Cross-chain Atomic Swaps between Ethereum and Monero

Elizabeth Binks
self@elizabeth.website

June 2022

,

Abstract

Atomic swaps are a method for trustlessly swapping assets between two blockchains. They are easily implemented between blockchains that have capability for hash timelocked contracts; however, when one of the chains does not have any scripting capabilities, such as Monero, it introduces new challenges. A BTC-XMR atomic swap protocol has already been described and implemented. This article describes an ETH-XMR atomic swap protocol that utilizes the smart contract capabilities of Ethereum. Taking into account Ethereum's gas cost, it also uses off-chain discrete logarithm equality proofs as a method for reduction of transaction fees.

1 Background

Atomic swaps are a method for trustlessly swapping assets between two blockchains. It is entirely peer-to-peer and requires no third party. Assuming the two participants are able to meet and agree on the amounts to swap, the protocol is able to proceed to completion. If the protocol is not able to complete, for example, if one party goes offline part-way through, both parties will be refunded.

This article describes a protocol for performing an atomic swap between Ethereum and Monero. However, it can be generalized to work between any chain with smart contract capabilities equivalent to Ethereum and any other chain without any such capabilities.

2 Situation

Say we have two parties, Alice and Bob, where Alice owns ETH and wants XMR, and Bob owns XMR and wants ETH. Alice and Bob meet and agree on the amounts to swap. The method they utilize to meet is out of the scope of this article, although a method is suggested in section 7.2.2.

3 Preliminaries

3.1 Ethereum

As part of its state transition, Ethereum executes its transactions through the Ethereum Virtual Machine (EVM), a nearly-Turing complete machine (the reason it is not is due to "gas", which puts an upper bound on the amount of operations a transaction can do). Thus, any execution that will not exceed the block gas limit can be performed. Gas is a measure of how expensive a transaction will be in terms of computing cost. Every instruction in the EVM has an associated gas cost based off of how expensive the computation is. The financial cost of a transaction is the gas required multiplied by the "gas price", which is the cost per unit of gas in ETH.

Ethereum uses the secp256k1 curve and ECDSA as its signing algorithm. Additionally, ECDSA has the property of key recovery; one can recover the signing public key given a signature and its message.

Unfortunately, Ethereum has no layer-1 privacy. Every transaction is publicly visible, including sender and recipient addresses, value transfers, smart contract calls, and smart contract bytecode. This limitation will be further discussed in section 7.

3.2 Monero

Monero is a privacy-preserving blockchain that obfuscates the link between sender and recipient, as well as the value transferred. It has no smart contract or scripting capabilities.

Unlike Ethereum, Monero uses the ed25519 curve. Monero also has two distinct types of private keys, spend key and view keys. Spend keys are used to spend from an address, while with a view key you can only view the balance of an address. The view key can be derived from the spend key, so by having a spend key to an address you also have the view key. However, you cannot derive the spend key from a view key.

3.3 Discrete logarithm equality proof

As part of the protocol, we wish to verify that a public key on the ed25519 curve has the same discrete logarithm as a public key on the secp256k1 curve. A zero-knowledge proof known as a discrete logarithm equality proof is used to prove equivalence of the secret key corresponding to public keys on either curve.

The details of the implementation are out of the scope of this paper, but are outlined in [3].

The high-level API used in the protocol is as follows:

- `dleq_prove(x)` $\rightarrow$ (`$P_a$`, `$P_b$`, `proof`) where x is a scalar and the discrete logarithm of P_a and P_b on the secp256k1 and ed25519 curves respectively.
- `dleq_verify(`P_a`, `P_b`, proof)` $\rightarrow$ `(true | false)`

4 Protocol

The follow sections describe an ETH-XMR atomic swap protocol between two parties, Alice and Bob, where Alice has ETH and wishes to swap for XMR, and Bob has XMR and wishes to swap for ETH.

4.1 On-chain functionality

In the first step of the protocol, Alice and Bob exchange newly-generated public keys on the ed25519 and secp256k1 curves. For this section, we will refer to only the secp256k1 public keys, as they will be stored on-chain. We will refer to Alice's secp256k1 public key as P_a, with a corresponding private key x_a. Similarly, Bob has a public key P_b and a private key x_b.

Additionally, the swap requires two timelocks, t_0 and t_1 where $t_0 < t_1$.

alice_address and bob_address are the participants's original Ethereum addresses, which should not correspond to the swap keys P_a and P_b.

This protocol requires a smart contract to be deployed to the Ethereum blockchain that contains the following functionality:

- `initiate(alice_address, bob_address,` P_a`,` P_b`,` t_0`,` t_1`,` *value*`)`: Alice is able to initiate the swap on-chain by storing P_a and P_b. She also specifies two timelocks t_0 and t_1. In the same step, she also pays the contract the agreed upon ETH *value*.

- `set_ready()`: After this is called, the contract allows Bob to call the claim(x_b) function, while simultaneously disallowing Alice from calling refund(x_a). Only Alice is able to call this function from `alice_address`. This function does not need to be called, as if t_0 passes, the contract acts the same as if set_ready() was called.

- `claim(`x_b`)`: This function is only callable by Bob from `bob_address`. Bob must provide the private key to P_b stored in the contract. The contract performs a scalar basepoint multiplication on secp256k1 to verify that x_b corresponds to P_b. If the verification succeeds, *value* ETH is transferred to Bob. Otherwise, the transaction reverts. This function is only callable after t_0 or `set_ready()` is called, and before t_1 passes.

- `refund(`x_a`)`: This function is only callable by Alice from `alice_address`. Alice must provide the private key to P_a stored in the contract. The contract performs a scalar basepoint multiplication on secp256k1 to verify that x_a corresponds to P_a. If the verification succeeds, *value* ETH is transferred to Alice. Otherwise, the transaction reverts. This function is only callable before t_0 or the contract is set to ready, or after t_1.

The timelocks t_0 and t_1 are placed to prevent front-running; ie. neither Alice or Bob are able to redeem funds at the same time. This prevents the case where one party submits their secret, and the secret is in the mempool, and the counterparty front-runs them claiming both ETH and XMR. The timelocks also prevent against parties going offline; for example, if Bob never claims the ETH, Alice is able to refund after t_1.

4.2 Protocol steps

In this section, addition between two private keys is defined as scalar addition over the ed25519 field. Addition between two public keys is defined at point addition over the ed25519 curve.

4.2.1 Success path

1. Alice and Bob meet and generate keypairs, where x_a and x_b are their respective secret keys, and P_a and P_b are their respective public keys. They also generate v_a and v_b from x_a and x_b which are private Monero view keys.

2. Alice sends Bob P_a. Bob sends Alice P_b and v_b. Note: Bob needs to send his private view key so that Alice is able to later confirm the locked XMR amount.

3. Alice calls `initiate` on-chain, locking her ETH in the smart contract and storing P_a and P_b.

4. Bob sees the swap initiation and confirms the amount locked and the public keys stored. He then locks the XMR in the account specified by public key $P_a + P_b$, which has the corresponding private spend key $x_a + x_b$ and private view key $v_a + v_b$. He notifies Alice that the XMR is locked.

5. Alice checks the XMR amount locked with view key $v_a + v_b$. She then sets the contract to ready by calling `set_ready()`, allowing Bob to call `claim(`x_b`)`. Note: this step is not strictly necessary, and if Alice does not call `set_ready()`, then after t_0 Bob is still able to claim. This simply speeds up the process.

6. Bob sees that the contract is ready and calls `claim(`x_b`)`, passing in his secret x_b. The contract verifies that x_b is the private key for P_b and transfers the funds to him.

7. Alice sees Bob's secret x_b and combines it with x_a to claim the XMR in the account $P_a + P_b$.

4.2.2 Refund path

1. Same as above.

2. Same as above.

3. Same as above.

4. The refund case occurs when either:

   (a) Bob does not lock the XMR, or locks an incorrect amount of XMR; or

   (b) Bob never claims the ETH.

5. Alice is able to call `refund(`x_a`)` passing in x_a. The contract verifies that x_a is the private key for P_a and transfers the funds to her.

6. Bob sees Alice's secret x_a and combines it with x_b to refund the XMR in the account $P_a + P_b$.

4.3 On-chain public key verification

The protocol can be implemented in two ways; one method would verify a secp256k1 public key on-chain, the second verifies an ed25519 oublic key on-chain. Since Ethereum uses secp256k1 natively, it's much more gas efficient to verify a secp256k1 key on-chain. If an ed25519 on-chain verification was done, the public keys exchanged in protocol step 1 would simply be the ed25519 public key corresponding to each party's secret. However, to verify a secp256k1 key on-chain, a DLEq proof is required to prove that the party's secret corresponds to a valid public key on both secp256k1 and ed25519 curves.

The protocol is modified as follows:

1. In step 1, each party generates their secret x.

   (a) They also generate a DLEq proof using x, which returns $P_{secp256k1}$, $P_{ed25519}$ and $proof$.

   (b) They send both public keys and $proof$ to the counterparty, which verifies them with `dleq_verify(`$P_{secp256k1}$`,` $P_{ed25519}$`,` $proof$`)`. If the proof is not valid, the protocol aborts.

Then, the public keys passed into the smart contract when Alice locks are the secp256k1 keys of each party.

5 Limitations

5.1 Privacy concerns

The current protocol has privacy concerns. On the Monero side, as all transactions are private by default, the receiver of the Monero has their privacy protected. However, on the Ethereum side, all transaction details are visible. Since the swap contract is deployed on-chain, even if the source code is not uploaded on a block explorer like Etherscan, it is still evident which accounts are interacting with the swap. Timing attacks can then be used to link an Ethereum account to a Monero account. Additionally, centralized parties can choose to ban accounts that interact with the atomic swap contract.

Section 7.1 and 7.2 aim to provide a high-level overview of the next iteration of the protocol which addresses these issues.

5.2 Griefing attacks

The protocol currently requires the ETH holder to move first, as if the XMR holder moves first, the ETH holder can simply go offline and the XMR is locked forever. This requires implementations to make the XMR holder the swap "maker"; ie. an XMR holder to wishes to swap for ETH must advertise a swap offer. Then, an ETH holder can act as the "taker", taking one of the existing offers by locking their ETH. Assuming the XMR holder actually holds the XMR they have advertised and is willing to follow the protocol, this scenario works. However, if the offer maker does not actually hold the XMR they claim to, or they go offline right after the offer is taken, then the ETH holder has no option but to refund their ETH. In this case, the ETH holder has lost some ETH due to the gas fees of initiating the swap as well as refunding, while the XMR holder has lost nothing.

6 Protocol using adaptor signatures

This section describes an ETH-XMR atomic swap protocol that uses adaptor signatures to greatly reduce gas fees. Unfortunately, this protocol does not work due to being unable to access account state within an Ethereum smart contract.

6.1 Preliminaries

6.1.1 Adaptor signatures

Adaptor signatures or one-time verifiably encrypted signatures are a method of encrypting a usual ECDSA or Schnorr signature with some encryption key. Given an encrypted signature and decryption key, the unencrypted signature is revealed. Alternatively, given an encrypted signature and corresponding decrypted signature, the secret decryption key can be revealed. This is how they are used within atomic swap.

The encryption key Y is derived from the secret key y with $Y = G * y$ where G is the elliptic curve base point.

The methods for ECDSA adaptor signatures is as follows:

Signing: `ecdsa_adaptor_sign`

- Input:
  - x: signing private key
  - Y: encryption public key
  - $message$: message to sign
- Output:
  - DLEq proof π
  - adaptor signature σ_a

Verification: `ecdsa_adaptor_verify`

- Input:
  - adaptor signature σ_a and DLEq proof π
  - P: signing public key
  - Y: encryption public key
  - *message*: message
- Output:
  - true or false

Signature decryption: `ecdsa_adaptor_decrypt`

- Input:
  - adaptor signature σ_a
  - y: decryption secret key
- Output:
  - ECDSA signature σ

Secret encryption key recovery: `ecdsa_adaptor_recover`

- Input:
  - Y: encryption public key
  - σ_a: adaptor signature
  - σ: ECDSA signature
- Output:
  - secret decryption key y or error

6.2 Protocol description

6.2.1 On-chain functionality

The following is a minimal smart contract for an adaptor-signature based atomic swap.

```
contract SwapV2 {
    address payable owner;
    address payable claimer;

    uint256 timeout_0;
    uint256 timeout_1;

    constructor(address _claimer, uint256 _timeoutDuration) public payable {
        owner = msg.sender;
        claimer = _claimer;
        timeout_0 = block.timestamp + _timeoutDuration;
        timeout_1 = block.timestamp + (_timeoutDuration * 2);
    }

    function claim() public {
        require(msg.sender == claimer);
        require(block.timestamp < timeout_1 && block.timestamp > timeout_0);
        claimer.transfer(this.value);
        selfdestruct();
    }

    function refund() public {
        require(msg.sender == owner);
        require(block.timestamp > timeout_1 || block.timestamp < timeout_0);
        owner.transfer(this.value);
        selfdestruct();
    }
}
```

6.2.2 Success path

1. Alice and Bob meet and generate ed25519 keypairs, where x_a and x_b are their respective secret keys, and P_a and P_b are their respective public keys. item Alice prepares to lock her ETH in a smart contract by determining the `CREATE2` address where the contract will be deployed to. If the contract is a "factory" contract (ie. a contract that can instantiate swaps within it) where the address is already known, this step can be omitted.

2. Alice sends Bob the swap contract address. Bob prepares a transaction tx_{claim} that calls `claim()`. He creates an adaptor signature $\sigma_{claim\_adaptor}$ on tx_{claim} with the public encryption key Y (and secret decryption key y. He sends this adaptor, his Ethereum address, and tx_{claim} to Alice. He also privately decrypts σ_{claim}, a valid signature on tx_{claim}, from σ_{claim} using `ecdsa_adaptor_decrypt`.

3. Alice checks that tx_{claim} is well-formed and locks her ETH in the swap contract, setting Bob's Ethereum address as the claimer.

4. Bob sees that the ETH was locked and confirms that the claimer address is correct. He then locks his XMR in the Monero address with public key $P_a + Y$, thus the private key is $x_a + y$.

5. After $timeout_0$ passes, Bob submits the signed tx_{claim} on-chain. This reveals σ_{claim}, which Alice combines with $\sigma_{claim\_adaptor}$ to reveal the secret encryption key y.

6. Alice claims the XMR in the acount $P_a + Y$ as she now knows the private key, which is $x_a + y$.

6.2.3 Refund path

1. Same as above.

2. Same as above.

3. Same as above. Additionally, Alice prepares tx_{refund} that calls `refund()`. She sends Bob an adaptor signature $\sigma_{refund\_adaptor}$ on tx_{refund} where the public encryption key is P_a. She also privately signs tx_{refund} using *ecdsa_adaptor_decrypt*, creating σ_{refund}. If Alice decides to call `refund()`, x_a is revealed to Bob.

4. If Bob locks his XMR, the step is the same as step 5 above.

5. If Bob locked his XMR, but was unable to claim the ETH before $timeout_1$, Alice will call `refund()`. Then, he can combine σ_{refund} with tx_{refund} to reveal x_a and regain access to the XMR locked in $P_a + Y_b$.

6.3 Limitation

Currently, this protocol is not possible. This is because there is no way to check for an account nonce within an Ethereum smart contract. Since account nonce is part of the transactions tx_{claim} and tx_{refund}, it's possible for a party to create a transaction and adaptor with some nonce nonce, but then submit some transaction in-between sending the adaptor and calling the swap contract. Then, the contract will still accept the transaction, but the signature will be on a transaction with nonce + 1. Since this transaction is not the same as that signed by the adaptor, the counterparty will be unable to extract the secret encryption key.

7 Further work

7.1 Preserving privacy by shielding ETH

As mentioned in section 5, the Ethereum side of the swap has effectively no privacy, as Ethereum has no layer-1 privacy. A potential method to circumvent this would be to add a ZK-shielded pool to the swap contract.

The method could work as follows:

- If an ETH-holder wishes to initiate a swap, they deposit their ETH into the swap's shielded pool and receive a receipt that acts as a proof of deposit. They can also use this receipt to simply withdraw the funds without doing a swap.
- When `initiate()` is called, instead of sending funds with the call, the receipt is passed and validated by the smart contract.
- When the funds are claimed from the contract, the receipt's funds are marked as withdrawn.

This method adds a layer of plausible deniability to users who deposit in the swap; it cannot be proven as it could previously that an atomic swap occured. It also has the effect of mixing swap users's funds.

This could be implemented as a custom solution to the swap or potentially as an integration with existing solutions. The benefit of integration with an existing solution would be a much greater anonymity set, but the drawback is that the swap contract must be interoperable with external smart contracts.

7.2 Network-level privacy

Currently, the protocol requires at least 2 messages to be exchanged. These are the key exchange messages exchanged by the swap participants to initiate the swap. Assuming the participants are able to watch events occuring on the blockchain, no other messages necessarily need to be sent.

7.2.1 Sender-recipient privacy

For a swap to initiate, the two parties must establish a connection between each other to exchange their swap keys. However, this reveals the IPs of each party to the counterparty. A party who wishes to monitor the network and determine who is requesting initiation of swaps, and with what amount, can easily do so without monetary recourse. They can simply spin up nodes that join the network and advertise swap offers and record who attempts to initiate with them, thereby determining IP addresses that want Monero. Alternatively, they can search for providers offering XMR, and thus find the IPs of users who reportedly own Monero.

A simple solution is for users of the swap to use a VPN service. This will hide their real IP from other network participants. The drawback is that the user must trust the VPN provider not to publish their network activity.

Two potential solutions for this are onion routing or a mixnet. Onion routing allows network messages to be wrapped in successive layers of encryption, which must be unwrapped by each relaying node one at a time until the destination is reached. This hides the sender of the message from the relay nodes as well as the receipient, and hides the recipient's address from the sender. There exist implementations of a Tor transport for the lip2p library used by the current swap implementation, which may be integrated for increased privacy.

Alternatively, a mixnet can also be utilized. A mixnet provides greater privacy and resistance to metadata attacks than onion routing. Messages sent via onion routing may be correlated by timing and packet size. A mixnet prevents against this by "mixing" different packets together before sending them out from a relay node, as to obfuscate the size of a packet. As well, the relay node holds on to them for random intervals of time. This prevents against both timing attacks at the cost of greater latency. Using a mixnet for the networking implementation of the swap would be ideal, as latency is not critical; the swap users can simply set a sufficiently large timeout to negate the effects of latency. However, more research would need to be conducted into existing mixnet implementations that can be used, which is out of the scope of this document.

7.2.2 DHT privacy

The current atomic swap implementation uses a distributed hash table (DHT) for discovery of peers on the network, particular those who are providing swap offers. While not necessaily within the scope of an atomic swap protocol, which does not dictate how the participants meet, any implementation that uses a DHT will need to consider the privacy implications. However, a DHT is preferred for peer discovery over centralized alternatives such as a website, which may be censored or taken down by the hosting party. Additionally, by discovering peers through a centralized service, the website host becomes aware of every IP that visits the website, which would not be the case with a swap implementation that operates entirely through a peer-to-peer network.

DHTs are used to store and retrieve content in a decentralized, peer-to-peer network. Content is replicated between nodes, which can then be retrieved by a user by querying nodes in the network to "get closer" and eventually retrieve the contract they are searching for. DHTs are used in a blockchain context for peer discovery. A node in the network will advertise themselves in the DHT which can then be found by another party searching through the DHT. Similarly, they are used in the swap implementation to discover peers.

Most DHT implementations are not privacy-preserving and leak metadata about who is hosting what content, who is requesting it, and when. Similarly to the previous section, someone who wishes to extract metadata from the network can create a large number of nodes, wait until the DHT content gets replicated to them, then determine who is requesting that data.

There has been research done into DHT implementations that preserve privacy of lookups, for example Octopus [4]. A swap implementation which aims to be as privacy-preserving as possible should use a privacy-preserving DHT implementation for peer and offer discovery.

7.3 Direct swaps for ERC20 tokens

An extension to the protocol could be direct swaps for ERC20 tokens. This could be implemented by modifying the swap contract to internally call `transfer()` on an ERC20 when `initiate` is called.

However, this method requires liquidity of ERC20 tokens, which may not be feasible especially for more niche tokens. Another method could be integrating with a DEX such as Uniswap so that once when swap completes, the ETH is automatically exchanged on the DEX for the desired token.

8 Conclusion

In summary, a protocol for trustless, peer-to-peer ETH-XMR atomic swaps has been described in this article. As well, limitations of the protocol in regards to privacy, griefing attacks, and gas costs has been discussed. Future research and implementation work has been outlined regarding privacy improvements and extensions.

9 Acknowledgements

Thank you to the Monero community for seeing the value in the project and funding this research. Additional acknowledgements to h4sh3d for first specifying Bitcoin-Monero atomic swaps which were the inspiration for this project. Thank you to Marcin Gorny and Aleixo Sanchez for helping bring this project off the ground and your contributions to the protocol. Also, thank you to Robert Hambrock for proposing and implementing the cross-curve DLEq trick into the protocol, which saves significantly on gas. Finally, thank you to Luke Parker, Justin Berman, and Crypt0-Bear for their help and support throughout this process.

References

[1] Joel Gugger. Bitcoin-Monero Cross-chain Atomic Swap, 2020. https://eprint.iacr.org/2020/1126.pdf
[2] Philipp Hoenisch and Lucas Soriano del Pino. Atomic Swaps between Bitcoin and Monero, 2021. https://arxiv.org/pdf/2101.12332.pdf
[3] Sarang Noether. Discrete logarithm equality across groups, 2018. https://www.getmonero.org/resources/research-lab/pubs/MRL-0010.pdf
[4] Qiyan Wang and Nikita Borisov. Octopus: A Secure and Anonymous DHT Lookup, 2012. https://arxiv.org/pdf/1203.2668

[5] discretelogcontracts. ECDSA adaptor signature specification, 2021. https://github.com/discreetlogcontracts/dlcspecs/blob
adaptor.md
[6] Lloyd Fournier. One-time Verifiably Encrypted Signatures, 2020. https://github.com/LLFourn/one-time-VES/blob/master/main.pdf