Atomic Swapping Bitcoins and Ethers

Léonard Lys, Arthur Micoulet, Maria Potop-Butucaru

Atomic Swapping Bitcoins and Ethers

1st Léonard LYS
PALO IT
LIP6, Sorbonne University
Paris, France
llys@palo-it.com

2nd Arthur MICOULET
PALO IT
Paris, France
amicoulet@palo-it.com

3rd Maria POTOP-BUTUCARU
LIP6, Sorbonne University
Paris France
maria.potop-butucaru@lip6.fr

Abstract—Blockchains interoperability is one of the hardest problems to be solved in the nowadays blockchain ecosystem that contains thousands of different blockchains. This paper focuses on swapping assets from a blockchain to another without a trusted third party. One recent scheme for atomically swapping assets, Atomic Cross Chain Swap (ACCS), has been formally analyzed in [2]. This paper proposes an implementation of an ACCS between the two most valued crypto-currencies today: Bitcoin and Ether.

Index Terms—Atomic transactions, Atomic Cross Chain Swaps, Bitcoin, Ethereum.

I. Motivation

Let's say Alice owns some Bitcoins and Bob some Ethers. If they want to exchange their assets, and that they do not trust each other, the current solution in production will require them to trust a third party (an exchange platform) that will handle the swap.

The scenario of the swap would be as follows: Alice would have to send her Bitcoins from her wallet to an address provided by the exchange, and Bob would have to do the same with his Ethers (1). Then each of them would have to make an order (2), wait for the platform to make both orders match (3) and let the platform handle the swap (4). Finally both of them would have to withdraw their respective assets to their blockchain wallets (5).

Centralized exchange platforms as described above do not publish the transaction between Alice and Bob to the blockchain. Instead, they maintain a private ledger listing the assets that have been deposited by their clients, but everything (including transaction) is made off-chain.

As stated in the white paper of the Bitcoin protocol [3] the main benefits of a decentralized system are lost if a trusted third party is required. Indeed this raises several flaws in the system:

- Security - From stage (2) to stage (4) all crypto assets are stored inside the exchange platform's blockchain wallets. At the time of writing, the richest Bitcoin address owned by an exchange platform, holds the equivalent of 1.5 billion USD inside a single wallet [1]. This makes it an attractive target for a malicious attacker and therefore it is a major single point of failure.
- Governance - Centralized exchange platforms are sitting on a huge amount of crypto assets. By moving, buying or selling those huge mass of assets, they have the capacity to influence the markets at the expense of the people (Whale effect).

Therefore we need to find a protocol that would allow Bob and Alice to swap their assets without having to trust each other or any third party. A proposed solution for this problem are Atomic Cross Chain Swaps. It has been a discussed topic in the blockchain ecosystem as early as 2013 [5] but the first formal analysis [2] defines an ACCS as a protocol that guarantees the atomicity of an exchange. It must guarantee that no party conforming to the protocol will lose its coins(1) and that by no means a malicious party can retrieve both coins(2).

In order to comply to those properties a protocol has been proposed under the name of Atomic Cross Chain Swap. It is assumed that assets on the blockchain can be automatically stored, locked and transferred by scripts or so-called smart contracts. The described protocol relies on hashed time locked contract [4]. Those contracts allow a party A to lock some assets (or coins) on a blockchain such that they can be unlocked in two manners: by party A after a period of time Δ or by party B right away but only if party B is able to provide a secret s. By setting two similar contracts on two different blockchains, A and B can safely exchange their coins without having to trust each other or any other third party.

A hashed time locked contract (or just HTLC) can be defined by parameters h, A, B, v, t. Parameter h is the computed image of a secret $h = H(s)$, H being a cryptographic hash function. Parameter A represents the blockchain address of the swap initiator and B the one of the recipient (or participant). Parameter v is the amount of coins transferred and t is the period of time during which the assets cannot be unlocked by A. The scenario of an Atomic Cross Chain Swap between two parties, Alice and Bob, exchanging one Ether for one Bitcoin is as follows:

- (1)Alice creates a secret s and computes its image $h = H(s)$. Then, Alice locks her Bitcoins on a hashed time-lock contract with parameters h, Alice's Bitcoin address, Bob's Bitcoin address, $v = 1$, $t = 2\Delta$.
- (2)Bob looks up Alice's contract and checks that the correct recipient, amount and time-lock have been set.

- (3)Bob creates a similar HTLC on the Ethereum blockchain with parameters h, Bob's Ethereum address, Alice's Ethereum address, $v = 1$, $t = \Delta$.
- (4)Alice unlocks the ether that has been locked by Bob. Doing so, she reveals to Bob the secret s that he did not know.
- (5)With the secret Bob just unveils, he can now withdraw the Bitcoin that has been locked by Alice.

The time-lock value matters. That is, the time-lock of the party that has generated the secret s (or the initiator) has to be higher than the other one (or the participant). Indeed if the Δ has been set for the contracts to elapse at the same time, then Alice could wait for the time to elapse, withdraw the Ethers when $t \to \Delta$ and then withdraw the Bitcoins at $t > \Delta$.

If the parameters have been correctly set, at no stage of this scenario any malicious party is able to withdraw both coins. If any party halts at any stage of the protocol, the secret s will not be revealed and eventually all contracts will time out and both parties will be able to withdraw their coins.

The motivation of our work is to prove the feasibility of implementing the ACCS protocol between Ethereum and Bitcoin.

II. Model

This protocol has been described with the assumption that blockchains are temper proof and public ledgers of transactions. It is also assumed that scripts or smart contracts can be set up to trigger transfers if and only if some conditions have been met. In the following section we review how accurate those assumptions are in a real world implementation.

A. Ethereum and Solidity

Ethereum operates using accounts and balances in a manner called state transitions. We can assimilate Ethereum as a worldwide replicated state machine.

In order to transfer some Ethers one has to create and sign a transaction message and broadcast it to the network. Such message consist of the following parameters : nonce, gas price, gas limit, toAddress, value [9]. Then the message has to be signed and finally broadcast to the network.

Solidity is a high level smart contract programming language. It allows one to create smart contracts with high level programming abstractions such as structures and complex objects. Solidity code is compiled into bytecode, published on the Ethereum blockchain and then executed by the Ethereum Virtual Machine or EVM [6].

B. Bitcoin and SCRIPT

Unlike Ethereum, Bitcoin is based on a list of Unspent Transaction Output or UTXO [7].

In order to transfer some Bitcoins one has, like in Ethereum, to create a transaction message, sign it and broadcasts it to the network. But unlike Ethereum, instead of only providing the sender's address, one has to provide a list of UTXO's. In UTXO based blockchains, the ownership of a coin boils down to the capacity to spend the said UTXO.

In order to build an UTXO Bitcoin is fitted with a scripting language named Bitcoin SCRIPT [8]: a list of instruction that defines how the UTXO can be spent.

III. ACCS implementation

When implementing ACCS, it is not necessary to implement it as a pair. Indeed, if one is able to implement the logic of an HTLC on a particular blockchain, then an ACCS can be executed with any other assets that can execute the same logic.

In a two party ACCS, there is an initiator and a participant. The initiator is the one that generates the secret s and computes its $h = H(s)$. The participant is the one that setups an HTLC with parameter h but without knowing the secret s.

A swap can be in five different states depending on the stage of the protocol:

- Invalid: nothing has happened yet.
- Initiated: the initiator has created the swap and locked some funds. But the swap has no participant yet.
- Open: the swap has been created, some funds are locked in and the participant has been set.
- Closed: the participant has unlocked the funds and transferred them to his address. The secret s has been reveled, the exchange has taken place.
- Expired: one of the party halted during the protocol so the contract eventually timed out and the initiator can withdraw its funds.

A. Ethereum Implementation

We made use of the high level programming abstractions to implement an HTLC in Solidity. We created a structure meant to keep in memory the parameters of a swap:

```
struct Swap {
  uint256 timelock;
  uint256 value;
  address ethTrader;
  address withdrawTrader;
  bytes32 secretLock;
  string secretKey;
}
```

Then we created a list of states and the functions that handle the state transitions. Full code available here [10]. This is how the state transition between INVALID and OPEN is done:

As you can see, we create a new swap object and store it inside an array. Then in order to trigger state changes, checks will be made on the said stored swap. For example, to go from OPEN to CLOSED we must verify that the

```
function participate(bytes32 _swapID,address _withdrawTrader,bytes32 _secretLock,uint256 _timelock)
public onlyInvalidSwaps(_swapID) payable {
  // Store the details of the swap.
  Swap memory swap = Swap({
    timelock: _timelock + now,
    value: msg.value,
    ethTrader: msg.sender,
    withdrawTrader: _withdrawTrader,
    secretLock: _secretLock,
    secretKey: ""
  });
  swaps[_swapID] = swap;
  swapStates[_swapID] = States.OPEN;
  // Trigger open event.
  emit Participate(_swapID, _withdrawTrader, _secretLock);
}
```

participant has in fact provided the good secret s. To do so, solidity provides a programming assumption called modifiers:

```
modifier onlyWithSecretKey(bytes32 _swapID, string _secretKey) {
  require(swaps[_swapID].secretLock == sha256(abi.encodePacked(_secretKey)),
  "modifier throws : onlyWithSecretKey");
  _;
}
```

Similar functions are used to check the validity of a state transition request. Refer to full code [10].

The smart contract is deployed on the blockchain and function calls are public. We will generate the transaction thanks to an online client and then sign it with our wallet. Finally a RPC (remote protocol call) provider will broadcast the transactions to the network.

B. Bitcoin Implementation

As explained earlier Bitcoin is not fitted with high level programming language but instead with a stack-based scripting language. Thus is not possible to save inside a complex object the parameters of the swap. Here is how the state transition between INVALID to OPEN is handled:

```
  .Script()
  .add('OP_IF')
  .add('OP_SHA256')
  .add(new Buffer(hashKey.toString(), 'hex'))
  .add('OP_EQUALVERIFY')
  .add(bitcore.Script.buildPublicKeyHashOut(bitcore.Address.fromString(toAddress)))
  .add('OP_ELSE')
  .add(bitcore.crypto.BN.fromNumber(timeLock).toScriptNumBuffer())
  .add('OP_CHECKLOCKTIMEVERIFY')
  .add('OP_DROP')
  .add(bitcore.Script.buildPublicKeyHashOut(bitcore.Address.fromString(fromAddress)))
  .add('OP_ENDIF');
```

It is a simple if-else statement representing the two manners in which an HTLC can be unlocked. See full code here [10]. As explained earlier, we cannot provide a single signature for a Bitcoin transaction to pass. We first have to gather some spendable UTXO. This is done thanks to an API. Then we create and sign the transaction. Finally, we broadcast it to the network using a Bitcoin client. Once this HTLC has been set up and published to the chain, the other party has to generate a transaction in order to spend the coins. The spender provides his signature and the secret s. If both are matching the HTLC UTXO, a brand new UTXO will be created on the bitcoin blockchain and will be spendable by the spender. This is how the spend transaction is built:

```
// setup the scriptSig of the spending transaction to spend the p2sh-cltv-p2pkh redeem script
refundTransaction.inputs[0].setScript(
    bitcore.Script.empty()
        .add(signature.toTxFormat())
        .add(new Buffer(myPublicKey.toString(), 'hex'))
        .add(new Buffer(toHex(key).toString(), 'hex'))
        .add('OP_TRUE') // choose the time-delayed refund code path
)
```

C. Tests

For Ethereum, there are javascript testing libraries available. We made use of them perform both unit and functional tests:

```
it(' should be able to deploy atomic swap contract', async () => {
      swapContract = await SwapEth.new({from:contractOwner});
      expect(swapContract).to.exist;
      })

it(' should be able to participate swap', async()=> {
      await swapContract.participate(swapIDpart, swapCloser, secretLock,
timeLock, {from:swapInitiator, value: swapValue});
      const check = await swapContract.check(swapIDpart);
      expect(check[0].c[0]).to.almost((Math.round(Date.now()/1000))+timeLock);
```

As Bitcoin has no virtual machine, unit and functional testing has to be done manually. A performance analysis is still to be performed.

IV. Discussions and Future work

Our implementation of an ACCS between Bitcoin and Ehereum, [10], assumes a pseudo finality in the block production. We indeed consider that Δ is very small compared to the time a transaction takes to be confirmed, resolving therefore all asynchrony related problems. However, our system is still vulnerable to DOS attacks or client crash or Byzantine failure. Moreover, even if we could ensure the atomicity of a swap, there would still be incentives for a Byzantine node to not respect the protocol. For example, we know that crypto-currencies are subject to very high volatility. Then an attacker could be interested on having a victim coin locked until the exchange rate changes.

Moreover we have not yet performed performance analysis of our implementation of an ACCS.

Our future work will be focused on studying the reliability and security of atomic swaps. To do so we will simulate both blockchains and see how our implementation resist to network overload or DDOS attacks. We would like to estimate the correct parameter Δ for each blockchain to assume a probabilistic finality and therefore reliability.

References

[1] Top 100 richest bitcoin addresses https://bitinfocharts.com/top-100-richest-bitcoin-addresses.html Last accessed July 5, 2018

[2] Maurice Herlihy, Atomic Cross-Chain Swaps. v4 May 2018 https://arxiv.org/pdf/1801.09515.pdf

[3] Bitcoin: A Peer-to-Peer Electronic Cash System, Satoshi Nakamoto August, 2008 https://bitcoin.org/bitcoin.pdf

[4] Bitcoinwiki Hashed timelock contracts https://en.bitcoin.it/wiki/HashedTimelockContracts As of July 1, 2018

[5] TierNolan Bitcoin talk forum
https://bitcointalk.org/index.php?topic=193281.msg
2224949#msg2224949 As of May 21, 2013
[6] Ethereum Virtual Machine
https://en.wikipedia.org/wiki/Ethereum#Virtual _Machine
Last accessed July 5, 2018
[7] Unspent Transaction Output UTXO
https://learnmeabitcoin.com/glossary/utxo
Last accessed July 5, 2018
[8] Bitcoin SCRIPT
https://en.bitcoin.it/wiki/Script Last accessed July 5, 2018
[9] Transactions readthedoc
https://web3j.readthedocs.io/en/latest/ transactions.html Last
accessed July 5, 2018
[10] Implementation of HTLC for bitcoin and ethereum
https://github.com/leoloco/BTC-ETH_HTLC Last accessed
July 5, 2018
[11] Bitcoin HTLC
https://github.com/leon-do/Bitcoin-HTLC-Wallet Last accessed
July 5, 2018