constructing the smart contract in Ethereum, we use the high-level language `Solidity`. The version of the solidity required is the **0.5.9**. We use also a framework to facilitate the building of smart contracts, the compilation and the deployment of them. This framework is the `Truffle Framework` Truffle is a NodeJS module. We use `Ganache` that is a program for emulating an Ethereum local wallet that generates us 10 addresses which each one has a balance of 100 ethers. It is a useful tool for testing our DApp quickly because we can incorporate these addresses into our software wallet `Metamask`.

8.2 Deployment

The contract when is fully tested in local can be deployed on a Testnet network to simulate a real use case instead of directly pushing it onto the mainnet. The Testnet used is `Ropsten` but we can also use other testnet.

8.3 Lock Function

```
1     function lock (address payable _receiver, bytes32 _hashlock, uint
          _block)
2         external
3         payable
4         noBusy
5         minimumFund
6         differentReceiver(_receiver)
7     {
8         ownerToContract[msg.sender] = LockContract(
9             msg.sender,
10            _receiver,
11            msg.value,
12            _hashlock,
13            block.number + _block
14        );
15
16        busyAddresses[msg.sender] = true;
17
18        emit NewLockContract(msg.sender, _receiver, msg.value,
              _hashlock, block.number + _block);
19    }
```

Listing 8.1. Implementation of the function lock.

The listing 8.1 describes how is implemented the function for locking `lock` funds into the smart contract. First we need parameters for validate data. We uses the type `bytes32` for the hashlock instead of using a string. This is because bytes take less bytes than the variable string at the compilation. Thus that allows to save fees. For being allowed to execute this function, we need to verify if the sender is not busy which means that he hasn't already a contract swap in progress. He need to send funds otherwise the function revert. And the receiver cannot be the the sender, therefore we need to verify the addresses. When all theses restrictions are verified, we can create a new `LockContract` and the address of the sender to true saying he can't call this function unless his swap is claim or refund.

8.4 Unlock function

```
1  function unlock(address _owner, bytes32 _preimage)
2          external
3          onlyReceiver(_owner)
4          busy(_owner)
5          hashlockMatches(_owner, _preimage)
6          claimable(_owner)
7          returns (bool)
8      {
9          LockContract memory c = ownerToContract[_owner];
10
11         busyAddresses[_owner] = false;
12
13         c.receiver.transfer(c.amount);
14
15         emit FundClaimed(c.receiver, _preimage, c.amount);
16         return true;
17     }
```

Listing 8.2. Implementation of the function unlock.

The listing 8.2 describes how is implemented the function `lock` for unlocking the smart contract and claim the funds. Only the the receiver of the contract can call this function for the security of the program. We need to check if the contract exists before. Then we check if the function can be claimable. For that we check the value is equal to the hashlock and the timelock is not reached. When all these conditions are fulfilled, the funds are transferred to the receiver of the LockContract. And we free the owner of the contract. A new swap can start again.

8.5 Refund function

```
1   function refund() external onlySender busy(msg.sender) refundable
        returns (bool) {
2          LockContract memory c = ownerToContract[msg.sender];
3
4          busyAddresses[msg.sender] = false;
5          c.sender.transfer(c.amount);
6
7          emit RefundSpent(c.sender, c.amount);
8
9          return true;
10     }
```

Listing 8.3. Implementation of the function refund.

The listing 8.3 describes how is implemented the function `refund` for spending the funds

of the smart contract and get the funds back. Only the the owner of the contract can call this function for the security reason. We need to check if the contract exists before. Then we check if the function can be refundable. For that we check if the timelock is reached. When all these conditions are fulfilled, the funds are transferred to the receiver of the LockContract. And we free the owner of the contract. A new swap can start again.

8.6 Interact with the contract

For executing the contract code in our DApp, we need to use Web3 Javascript Library. web3.js is a collection of libraries that allow us to interact with a local or remote ethereum node, using an HTTP or IPC connection. The web3 JavaScript library interacts with the Ethereum blockchain. It can retrieve user accounts, send transactions, interact with smart contracts. The library used for the project is Web3x, a port of web3.js to TypeScript with a focus on tiny builds (~150k uncompressed), perfect types and clean modular code. The purpose of using web3x instead of web3 is to simplify the development.

Chapter 9

Results

In this chapter, we analyze the benefit of the atomic swap in comparison standard platform of exchange. We analyze also certain result of the implementation that can be review for an improvement.

9.1 Benefits of the Atomic swap

9.2 Limitation of the atomic swap

9.3 Problem of the block explorer

The project is functional by using the combination of centralized and decentralized components. For getting the data from the blockchains, we need to use a block explorer. A block explorer is a centralized tool that provides a view of cryptocurrency transactions online. UNfortunately this technic has some issues, the main issue is that we are dependant of the block explorer. That means, if we cannot get information from it, our program is unusable. To resolves this issues for getting the data, we can run our node of each blockchain in a local server. It is the best approach for being decentralized and especially not depending from a third party. However we sacrifice the user-friendly concept of our program, because that's need for each user who wants use our program to install and run also a node in local into his computer. This step can complicated for some users and even they can be lost for using our programs. It is why we have prefer to keep block explorer, the users has the minimum of task and we conserve the interface user-friendly.

Chapter 10

Discussion

The cs

10.1 Conclusion

10.2 Perspective

Appendix A

The `devdoc` & `userdoc` in Raw Markdown Of The HashedTimelockContract

```
- [HashedTimelockContract](#hashedtimelockcontract)
  - [*function* ownerToContract](#function-ownertocontract)
  - [*function* refund](#function-refund)
  - [*function* busyAddresses](#function-busyaddresses)
  - [*function* unlock](#function-unlock)
  - [*function* lock](#function-lock)
  - [*event* NewLockContract](#event-newlockcontract)
  - [*event* FundClaimed](#event-fundclaimed)
  - [*event* RefundSpent](#event-refundspent)

# HashedTimelockContract

Luca Srdjenovic <luca.srdjenovi@gmail.com> Cross chain Atomic-Swap This contract referring the protocol BIP 199 to exchange Ether for Bitcoin in Ethereum.


## *function* ownerToContract

HashedTimelockContract.ownerToContract() `view` `41b0b808`


Inputs

| **type** | **name** | **description** |
|-|-|-|
| *address* |  | undefined |


```

```
28
29 ## *function* refund
30
31 HashedTimelockContract.refund() `nonpayable` `590e1ae3`
32
33 **For getting the refund, the function must be refundable, timelock >
       block.number**
34
35 > Refund the sender when the timelock is reached.
36
37
38
39 Outputs
40
41 | **type** | **name** | **description** |
42 |-|-|-|
43 | *bool* |  | undefined |
44
45 ## *function* busyAddresses
46
47 HashedTimelockContract.busyAddresses() `view` `6e77860d`
48
49
50 Inputs
51
52 | **type** | **name** | **description** |
53 |-|-|-|
54 | *address* |  | undefined |
55
56
57 ## *function* unlock
58
59 HashedTimelockContract.unlock(_owner, _preimage) `nonpayable` `785fd544`
60
61 **For unlocking the contract, the caller must be the receiver and
       timelock < number.block**
62
63 > Unlock the contract with the address of the caller. This function
       allows to withdraw the balance of the contract by the receiver's
       address.
64
65 Inputs
66
67 | **type** | **name** | **description** |
68 |-|-|-|
69 | *address* | _owner | The address of the recipient of the balance. |
70 | *bytes32* | _preimage | The preimage to unlock. |
71
72 Outputs
73
74 | **type** | **name** | **description** |
```

```
|-|-|-|
| *bool* |  | undefined |

## *function* lock

HashedTimelockContract.lock(_receiver, _hashlock, _block) `payable` `a80de0e8`

**Lock `a new LockContract` from the account of `message.caller.address()` with the differents parameters.**

> Create a new LockContract and lock it.

Inputs

| **type** | **name** | **description** |
|-|-|-|
| *address* | _receiver | The address of the recipient of the Ethers. |
| *bytes32* | _hashlock | The hashlock of the contract. |
| *uint256* | _block | The number to add for the timelock. |

## *event* NewLockContract

HashedTimelockContract.NewLockContract(sender, receiver, amount, hashlock, timelock) `4012c4df`

Arguments

| **type** | **name** | **description** |
|-|-|-|
| *address* | sender | indexed |
| *address* | receiver | indexed |
| *uint256* | amount | not indexed |
| *bytes32* | hashlock | not indexed |
| *uint256* | timelock | not indexed |

## *event* FundClaimed

HashedTimelockContract.FundClaimed(receiver, preimage, amount) `d9546f4c`

Arguments

| **type** | **name** | **description** |
|-|-|-|
| *address* | receiver | indexed |
| *bytes32* | preimage | not indexed |
| *uint256* | amount | not indexed |

## *event* RefundSpent

HashedTimelockContract.RefundSpent(owner, amount) `6e6abc5f`
```

```
123
124 Arguments
125
126 | **type** | **name** | **description** |
127 |-|-|-|
128 | *address* | owner | indexed |
129 | *uint256* | amount | not indexed |
130
131
132 ---
```

Listing A.1. The `devdoc` and `userdoc` in `markdown` format of the HashedTimelockContract, generated by `solmd`.

Appendix B

External files

B.1 Specifications of the project

B.2 Planning of the project

B.3 Logbook of the project

List Of Abbreviations

BIP Bitcoin Improvement Proposal. 7, 21, 23

DApp Decentralized Application. 9, 13, 45, 48

EC Elliptic Curves. 15, 38

ECC Elliptic Curves Cryptography. 11, 15, 16

ECDSA Elliptic Curve Digital Signature Algorithm. 27, 30, 36

EVM Ethereum Virtual Machine. 9, 10

HD Hierarchical Deterministic. 36

HTLC Hashed Timelock Contracts (HTLC). 20

P2PKH Pay To Public Key Hash. 5, 6

P2SH Pay To Script Hash. 7, 30

PKH Public Key Hash. 30, 36

SegWit Segregated Witness. 7, 38, *Glossary:* Segregated Witness

UML Unified Modeling Language. 32

UTXO Unspent Transaction Output. 4, 5, 9

WIF Wallet Import Format. 38

Glossary

Segregated Witness Segregated Witness is an update to the Bitcoin software, designed to fix a range of serious issues such as solving transaction malleability, a well-known weak spot in Bitcoin software and improving scalability. 26

Unspent Transaction Output (UTXOs) UTXO is an unspent transaction output that can be spent as an input in a new transaction. 39

Web3 Web3 often refers to `web3js`, the Javascript implementation of the Ethereum JSON-RPC. It may also refer to other implementation in different languages. Overall it is the technology aiming to build the next and more decentralised version of the web 2.0 we know today. 48

Bibliography

Sean Bowe and Daira Hopwood. Hashed time-locked contract transactions, Mars 2017. URL https://github.com/bitcoin/bips/blob/master/bip-0199.mediawiki.