An Analysis of Atomic Swaps on and between Ethereum Blockchains using Smart Contracts

Research Project I
Security & Network Engineering

Bennink, Peter
University of Amsterdam
peter.bennink@os3.nl

Gijtenbeek, Lennart van
University of Amsterdam
lgijtenbeek@os3.nl

Supervisors:
Deventer, Oskar van
oskar.vandeventer@tno.nl
Everts, Maarten
maarten.everts@tno.nl

February 11, 2018

Abstract

This research study analyzes the different methods that exist to execute atomic swaps on and between Ethereum blockchains. Both coins and tokens have been covered to account for every combination of single-chain and cross-chain swaps. We have designed and developed single-usage swap contracts that require a new contract to be published for every swap. These contracts are compatible with the ERC-20 token standard, and can thus be applied to a wide range of use cases. The reason these contracts are considered reliable and provide an atomic swap is explained in detail. To increase the scalability of the atomic swap process, we have also investigated whether or not creating reusable smart contracts is a possibility. Reusable contracts come with additional complexity to achieve correctness of the atomic swap, since they must be able to consistently perform swaps between different clients at the same time, without these swaps interfering. Besides a general analysis on reusable contracts, we have designed and implemented a reusable single-chain token swap contract as a proof of concept. For the other cases of atomic swaps, we found theoretical solutions which we discuss in the report. Based on our analysis and experiments we conclude that atomic swaps and projects that implement them (i.e. decentralized exchanges) have a good chance of succeeding. The usage of smart contracts has proven to be a good method to implement this functionality.

Contents

1 Introduction

Due to the popularity of cryptocurrencies, the number of coins and tokens as well as blockchains is steadily increasing. Currently, third parties are needed to exchange coins and tokens between wallets. Both parties send their funds to the exchange, and the exchange then sends them to the opposite party. Exchanges typically charge a fee for this. They are the trusted third party, but in recent years these exchanges have on multiple occasions in some way broken this trust [12, 20, 19, 2]. The centralization of these exchanges stands in stark contrast to the typically decentralized blockchain technology, which makes them a major weakness.

The concept of atomic swapping has been receiving a lot of media attention, since it aims to resolve the problem explained above [11]. An atomic swap is an exchange of funds (coins and/or tokens) that either happens completely or not at all. This happens without the funds going through a third party, which means that the centralized exchanges seen today would be playing a different role. They would only negotiate transactions between clients, but the actual transaction of funds would happen between the two clients. One could argue that this increases the decentralization (and therefore security and reliability) of the entire blockchain infrastructure.

In this research paper, the newest developments regarding atomic swaps are investigated. We performed an analytical study of the different possibilities for executing atomic swaps with respect to tokens and coins. With the knowledge gained from this study we developed our own atomic swap implementations. The main focus is the Ethereum blockchain [14, 6], as this is currently the biggest blockchain-based platform in terms of the number of different active tokens. [1] The two types of atomic swaps that we investigate are swaps on a single blockchain (single-chain), and swaps between two separate blockchains (cross-chain). Within these two categories we can further identify different cases of atomic swaps based on the two types of funds that are swapped, namely coins or tokens. This gives us the following five different types of atomic swaps: single-chain token swaps, single-chain coin/token swaps, cross-chain coin swaps, cross-chain token swaps, and cross-chain coin/token swaps. Single-chain coin swaps are not relevant, as any blockchain can contain one coin at most, which means that this type of swap does not exist. Our implementation makes use of smart contracts published on the blockchain to conduct the atomic swap between the two parties.

The results of the investigation are positive. We were able to perform an atomic swap for all five situations, by using smart contracts to act as the third party. The main contribution to this field of research is the design and implementation of a single-chain atomic swap for coins and tokens, as well as the extension of the *Hashed TimeLocked Contract* (used in cross-chain swaps) to also function with tokens. This type of contract will be explained in detail in section 3.1.3. We argue that for these different scenarios the process is indeed

[1] Cryptocurrency Market Capitalizations - Tokens

atomic and safe to conduct for both parties. These contracts are only valid for that specific transaction, and will have to be deployed again for every transaction. A general investigation into the intricacies that come with designing a correct, secure and scalable reusable contract has also been done. For one type of swap (the single-chain token swap) we succeeded in creating a proof of concept contract that can be used repeatedly and concurrently.

2 Objectives

We conducted this research in cooperation with TNO (Nederlandse Organisatie voor Toegepast Natuurwetenschappelijk Onderzoek, English: Dutch Organisation for Applied Scientific Research) [10]. TNO has launched their own Ethereum blockchain called Techruption [2], which is based on the Quorum blockchain [7, 9]. The goal of this project is to get a better understanding of the current state of atomic swaps in blockchain technology. The main research question of this project is *'Are there reliable methods for making atomic swaps on and between blockchains?'*.

The Ethereum platform currently hosts the most tokens. As of the 31st of January 2018, coinmarketcap.com lists 577 tokens, 470 of which are based on the Ethereum blockchain[3]. Therefore, to maximize the practical use of this project we have chosen to investigate atomic swaps on this blockchain. The Ethereum code has also been forked to create other blockchains which can make use of contracts built for the Ethereum blockchain [7]. As such, this project will also look into cross-chain transactions.

2.1 Methodology

This research study consists of two main parts: a literature study on the concept of atomic swaps and the progress made in implementation, and the actual implementation of atomic swaps on the Ethereum blockchain. Based on the knowledge gained from the literature study, we research into the possibilities smart contracts have to support the atomic swaps for the five possible scenarios listed below:

1. Atomic on-chain token swaps
2. Atomic on-chain coin/token swaps
3. Atomic cross-chain coin swaps
4. Atomic cross-chain token swaps
5. Atomic cross-chain coin/token swaps

[2] Techruption - techruption.org
[3] Cryptocurrency Market Capitalizations - Tokens

The cross-chain cases appear more difficult, since here the atomic swap is conducted between two blockchains that cannot directly communicate with each other, which means we need to find a solution that makes sure all required actions can only happen in a certain order, and neither of the two clients nor an outsider could in some way misuse the contracts in such a way that anyone loses their ownership. The off-chain communication that is necessary to perform the atomic swap between the two clients (e.g. via a decentralized exchange) is out of the scope for this project.

3 Literature study

In this literature study we aim to find out more about the context of atomic swaps and Ethereum to get an idea of what has already been done, and how. Besides that, we also touch upon two decentralized exchanges that are currently in development, to get a better idea of the real world uses for atomic swaps.

3.1 Ethereum

Ethereum is the name of the cryptocurrency platform created by Vitalik Buterin in 2015 [14, 6]. The goal was to create a blockchain-based platform with a scripting language that can be used for application development, something Bitcoin largely lacked [21]. Bitcoin does have a scripting language, Script, but it is too basic to perform more complex tasks. The native currency on the Ethereum blockchain is called Ether (ETH).

3.1.1 Smart contracts

Smart contracts are programs that can be published on the Ethereum blockchain. The main scripting language Ethereum uses is called Solidity, a 'contract-oriented, high-level language for implementing smart contracts, influenced by C++, Python and JavaScript and designed to target the Ethereum Virtual Machine (EVM)' [4]. Similar to wallets, smart contracts have a public address and an Ether balance. However, they do not have a private key associated with them. Smart contracts are able to send funds to other parties on the blockchain, and they are also able to receive funds. For example, the smart contract could perform transactions of ETH or Ethereum tokens, based on certain conditions.

For a small (variable) amount of ETH (called Gas), every node has the ability to deploy a smart contract on the blockchain. When a smart contract is executed, miners will verify the transaction in order to reach consensus about the global transaction history. To execute a write function call to a smart contract, the caller also pays a fee in Gas. Thereby, continuously executing code to DoS the miners on the network is not as easily done as on the Internet, since it would cost the attacker too much funds to perform the attack.

[4] Solidity - ReadTheDocs: https://solidity.readthedocs.io/en/develop/

Once a smart contract is deployed, the code is immutable and can be open-sourced, which makes the contract fully transparent. Since the code is static from the point that the contract is published, it is important to make sure code works as it is supposed to and is bug-free. In order to test smart contracts, developers can first pre-publish them on the Ethereum test networks (e.g. Rinkeby and Ropsten). Smart contract functions can be called upon by other contracts and/or wallets, unless it self destructs rendering it unusable. The amount of Gas it costs to perform instructions on a smart contract depends on the complexity of the instructions. This is a constant factor; it is implemented in this way to decouple the instruction costs from the Ether price at that moment. Ethereum smart contracts have a wide range of applications [8].

3.1.2 ERC-20 Token Standard

A popular application of the smart contract is to publish tokens on the Ethereum blockchain. A token is a subcurrency or some other unit of value, that exist on top of the existing blockchain next to its native currency. One of the reasons to deploy a token on the Ethereum blockchain instead of creating a whole new blockchain is the fact that a lot of use cases for tokens have requirements that are met by the Ethereum blockchain. Setting up a new blockchain to include these requirements is often not necessary, involves more work, and leaves users with multiple separate blockchains to synchronize.

The ERC-20 Token Standard defines a default setup to implement tokens on the Ethereum blockchain [23]. In order to conform to this standard, one must implement a set of basic functions in the smart contract. The ERC-20 standard allows for a smoother integration of tokens with third party software, as well as other tokens and smart contracts. This is the very reason the implementation-part of this project will focus on compatibility with the ERC-20 standard.

In order for a smart contract to qualify as an ERC-20 token, the following functionality must be implemented:

- Return the total supply of tokens.
- Request the amount of tokens a specific wallet holds.
- Transfer a number of tokens from the caller to a specified destination.
- Approve another wallet to spend a certain amount of tokens on behalf of the caller.
- Return the amount of tokens a wallet is allowed to spend on behalf of another wallet.
- If the caller is approved by the spender, transfer a number of tokens from the spender to a specified destination.

The implementation of the ERC-20 standard used in this study can be found at the Git repository for this project[5].

[5] Git repository: https://github.com/clvang000/SNE_TNO_RP1

3.1.3 Atomic swaps

Atomic swaps for blockchains is a new subject that is currently being researched by a number of separate parties [11]. An atomic swap can be defined as a transaction between two parties that does not depend on a third party, for instance a centralized exchange, and either happens in full, or not at all. The reason this is preferred is the fact that trusting a third party is a risk, as that third party could, accidentally or on purpose, leave one or both clients without their funds [12, 20, 19]. What makes the Ethereum blockchain particularly suitable for atomic swaps is the fact that it supports smart contracts, and the fact that Solidity, a language used to write these contracts, is Turing-complete.

A method used to implement cross-chain atomic swaps with contracts has existed for a few years already. These contracts are known as *Hashed TimeLock Contracts (HTLC)* [22]. The idea is that both contracts (one on each blockchain) store the hash of a secret key initially only known to one of the two parties (client A from now on). Both parties publish a swap contract on the two different blockchains and commit to locking their funds in the contracts for a predefined amount of time. The time limit on the contract on which client A deposits their funds is longer than that of the contract deployed by client B. Only after this time limit has passed can they request a refund. Client A deposits their funds first, and only after client B has checked that this deposit is right, they deposit as well. Only when a claim-request *specifying the secret key* is sent to the contract, will the contract transfer the funds. The swap contract verifies the key by generating a SHA-256 hash from it and checking whether it equals the hash that is hard-coded into the contract. Client A can now use the key to claim the funds put up by client B, an action that makes the key publicly available. Client B can then claim the funds from the swap contract put up by client A, using the secret key.

Because the time limit on the contract that client A published is longer than that on the contract client B published, there is no chance that client A can simultaneously claim client B's funds and refund their own (because a refund can only be requested when the time limit has passed). If client A does *not* claim their funds, client B cannot either, and if client A *does* claim their funds, client B automatically can as well. This is exactly what an atomic swap entails.

3.2 Decentralized exchanges

The main advantages of decentralized exchanges over centralized exchanges are that you have control over your funds, and there is less risk of loss of funds if the exchange is hacked. Using atomic swaps, these exchanges would no longer need to be part of the actual transfer, and only negotiate the transfer between two parties. There are several new projects that are currently under development, and aim to set up a decentralized exchange in the near future. Since these projects are still under development, we only briefly discuss two of such projects below, to show the contribution atomic swaps can make to the subjects of blockchain and cryptocurrencies. As future research, we could perform an

in-depth analysis of these exchanges once they are deployed.

Altcoin.io This upcoming exchange [1] is described on their website as "A truly decentralized cryptocurrency exchange. Powered by Atomic Swaps.". Besides the decentralized nature of the coin and token transfers initiated via this exchange, the front-end of the exchange is also implemented with IPFS (a distributed hypertext protocol)[13], adding an additional layer of decentralization and making it harder to target with DDoS attacks. Their main focus is the Ethereum blockchain, but it is also meant to swap funds with other blockchains, such as Bitcoin. On 7 October 2017, the team behind altcoin.io was the first to perform an atomic swap for the Ethereum between Bitcoin blockchains [5].

Web3.js is an Ethereum JavaScript API that is used for a lot of these types of projects, including the web-wallet[6] and the online Solidity editor[7] of the Ethereum Foundation. This API allows web-based Ethereum tools to interact with local wallets, and Altcoin.io will seemingly also be using this for their platform.

Blockport.io Blockport is an upcoming project that aims to provide a hybrid cryptocurrency exchange that combines aspects from both centralized and decentralized exchanges [3, 4]. The Blockport whitepaper states that their "intuitive trading platform reduces counterparty risk, transaction fees and vulnerability to fraudulent activities. We aim to provide liquidity, security, transparency and an improved user-experience."

4 Design and Implementation

In this section we explain the implementation of each type of atomic swap based on the findings in section 3. These implementations have been tested on the Rinkeby and Ropsten Ethereum test networks, which are provided by the Ethereum Foundation for development purposes [17, 16]. The processes have been split up into steps to make it as clear as possible. The swap contracts discussed in sections 4.1 and 4.2 are only used a single time for one specific swap between two clients. For the single-chain token swap we created a proof of concept for a reusable swap contract. This contract is discussed in section 4.3, where we also discuss reusable contracts for the other types of swaps.

In this chapter multiple references will be made to the smart contracts that were developed. These contracts are available at the Git repository for this project[8]. All five contracts contain the same two functions that can be called by the clients: *claim()* and *refund()*. The claim()-function is used to claim the funds on the contract. In the case of the cross-chain swaps this function requires a secret key as input (since a HTLC contract is used).

[6] Ethereum web wallet - wallet.ethereum.org
[7] Solidity editor - remix.ethereum.org
[8] Git repository: https://github.com/clvang000/SNE_TNO_RP1

While the implementation of the client-side software is not within the scope of this project, the theory behind it needs to be worked out to find out whether the proposed solution is feasible as a whole. To this end both the single-chain and the cross-chain solution include a paragraph at the end where all design choices are substantiated, as well as the reason we believe this solution to be atomic.

4.1 Single-chain

The single-chain swaps are done by having both parties send their funds to a single swap contract. The single-chain implementations depicted in Figures 1 and 2 have a certain amount of commonality at the start of the process. Therefore, these steps will be explained first, after which the subsections dedicated to the specific swaps explain the remainder of the steps for each case. Please note that in the single-chain atomic swap process, we found there to be no direct need to use a HTLC. The steps discussed below correspond to the numbers in Figures 1 and 2.

1. The clients first need to agree on the *amount of tokens* each party sends to the other, a *time limit*, and which party will deploy the contract necessary for the swap (client A from now on). This all happens *off-chain*, and the way this communication happens is not in the scope of this project. The time limit is used as a safety measure. The whole transaction has to happen within this time limit for the transaction to succeed. Only after the time limit has passed are the parties allowed to request a refund. This measure is necessary from a user experience perspective, as it makes transactions much swifter, because neither party can keep the other party waiting indefinitely.

2. When the details from the first step have been agreed upon, the parties exchange their Ethereum addresses. This happens off-chain as well.

3. Both parties now have all the information to create and deploy the contract. While only one of the clients has to actually deploy the contract, it is important that both are able to compile it, for reasons that become apparent in steps 4 and 5. In our case, *client A generates and deploys the contract*. This contract is slightly different for the two single-chain cases, but the general setup is the same. More information about the difference between tokens and coins can be found in section 3.1.

4. Client A sends the address of the swap contract it published, the source code for the contract, the compiler used, and the arguments used to client B.

5. Client B compiles the contract themselves using the available information, and *compares it* to the contract on the chain. This is an easy way to make sure that the contract that was deployed was exactly what both parties agreed upon.

6. Client B confirms to client A that they accept the contract as it was deployed. If the contract was not as expected, the swap can either be stopped completely, or re-tried. This is more related to the user experience than to atomic swap process, and happens completely off-chain, thus it does not fall not within the scope of the project.

This is the point where the token swap and the coin-token swap start to differ in their implementation. The following two subsections are thus continuations of steps 1-6 explained above.

4.1.1 Single-chain token swap

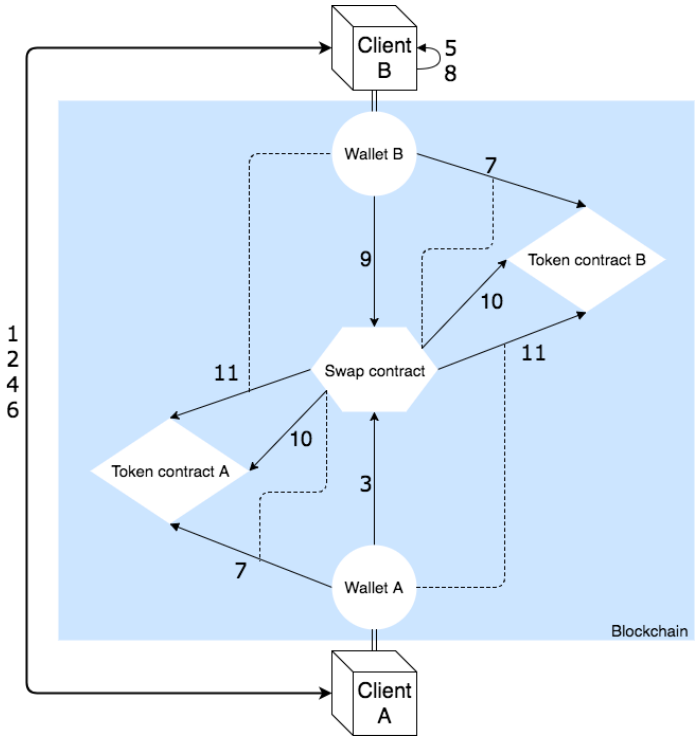

Figure 1: Single-chain token swap

7. At this point it is safe for both parties to send their tokens to the contract. This happens in the following way: client A executes *transfer()* on token contract A, in which they request the contract to move tokens from their own wallet to the swap contract. Client B does the same with token contract B. The order in which this happens does not matter, as the contract does not execute any transaction until all funds are on the contract.

8. One of the clients checks whether the funds have been transferred to the swap contract. This happens by indexing the blockchain and checking the transactions. Which party does this can be decided off-chain, for the purposes of this explanation client B will do this.

9. Once client B sees the right amounts on the swap contract, they send a confirmation of this to the swap contract (by calling *claim()*).

10. The swap contract checks if it has received the agreed upon funds from both clients. It has to request its balance at both of the token contracts, as it does not have that information itself.

11. If the contract concludes the funds have both been received, it will proceed to transfer the funds to the clients. It does this by executing *transfer()* on each token contract, requesting a token transfer to the corresponding wallets.

Validation This process should be atomic due to multiple factors. Firstly, once the funds are on the swap contract, only the contract itself can move the funds. Executions of Ethereum smart contract functions are atomic as a whole, thus we can be confident that any function is either completely executed or not at all [15, 14]. The two clients can call two functions on the contract to move the funds, *claim()* and *refund()*. At no point is either client capable of stealing the other client's funds. The following snippet of code demonstrates this:

```
1.  function claim() onlyParticipant public returns (bool) {
2.      uint token1_balance = token1_instance.balanceOf(this);
3.      uint token2_balance = token2_instance.balanceOf(this);
4.      if (token2_balance >= amountOf_token2 &&
5.        token1_balance >= amountOf_token1 &&
6.        now < timeOut) {
7.          token1_instance.transfer(clientB, token1_balance);
8.          token2_instance.transfer(clientA, token2_balance);
9.          selfdestruct(clientA);
10.     } else {
11.         return false;
12.     }
13. }
```

This function checks the amount of each token it owns with the respective token contracts (line 2 and 3). If these amounts are at least the amount of tokens that was agreed upon, the transfer happens. As only this contract can transfer these tokens, there is no risk of either person not receiving their tokens: the transfer only happens when both parties sent (at least) the correct amount of tokens, and then one of them calls *claim()*, which sends the funds to both parties at once. The reason we chose to send all tokens (even if either party has sent more than was agreed upon) is the fact that otherwise those tokens will get lost. As

all addresses involved in the transaction are hardcoded in the contract, outsiders are not be able to re-route the funds when claimed or refunded. The contract self-destructs after the transfer has happened (line 9), meaning that from that point on interaction with the contract is not possible anymore.

The single-chain swaps do not require a secret key (as used with HTLCs) because there is only one contract. The cross-chain swaps have two contracts that should only allow certain functions to be executed once certain steps have been executed on the other contract. Because the chains cannot see what happens on the other chain, the key is necessary to make sure that clients do not even have the option of executing functions out of order, which might result in stolen funds.

Aside from the atomicity, the most important part of the whole process from a security-perspective is the client-side contract confirmation. This part of the process makes sure that neither party can deploy a contract different from what the other party expects. This is very similar to the contract confirmation that Etherscan has in place [9]. The individual that deployed the contract can send the source code and other relevant information to Etherscan via their website [10], so that Etherscan can then compile the contract under the exact same circumstances. If their compile is the same that is found on the chain, the contract is verified. This recompile should happen locally.

4.1.2 Single-chain coin-token swap

While the differences between the tokens swap and the coin-token swap are minor, they are significant enough to cause confusion if not highlighted. The main difference is the fact that the swap contract does not have to do an external call with a token contract whether the funds have been deposited (for the Ether).

[9] Etherscan.io - Smart Contract of OmiseGo - https://etherscan.io/token/OmiseGo#readContract
[10] Etherscan.io - Verify Contract - https://rinkeby.etherscan.io/verifyContract

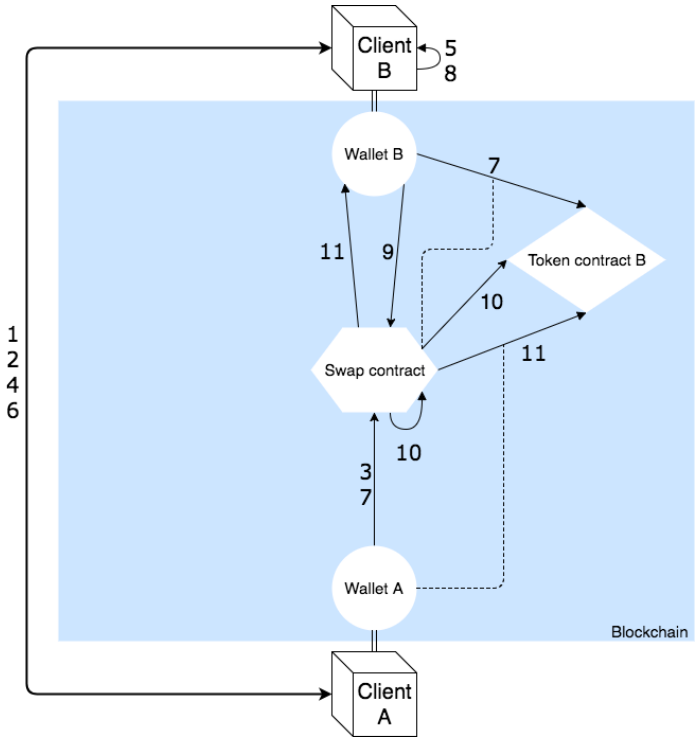

Figure 2: Single-chain coin-token swap

7. Client A sends coins to the swap contract. Client B executes *transfer()* on token contract B, in which they request the contract to move tokens from their own wallet to the swap contract. The order in which this happens does not matter.

8. Client B checks whether the funds have been transferred to the swap contract.

9. Once they see the right amounts on the swap contract, they send a confirmation of this to the swap contract (by calling *claim()*).

10. The swap contract checks that it has received the agreed upon funds from both clients. It has to ask token contract B what its balance is, as it does not have that information itself. For the coin it can just check its own balance without making any further requests to outside sources.

11. If the contract concludes the funds have both been received, it will proceed to transfer the funds to the clients. For the token transfer, it does this by executing *transfer()* on the token contract that is involved. The coins can just be sent in the regular way.

4.2 Cross-chain

The swap method for the cross-chain atomic swaps uses the Hashed TimeLock Contract explained in the literature study. The basis for the code used in these swaps comes from previous research [11]. We have extended and implemented this method to be fully compatible with ERC-20 compliant tokens. As with the single-chain swaps, the cross-chain atomic swap implementations depicted in Figures 3, 4 and 5 show a certain amount of commonality. These steps will thus be explained first, after which the subsections dedicated to the specific swaps explain the remainder of the steps.

1. The clients first need to agree on the *amount of tokens* each party sends to the other and the two *time limits*. These time limits are used as a safety measure. One time limit must be significantly longer than the other. This step happens off-chain.

2. One of the clients (client A from now on) chooses a *secret key*, and then hashes it. This hash is what makes sure that at no point in the transaction either user can steal the other parties funds.

3. The two clients exchange their Ethereum addresses, and client A also sends the hash of the secret key to client B.

4. Both clients then create and deploy a swap contract on 'their' blockchain incorporating the time limits, the Ethereum addresses and the hash of the secret key. Client A generates the contract on blockchain A, and client B generates the contract on blockchain B. It is important that the contract made by client A gets the longer time limit.

5. Now that the swap contracts have been generated, the clients exchange the contract addresses, the source code for the contracts, the compiler used, and the arguments used. This step happens off-chain.

6. Both clients compile the contract the other party made, and *compare it* to the contract that was published on the blockchain. This is done to verify whether the deployed contracts contain what was agreed upon.

7. Assuming the contract is as expected, the clients send each other a confirmation and the process can continue. If there is an issue, the whole process is to be repeated. This step happens off-chain.

This is the point where the three swaps start to differ in their implementation. The following subsections are thus continuations of steps 1-7 discussed above.

[11] HTLC cross-chain coin swap contract - Github

4.2.1 Cross-chain coin swap

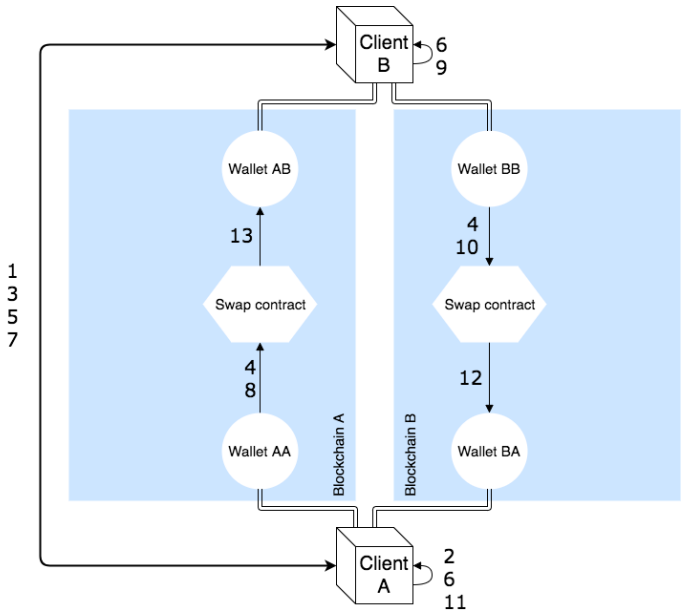

Figure 3: Cross-chain coin swap

8. Client A sends their funds to swap contract A. It is important for client A to be the first to deposit their funds, for reasons that will become apparent in the next steps.
9. Client B has access to the blockchain, and can thus see when and how much client A has sent to swap contract A. This happens by indexing the blockchain and checking the transactions. If this is the amount the parties agreed upon, the process can continue.
10. Client B transfers their funds to swap contract B.
11. Client A checks whether the transaction has succeeded and whether it was the right amount. This happens locally.
12. Client A now calls the *claim()* on contract B using the secret key to claim the funds. In the process of claiming the funds, the secret key is sent in plaintext to contract B, and from then on is available on the blockchain for anyone to see, including client B.
13. Client B uses the secret key to *claim the funds* on swap contract A. The destination address in the swap contract is hard-coded, which means that it does not matter that the secret key is no longer secret; even if someone else would use it to call the *claim()*, the funds would still be sent to client B.

Validation The validity of the cross-chain swap contracts is again confirmed by recompiling the contracts off-chain and comparing them to the versions that were published on-chain. The main difference with respect to the single-chain swaps is the fact that it requires client-side input during the swap, i.e. retrieving the key from blockchain B and then using it to claim the funds on blockchain A. This is because the two chains cannot execute things on the other chain. This client-side input can happen anywhere between the moment client A claims their funds and the moment swap contract A expires, which should result in no loss of funds. However, client side input halfway through the swap does leave open the possibility of non-atomicity. Alternatives to the HTLC should be considered to solve this problem, as this is not a consequence of the implementation, but of the theory of the HTLC, which depends on user input during the swap. Assuming client B is able to execute *claim()* on the contract on blockchain A, the method using the secret key appears to be secure, as the secret key stays locally on the machine of client A until they claim their funds.

4.2.2 Cross-chain token swap

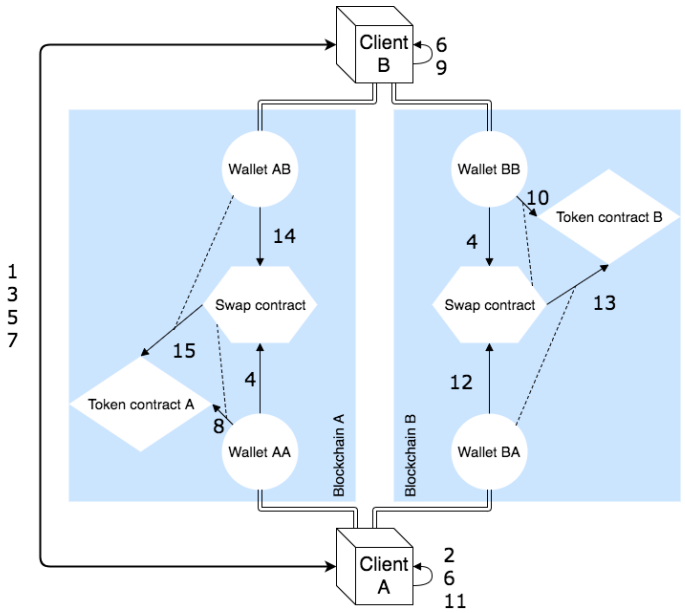

Figure 4: Cross-chain token swap

8. Client A calls *transfer()* on token contract A such that their tokens are transferred to swap contract A. It is important for client A to be the first to deposit their funds.

9. Client B has access to the blockchain, and can thus see when and how many tokens client A has deposited to swap contract A. This happens

by indexing the blockchain and checking the transactions. If this is the amount the parties agreed upon, the process can continue.

10. Client B transfers their funds to swap contract B by calling *transfer()* on token contract B.
11. Client A checks whether the transaction has succeeded and whether it was the right amount. This happens locally.
12. Client A now calls the *claim()* on contract B using the secret key to claim the funds.
13. Swap contract B calls *transfer()* on token contract B to move its tokens to the wallet of client A.
14. The key is now freely available on the blockchain. Client B uses the key to *claim the funds* from swap contract A.
15. Swap contract A now calls the *transfer()* on token contract A to move its tokens to client A's wallet.

4.2.3 Cross-chain coin-token swap

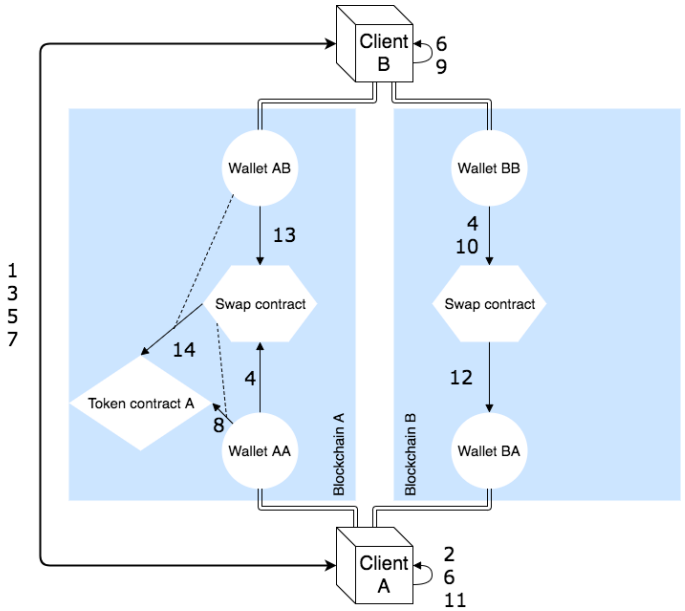

Figure 5: Cross-chain coin-token swap

8. Client A calls *transfer()* on token contract A such that their tokens are transferred to swap contract A. It is important for client A to be the first to deposit their funds.

9. Client B has access to the blockchain, and can thus see when and how much client A has sent to swap contract A. This happens by indexing the blockchain and checking the transactions. If this is the amount the parties agreed upon, the process can continue.
10. Client B transfers their funds to swap contract B.
11. Client A checks whether the transaction has succeeded and whether it was the right amount. This happens locally.
12. Client A now calls *claim()* on contract B using the secret key to claim the funds.
13. The key is now freely available on the blockchain. Client B now uses that same key to call the claim function on swap contract A.
14. Swap contract A now calls *transfer()* on token contract A to move its tokens to client A's wallet.

4.3 Reusable contracts

The different types of atomic swap implementations we have discussed use a swap contract that is used for one single swap. In the long run it is not very efficient to publish a new swap contract on the blockchain for each atomic swap. The reason being that there will be a lot of duplicate code on the blockchain, needlessly increasing its size. Deploying a new contract on the blockchain also has costs associated with it, costs that would be much lower if we were to create a contract that can be used multiple times. The ideal swap contract must therefore be designed with scalability in mind.

For both the single-chain and cross-chain variant a reusable contract that is both secure and scalable is a valuable improvement. The question of whether or not the process is fully atomic and secure for both parties is more involved than with a single-swap contract, because multiple parties could be performing swaps at the same time. We find that there are two approaches:

1. Both wallets deposit their funds (coins or tokens) to the reusable contract (similar to the single-swap contracts).
2. In case the swap involves tokens, it is also possible to approve the reusable contract to transfer tokens on behalf of a wallet (making use of the *approve()* and *transferFrom()* functions on the token contract). This is made possible because of the ERC-20 standard, and not a native option for ETH.

In the first approach, it is essential that the contract is able to check which wallets deposited which coins and/or tokens to it. Since multiple swaps may be going on in the same time frame, the contract needs a method to verify that the two wallets involved in a swap indeed send the right amount of funds. If the contract is not able to verify this, it may get things mixed up and use funds destined for another swap happening at the same time.

The standardized ERC-20 token implementation allows us to request the total amount of tokens we have in our wallet. However, in a situation where one contract is used for multiple swaps at the same time, the contract should also be aware of what funds are related to what swap. If this is not the case, a situation could arise where the contract sends funds that were deposited by a wallet not involved in the current swap. In order to remedy this situation, we find that the ERC-20 token would need to be extended. This would mean that the ERC-20 contract needs to keep track of the transactions done to the swap contract(s) such that the swap contract can query the token contract on whether or not a specific wallet has deposited sufficient funds to it to perform a swap. In order for this method to work properly, we make the assumption that there are no third parties that fund the part of any of the two parties to the swap contract.

When it comes to the amount of ETH that is deposited to the swap contract, we do not have to deal with additional token contracts. For example, one can keep track of the amount of ETH that is deposited by each wallet by extending the *function() payable* (the default function that is executed when ETH is transferred to a smart contract) in the swap contract to store this information. Similar to the proposed solution for tokens, this method also only works with additional storage of transaction data. However, in this case the problem can be solved without extending the code of the token contracts. Although not ideal, it does seem that both of these solutions (for tokens and ETH) would still be more efficient with respect to scalability than publishing a new contract for each swap.

As mentioned, the second approach only works for tokens. In this setup, the corresponding wallet(s) would need to call the *approve()* function on the ERC-20 token that it wants to transfer, to allow the swap contract to call the *transferFrom()* function on the ERC-20 token on their behalf. With respect to scalability, this method is much more efficient for tokens than the first approach explained earlier.

We have implemented a proof of concept for a reusable swap contract that uses this approach for the single-chain token swap case (case 1). This implementation follows the same general method as for the single swap contract explained before (see section 4.1.1). In our implementation, the *transferFrom()* function is called twice in the *claim()* function on the reusable contract. This could potentially lead to concurrency issues when the first call succeeds and the second fails (or the other way around). Therefore, we have made use of the *require()* construct in Solidity, for both of these *transferFrom()* function calls. This makes sure that if either of the transfers fail (e.g. due to insufficient funds and/or allowance), all changes are rolled back and the execution is aborted. This ensures the atomicity of the *claim()* transaction.

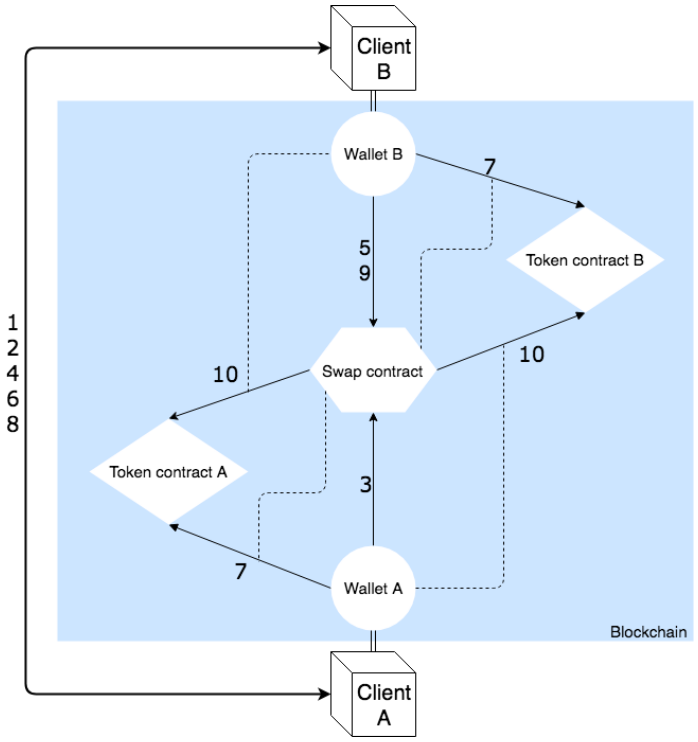

Figure 6: Visualization of the single-chain token swap

The steps for this swap are as follows:

1. The clients first need to agree on the *amount of tokens* each party sends to the other and which party will deploy the contract necessary for the swap (client A from now on). This all happens off-chain, and the way this communication happens is not in the scope of this project.

2. When the details from the first step have been agreed upon, the parties exchange their Ethereum addresses. This happens off-chain as well.

3. Client A now initiates a new swap on the contract using the *initiateNewSwap()* function. In the PoC they can choose the identifier of this swap themselves, in the final product the idea would be that the ID gets generated by the contract and returned to the client.

4. Client A then sends that ID to client B.

5. Client B can then calls *validateSwapInstance()* on the swap contract, to validate that client A has given the contract the correct input. This function returns a boolean, indicating whether or not the information is correct.

6. Client B sends client A a confirmation that they agree with the initiated swap
7. Now both parties can use *approve()* on their respective token contract to allow the swap contract to handle a specific amount of their funds.
8. Client A then sends client B an off-chain confirmation that they set the right allowance. Client B could also in intervals check the allowance client A has given the contract, or a combined version of these methods, where client A sends the off-chain confirmation, after which client B checks this on-chain. We have not decided which method is best.
9. When client B sees that client A has set their allowance, and client B has as well, they can call *claim()* on the contract, with the ID as parameter.
10. The contract then immediately tries to transfer the money from client A's wallet to client B's wallet, and vice versa. The contract uses the *transferFrom()* function on the token contracts for this. As explained, these two transactions happen either both, or not at all.

The contract is also available at the Git repository[12]. It is also worthy to note that for this contract there is no need for a *refund()* function, since the funds are not deposited to the swap contract. This is still a proof of concept, and further development is required before this contract can be used in practice. However, we feel that the main issues for performing such an atomic swap have been tackled by means of performing the swap via allowances on the ERC-20 token contracts.

5 Discussion

The atomic swap is currently a popular subject and there is still a lot of research that can be done. The suggestions for future research below are based on our own experiences during this project. Some are follow-ups to the research done in this project, others are based on things we found during the project that require more research but were out of scope.

There are several aspects of our research that can be extended. Primarily, the scalability of the swap contracts is an interesting subject. If implemented in production, we find that security and scalability are of utmost importance. Therefore, besides experimentation with reusable contracts, research into potential attack vectors and security details are also required.

5.1 Future research

5.1.1 Off-chain communication

Something that was simply outside of the scope of this project was the off-chain communication between clients that is necessary for initiating a transaction

[12] Git repository: https://github.com/clvang000/SNE_TNO_RP1

as well as exchanging data during the transaction. There are lots of ways to do this, but as this is a relatively new problem, chances are there are still improvements to be made. Web3.js, for instance, is 'a collection of libraries which allow you to interact with a local or remote Ethereum node, using a HTTP or IPC connection' [18]. This is one way for web-based decentralized exchanges to interact with wallets on client computers, which allows these exchanges to execute the steps required for such a swap in an efficient manner. The client still has full control over the actual execution, as each transaction has to be confirmed by them.

5.1.2 Attack vectors

For the cross-chain atomic swaps (HTLC), we briefly want to mention a possible attack vector here that we have come across. The question that we raise here is the following: is it possible that once both client A and B have both deposited their funds to the corresponding swap contract, client A will claim its funds from swap contract B and then make it so that client B cannot claim their funds (for instance with a DoS attack) until the timeout is expired on swap contract A, such that he/she can then call *refund()* on swap contract A, leaving client A with all the funds? It is worthy to mention that the HTLC may be slightly flawed in this respect. Next to this attack vector, it is important to research the possible other attacks on smart contracts. This would be essential in order to confidently say that the contracts we developed, or may develop in the future, are not vulnerable to any known exploits.

There are also other attack vectors related to the HTLC. For instance, client B might be able to retrieve the key generated by client A before putting their own funds on the contract on blockchain B, allowing them to pull client A's funds directly without risking their own funds at all. The client-side actions should be analyzed as well.

Another general issue with the timeout used in the swaps is the fact that the exchange rate of the tokens/coins that are being swapped will vary during the process of the swap. Based on this information either party may decide they want to stop the transaction or possibly not even send the funds to the swap contract. However, if either party has sent their funds, and the other party decides they do not want to go through with it, the first party will have to wait until the timeout has passed to refund their funds. With the volatility of cryptocurrencies this might be something to look into, possibly by shortening the timeout or creating another theory altogether.

5.1.3 Reusable contracts

As of yet, we have developed a novel way (to our knowledge) to perform an atomic swap of tokens with a reusable smart contract on the Ethereum blockchain. We believe this to be an efficient method of doing so, since no additional transaction information will need to be stored on the blockchain, in order to make the transition from a single-usage contract to a reusable contract. An interesting

future field of study, would be to also further analyze the other four cases of atomic swaps that we investigated, and see if there are any efficient implementations of reusable contracts possible. Whenever a token is involved, we can use the transfer method that we developed for the single-chain token transfer. With respect to coins, we explained that the swap contract will most likely need to keep track of its transaction history. We could investigate if applying this method for coins is 1. possible and 2. more efficient than using single-swap contracts, and if so, how much efficiency is gained with regard to scalability. The development of cross-chain reusable contracts is challenging as well and it would be an interesting subject to further investigate.

5.1.4 Atomic swaps on other blockchains

Now that we have obtained knowledge and practical experience of how to implement atomic swaps using smart contracts on the Ethereum blockchain it would also be interesting to find out what possibilities there are to implement atomic swaps on and with other blockchains.

5.1.5 Analysis of decentralized exchanges

While this project already includes an analysis of different decentralized exchanges, these projects are either not yet released, or still very new. This means that in a short while the landscape will have changed considerably. Future research could therefore analyze these projects in more detail. There are multiple aspects to these exchanges. Firstly, how decentralized are they really, ie. is the whole platform distributed, or just the financial aspect? For instance, as has been discussed in section 3.2, Altcoin.io seems to be very decentralized based on the information currently available on their website. Another aspect is security. Decentralization does not plug every security hole, and decentralized exchanges should be tested on security and reliability just as rigorously as their centralized counterparts.

6 Conclusion

In this report, we have explained the smart contracts that we developed to perform a single-chain atomic swap between two parties. We have implemented this method for swapping tokens and for swapping coins and tokens. The tokens involved in this atomic swap can by any type of ERC-20 compatible token. This can be specified by the creator of the contract, when he/she published the contract on the blockchain. Also, we have extended the already existent method of performing a cross-chain coin swap (using a HTLC) by implementing smart contracts that also function for ERC-20 compatible tokens. For both the single-chain and cross-chain atomic swaps, we have explained why we regard the process is indeed atomic and reliable. Besides the research into these single-swap contracts, we have looked into a more scalable method of performing the swaps, which is to deploy a swap contract on the blockchain only once, instead

of deploying a new contract for each swap. By extending the single-chain token swap contract, we have developed a multi-swap (reusable) contract that is much more scalable than the original contract. This reusable contract is a proof of concept and shows that such a method of performing atomic swaps is indeed possible for tokens. More research can still be done into varying subjects, such as reusable contracts and possible attack vectors. This project has shown the versatility that smart contracts have on the Ethereum blockchain. It is clearly possible to use them for a wide range of applications, including using them as a third party to oversee the process of atomic swaps.

References

[1] altcoin.io: A truly decentralized cryptocurrency exchange. https://www.altcoin.io/.

[2] Bitgrail cryptocurrency exchange claims $195 million lost to hackers. "fortune.com/2018/02/11/bitgrail-cryptocurrency-claims-hack".

[3] Blockport: The first social crypto exchange. Based on a hybrid-decentralized architecture. https://blockport.io/.

[4] Blockport Whitepaper v1.0.5. https://blockport.io/read-the/whitepaper.pdf.

[5] Ethereum Atomic Swap. https://github.com/AltCoinExchange/ethatomicswap.

[6] Ethereum Project. https://www.ethereum.org/.

[7] Quorum: Advancing Blockchain Technology. https://www.jpmorgan.com/global/Quorum.

[8] State of the DApps: 977 Projects Built on Ethereum. https://www.stateofthedapps.com/.

[9] The Techruption Blockchain Project. https://blockchain.tno.nl/projects/techruption/.

[10] TNO: innovation for life. https://www.tno.nl/nl/.

[11] Atomic Action: Will 2018 Be the Year of the Cross-Blockchain Swap?, Jan 2018. https://www.coindesk.com/atomic-action-will-2018-year-cross-blockchain-swap/.

[12] Chris Baraniuk. Bitfinex users to share 36% of bitcoin losses after hack, Aug 2016. http://www.bbc.com/news/technology-37009319.

[13] Juan Benet. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561*, 2014.

[14] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. 2014.

[15] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. *arXiv preprint arXiv:1702.04467*, 2017.

[16] Ethereum. Clique PoA protocol & Rinkeby PoA testnet Issue #225 ethereum/EIPs.

[17] Ethereum. ethereum/ropsten, Aug 2017.

[18] Ethereum. ethereum/web3.js, Jan 2018. https://github.com/ethereum/web3.js.

[19] Jim Finkle and Jeremy Wagstaff. Hackers steal $64 million from cryptocurrency firm NiceHash, Dec 2017.

[20] Robert McMillan. The Inside Story of Mt. Gox, Bitcoin's $460 Million Disaster, Jun 2017. https://www.wired.com/2014/03/bitcoin-exchange/.

[21] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[22] Joseph Poon and Thaddeus Dryja. The Bitcoin Lightning Network: Scalable off-chain instant payments. *Technical Report (draft)*, 2015. https://lightning.network/lightning-network-paper.pdf.

[23] Fabian Vogelsteller and Vitalik Buterin. ERC-20 Token Standard. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-token-standard.md.