Transferable Cross-Chain Options

Daniel Engel
Computer Science Dept., Brown University
Providence, RI, USA

Yingjie Xue
Computer Science Dept., Brown University
Providence, RI, USA

ABSTRACT
An option is a financial agreement between two parties to trade two assets. One party is given the right, but not the obligation, to complete the swap before a specified termination time. In today’s financial markets, an option is considered an asset which can itself be transferred: while an option is active, one party can sell its rights (or obligations) to another.

Today’s blockchains support simple options in the form of cross-chain atomic swap protocols where one party has the choice whether to complete the swap. The options implemented by these cross-chain protocols, are, however, transferable.

This paper proposes novel distributed protocols for transferable cross-chain options, where both option owners and providers can sell their positions to third parties. The protocol ensures that none of the parties can be cheated, that no unauthorized party can interfere, and that the transfer succeeds if the buyer and seller faithfully follow the protocol.

ACM Reference Format:
Daniel Engel and Yingjie Xue. 2022. Transferable Cross-Chain Options. In 4th ACM Conference on Advances in Financial Technologies (AFT ’22), September 19–21, 2022, Cambridge, MA, USA. ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3558535.3559774

1 INTRODUCTION
An option is a financial agreement between two parties, say, Alice and Bob. Alice owns some units of “florin” cryptocurrency, while Bob owns some units of “guilder” cryptocurrency. Alice and Bob agree that Alice will purchase from Bob the right, but not the obligation, to buy 100 guilders from him in return for 100 florins from her, at any time before, say, next Tuesday. If the value of guilders relative to florins goes up before Tuesday, Alice will exercise her option by executing the trade, and otherwise she will keep her florins and walk away. Alice pays Bob a fee, called a premium, to compensate him for his inconvenience and risk.

In the world of decentralized finance (DeFi), florins and guilders are managed on distinct blockchains, Alice and Bob are autonomous agents, and they do not trust each other. Moreover, they have no recourse to third-party arbiters or to courts of law. Several non-trivial distributed protocols have been proposed to execute cross-chain swaps and options [7, 11–14, 16, 21–23]. Simplifying somewhat, these protocols guarantee that if Alice and Bob are both honest, then the deal unfolds as planned, but if either party cheats, the honest party ends up in what it considers a satisfactory position.

The contribution of this paper is to take the notion of a cross-chain option to the next level: while Alice’s unexercised option has value, she should be able to sell it to a third party, Carol. In conventional finance, such a transfer is simple: Bob grants Alice an options contract, Alice signs over that contract to Carol, and all agreements are enforced by civil law. In the lawless world of decentralized finance, by contrast, transferable options require a carefully-crafted distributed protocol.

We propose novel cross-chain protocols, to support transferable cross-chain options. Additionally, we prove the security properties of our proposed protocols, and detailed pseudocode for the protocols is provided. Informally, our proposed protocols can address option transfer in the following scenarios:

1. Alice (the option owner, i.e the one who has a right to buy an asset) transfers her position to another party (Carol).
2. Bob (the option provider, i.e. the one who provides the owner the right) transfers his position to another party (David).
3. Alice and Bob concurrently transfer their positions to Carol and David, respectively.

Take the option position transfer between Alice and Carol for example. The proposed transfer protocol ensures that if Alice and Carol are honest, then Alice relinquishes her rights, Carol assumes Alice’s rights, and Bob cannot veto the transfer. If Carol cheats, then Alice’s rights remain with Alice.

We view the proposed distributed protocols as a first step toward a more ambitious goal. One can imagine more complex cross-chain deals where parties acquire various rights and obligations (options, futures, derivatives, and so on). It should be possible for a party who holds unrealized rights or obligations to sell those rights or obligations to another party, atomically transferring its position without being hindered by any third party. Here, we show how to make certain transferable cross-chain option deals, but someday we hope to do the same for arbitrary cross-chain deals.

This paper is organized as follows. Our model is given in Section 2. We provide an overview of our proposed transferable two-party swap protocols in Section 3 and detailed protocols are described in Section 4. We prove security properties of our proposed protocols in Section 5. In Section 6, we describe related work. Finally, we conclude in Section 7.

2 MODEL
For our purposes, a blockchain is a tamper-proof distributed ledger (or database) that tracks ownership of assets by parties. An asset can be a cryptocurrency, a token, an electronic deed to property, and so on. A party can be a person, an organization, or even a contract (see below). There are multiple blockchains managing different kinds of assets. We focus here on applications where mutually-untrusting
parties trade assets across multiple blockchains. Because we treat blockchains simply as ledgers, our results do not depend on specific blockchain technology (such as proof-of-stake vs proof-of-work). We assume only that ledgers are highly available, tamper-proof, and capable of running smart contracts.

A smart contract (or “contract”) is a blockchain-resident program initialized and called by the parties. A party can create a contract on a blockchain, or call a function exported by an existing contract. Contract code and state are public, so a party calling a contract function knows what code will be executed. Contract code is deterministic because contract execution is replicated, and all executions must agree.

A contract can read or write ledger entries on the blockchain where it resides, but it cannot actively access data from the outside world, including calling contracts on other blockchains. Although there are protocols that allow blockchains to communicate (cross-chain proofs), they have weaknesses that make them difficult to use in practice: for example, incompatibility problems, lack of decentralization, and non-deterministic guarantees of message delivery [20]. Therefore, we assume different blockchains do not communicate.

A contract on blockchain A can learn of a change to a blockchain B only if some party explicitly informs A of B’s change, along with some kind of “proof” that the information about B’s state is correct. Contract code is passive, public, deterministic, and trusted, while parties are active, autonomous, and potentially dishonest.

At any given time each party has a given financial position. This is characterized by their current and future financial obligations/allowances. Financial positions are managed by rules on contracts that can reside across multiple distinct blockchains.

Our execution model is synchronous: there is a known upper bound $\Delta$ on the propagation time for one party’s change to the blockchain state, plus the time to be noticed by the other parties. Specifically, blockchains generate new blocks at a steady rate, and valid transactions sent to the blockchain will be included in a block and visible to participants within a known, bounded time $\Delta$.

As noted, we assume blockchains are always available, tamper-proof, and that they correctly execute their contracts. Although parties may display Byzantine behavior, contracts can limit their behavior by rejecting unexpected contract calls.

We make standard cryptographic assumptions. Each party has a public key and a private key, and any party’s public key is known to all. Messages are signed so they cannot be forged, and they include single-use labels (“nonces”) so they cannot be replayed.

### 3 PROBLEM OVERVIEW

One way to convey the challenge presented by making options transferable is to trace the evolutionary path that leads from 20th Century distributed systems to modern decentralized finance (DeFi). Suppose, in 1999, Alice and Bob agree that she will transfer 100 dollars to him at a New York bank, and he will transfer 100 euros to her at a Paris bank. Each bank maintains its own database, which communicate over a network. The swap protocol must tolerate hardware failures: databases can crash and messages can be lost. Each bank makes a tentative transfer. In the classical two-phase commit protocol [2], each bank records its tentative transfers “somewhere safe” (on a magnetic disk that survives crashes) and sends to a trusted coordinator a vote whether to commit or abort. If both banks vote to commit, the transfers are installed, and if either votes to abort, or does not vote in a reasonable time, the transfers are discarded, leaving both databases unchanged.

The two-phase commit protocol established a pattern for later blockchain protocols. Today, Alice and Bob want to trade units of cryptocurrency. They agree to exchange 100 of her (electronic) florins for 100 of his (electronic) guilders. Each cryptocurrency is managed on a distinct blockchain. Alice and Bob must agree on a swap protocol that tolerates not only hardware failures, but Byzantine failures by participants: each party must protect itself if its counterpart cheats by departing from the agreed-upon protocol.

Atomic swap protocols based on hashed timelock contracts (HTLCs) [13, 17] mimic two-phase commit. In the HTLC protocol’s first phase, each party places the assets to be transferred “somewhere safe”. In place of writing to magnetic disk, Alice transfers her florins to an escrow account controlled by a smart contract, a program that decides, based on later input, whether to commit by transferring Alice’s florins to Bob, or to abort by refunding Alice her florins. Then Bob does the same. Each escrow transfer is effectively a vote to commit. When both votes are confirmed, Alice releases a secret that causes the contracts to complete the swap. Otherwise, if the escrow transfers are not confirmed in reasonable time, the contracts refund the assets to their original owners. This description elides many technical details, but the HTLC protocol’s overall structure is remarkably similar to two-phase commit, despite substantial differences in their failure models.

While technically correct, HTLC swap protocols are flawed: once both assets are escrowed, Alice can take her time deciding whether to trigger the swap. Cryptocurrencies are notoriously volatile, so if the value of Bob’s guilders goes up relative to Alice’s florins before the timeout expires, she can choose to complete the swap at the last minute. If the value goes down, she is free to walk away without penalty. Bob may be reluctant to accept such a deal. This protocol is unfair to Bob because only Alice has optionality: at the end, he cannot back out of the deal, but she can. If she does back out, he gets his guilders back, but only after a possibly long delay while the market is moving against him. Bob thus incurs the opportunity cost of not being able to use his coins while they are escrowed. By contrast, in conventional finance, this deal would be structured as an option contract, where Alice pays Bob a fee, called a premium, to compensate him if she walks away without completing the deal.

Cross-chain atomic swap options require carefully-defined distributed protocols [16, 23]. The protocol proposed by Xue and Herlihy [23] is structured like an iterated two-phase commit protocol: in the first phase the parties escrow premiums. If all goes well, in the next phase they escrow coins. If all goes well in the final phase they complete the swap. Any party who drops out of the protocol ends up paying a premium to the other, and in the end, Alice has the optionality, but Bob has been compensated for his risk.

We are now ready to address the main topic of this paper. As long as Alice has optionality, that optionality has value. Alice should be able to sell her position to a third party Carol. (Bob may also want to sell his position to a third party David). For starters, we focus on Alice’s position transfer since Bob’s transfer will be similar. There are many reasons Alice and Carol might agree to such a...
deal. Alice may want to liquidate her position because she needs cash. Perhaps Alice and Carol have different opinions on the future values of florins versus guilders, or they have different levels of risk tolerance.

A transferable atomic swap option (or “transferable swap”) protocol roughly must satisfy the following properties. (We define these properties more precisely in Section 5.)

- **Liveness:** If Alice and Carol both follow the protocol, then (1) Carol acquires the right to buy Bob’s coins at the same price and deadline, (2) Alice loses that right, and she is paid by Carol, (3) Bob cannot veto the transfer.
- **Safety:** As long as one of Alice or Carol follows the protocol, (1) the protocol completes before the option expires, and (2) when the protocol completes, exactly one of Alice or Carol has the right to trigger the atomic swap with Bob, (3) Bob’s position in the option does not change if he conforms.

Here is a high-level summary of our protocol. The contract linking Alice and Bob is a delayed swap: Alice escrows her asset, then Bob escrows his. These escrow contracts are controlled by a hashlock h. Until each contract’s timeout expires, that contract will complete its side of the swap when Alice produces a secret A1 such that h = H(A1), where H(·) is a predefined cryptographic hash function. Alice has optionality because she alone knows the secret.

Note that the option we discuss here is different from that found in traditional finance, where the law enforces all required actions. The option we describe here requires Alice to escrow her asset before acquiring optionality. If she doesn’t commit her assets to be locked during the protocol, she could always launch a double spend attack to spend those assets elsewhere but meanwhile claim Bob’s assets by exercising the option.

Here is how Carol can buy Alice’s position. First, there must be enough time to complete this transfer before Alice’s option expires. Roughly speaking, Carol generates a secret, then Alice and Carol swap the roles of Carol’s and Alice’s secrets: following the transfer, Alice’s secret will no longer trigger the swap with Bob, but Carol’s secret will.

There are two contracts (discussed in detail below). Contract AB controls Alice’s possible payment to Bob, while BA controls Bob’s payment to Alice. The optionality transfer itself is structured like a cross-chain swap, exchanging roles instead of assets. In the first phase, Alice marks contracts AB and BA as mutating, temporarily preventing any transfer to or from Bob. This step prevents Alice or Carol from creating a chaotic situation by triggering a partial asset swap with Bob while the optionality transfer is in progress. In the second phase, Carol replaces the (hash of) Alice’s secret and address in both AB and BA with the (hash of) Carol’s secret and address. Just as for regular swaps, the optionality transfer protocol will time out and revert if Alice or Carol fails to take a step in time.

There is still a danger that Alice and Carol might cheat Bob by making inconsistent changes to contracts AB and BA. Because AB and BA cannot coordinate directly, the protocol has a built-in delay to give Bob an opportunity to contest a malformed transfer and to revert its changes. If the transfer is well-formed, Bob can expedite the protocol by actively approving of the transfer, perhaps in return for an extra premium. If Bob remains silent, the protocol will proceed after the delay expires. Bob contests the transfer by providing proof that Alice signed inconsistent changes to AB and BA.

To keep the presentation uncluttered, we omit some functionality that would be expected in a full protocol, but that is not essential for optionality transfer. For example, there would be additional steps where parties deposit premiums to compensate one another if one party leaves the other’s assets temporarily trapped in escrow, where Alice pays Bob a premium to encourage prompt transfer approval, and where Alice posts a bond to be slashed if she is caught sending inconsistent information to AB and BA.

### 4 TWO-PARTY TRANSFERABLE SWAP

#### 4.1 Overview

We will now describe several protocols for transferring option positions. First we will describe a protocol that allows Alice to transfer her position as the option owner in a swap with Bob, to a third-party Carol. Next, we will give a protocol that allows Bob to transfer his position as the option provider in a swap with Alice, to a third-party David. Finally, we show that both of these protocols can be run concurrently and can also support multiple concurrent buyers. Solidity code for these contracts appears in Appendix A.1.

A protocol that follows the protocol is conforming, and a party that does not is adversarial. A party’s principal is the assets escrowed at a contract during the protocol execution. We use transfer and replace interchangeably. First we describe the initial setup where Alice and Bob first lock their principals using a hashlock. Throughout this paper, the party who owns the preimage to the hashlock (Alice, the initial owner of the option) is called a leader and the counterparty (Bob) is called a follower. This terminology is similar to prior two-party, HTLC-based cross-chain swap protocols [10, 17, 24].

**Setup.**

1. **Agreement.** Alice and Bob agree on a time startLeader to start the protocol, usually shortly after this agreement. Alice and Bob agree on the amounts AssetA, AssetB forming their principals, as well as the number of rounds dT1, where after T = startLeader + dT · Δ when Alice’s optionality on the BA contract expires. Alice generates a secret A1 and H(A1) which is used as swap_hashlock to redeem both principals as in a typical two-party swap.

2. **Escrow.** After the agreement achieved,
   a. Within Δ, Alice publishes the AB contract, escrowing AssetA, setting H(A1) as the swap_hashlock and set T + Δ as the timeout for AB contract to expire.
   b. Once Bob sees AssetA is escrowed with the correct timeout, before Δ elapses, Bob creates the BA contract, escrowing AssetB, setting H(A1) as the swap_hashlock and set T as the timeout for BA contract to expire.

A swap option between Alice and Bob, where Alice has optionality, is determined by Alice and Bob’s addresses, i.e. sender and receiver in contracts, the swap_hashlock used for redemption, and the timeout T for refunds. To make the swap option transferable to Carol, we will need to replace Alice’s address with Carol’s as sender in the AB contract and as receiver in the BA

1We inherit the requirement dT ≥ 4 from the standard two-party swap protocol [13]
contract, and replace the \textit{swap\_hashlock} with a new one generated by Carol. In short, we must atomically replace the fields \((AB\text{.}\text{sender}, BA\text{.}receiver, AB\text{.}\text{swap\_hashlock}, BA\text{.}\text{swap\_hashlock})\).

The contracts export the following functions of interest:

- Asset transfer related: the receiver calls \textit{claim()} to withdraw the asset in the contract. This function requires that the preimage of the \textit{swap\_hashlock} is sent to \textit{claim()} before the contract expires. The sender calls \textit{refund()} to withdraw the asset after the contract expires.
- Option transfer related: \textit{mutateLockLeader()} temporarily freezes the assets and stores a \textit{replace\_hashlock} used to replace Alice’s role (the replacement between Carol and Alice is also enforced by a hashlock mechanism) and the new \textit{swap\_hashlock} if Carol successfully replaces Alice. The call to \textit{replace()} completes the replacement for Carol.

For example, consider the \textit{AB} contract. The \textit{mutateLockLeader()} function freezes the assets, tentatively records the new \textit{sender} as \textit{candidate\_sender}, the new \textit{swap\_hashlock} for redemption, and records a short-term \textit{replace\_hashlock} to be used by \textit{replace()} to finalize the replacement. Later, when \textit{replace()} is called with the secret matching the \textit{replace\_hashlock}, these tentative changes become permanent. The \textit{replaceLeader()} function has a timeout. If that timeout expires, the internal \textit{revertLeader()} function unfreezes the assets, discards the tentative changes, and restores the contract’s previous state.

The challenge is how to ensure the transfer is atomic on both contracts. If the transfer is not atomic, Bob will be cheated if his principal can be claimed but he cannot claim Alice’s principal. To protect Bob, the contract allows Bob to inspect the changes and ensure atomicity by relaying changes on one contract to another.

4.2 Transfer Leader Position

Our protocol will consist of 3 phases, each of which we describe in turn:

1. \textit{Mutate Lock Phase}. Alice locks the assets on both contracts and tentatively transfers her position to Carol.
2. \textit{Consistency Phase}. Bob makes sure the tentative changes on both contracts are consistent.
3. \textit{Replace/Revert Phase}. Carol replaces Alice’s role or she gives up and Alice gets her option back.

\textit{I: Mutate Lock Phase}. Assume Alice and Carol agree on Asset\(_C\) for transferring her role to Carol at time \textit{startLeader}, which is the start time the following executions are based on\(^2\). Carol has secrets \(C_1, C_2\) and generates hashlocks \(H(C_1), H(C_2)\). Let \(C_{msg} = [H(C_1), H(C_2)]\) be the message Carol sends to specify that she would like to use \{\textit{replace\_hashlock}, \textit{swap\_hashlock}\}, respectively.

In this phase, Carol prepares tentative payment Asset\(_C\) to Alice to replace Alice’s role. Alice locks the \textit{AB} and \textit{BA} contracts so that the currently escrowed assets cannot be claimed from the original swap.

1. By \textit{startLeader} + \(\Delta\), Carol creates the CA contract, escrowing Asset\(_C\), setting the \textit{swap\_hashlock} as \(H(C_1)\) for Alice to redeem Asset\(_C\), and the timeout to be \textit{startLeader} + 9\(\Delta\). She also sends \(C_{msg} = [H(C_1), H(C_2)]\) to Alice.
2. If the previous step succeeds, before \(\Delta\) elapses, Alice concurrently calls \textit{mutateLockLeader()} on both \textit{AB} and \textit{BA} contracts, tentatively setting \{\textit{replace\_hashlock} = \(H(C_1)\), \textit{swap\_hashlock} = \(H(C_2)\)\} to both contracts, and setting \textit{candidate\_sender} = Carol\textit{.address} on \textit{AB} and \textit{candidate\_receiver} = Carol\textit{.address} on \textit{BA} contract. If Alice is conforming, \textit{mutateLockLeader()} should be called by \textit{startLeader} + 2\(\Delta\) on both \textit{AB} and \textit{BA}.

\textit{II: Consistency Phase}. It is possible that in the worst case, in the Mutate Lock Phase, Alice could report inconsistent tentative changes to \textit{AB}, \textit{BA} by either mutating one contract but not the other, or by reporting different hashlocks to each contract. Because the two contracts cannot communicate, they have no way of knowing what is happening on the other contract. One such example of an attack is shown in Fig. 2.

Because of these attacks, in the \textit{Consistency Phase}, Bob is given a period to ensure both \textit{AB} and \textit{BA} have the same changes. If one of the contracts is not changed, then Bob forwards the change (by forwarding Alice’s signature on the contract she signed) to the other contract. If both contracts are tentatively changed but the changed are different, then Bob can call \textit{contestLeader()} to prove that Alice has lied and reported inconsistent changes signed by Alice.

1. If Bob sees that Alice only calls \textit{mutateLockLeader()} on \textit{AB}(BA), then within \(\Delta\), Bob should call \textit{mutateLockLeader()} on \textit{BA}(AB) contract and set \{\textit{replace\_hashlock} = \(H(C_1)\), \textit{swap\_hashlock} = \(H(C_2)\)\} by forwarding Alice’s signature.
2. If Bob sees Alice has called \textit{mutateLockLeader()} on both \textit{AB} and \textit{BA}, but the changes are different, i.e. either one or more hashlocks are not the same, or the new candidate sender does not correspond to the new candidate receiver, then before a \(\Delta\) elapses, Bob should call \textit{contestLeader()} on both contracts, forwarding Alice’s signature on one contract to the other. In this case, the tentative change will be reverted.
3. If Bob sees Alice has called \textit{claim()} with her secret \(A_1\) on \textit{BA} to redeem Asset\(_B\), and she also successfully called \textit{mutateLockLeader()} on \textit{AB}, then Bob should call \textit{contestLeader()} on \textit{AB} before a \(\Delta\) elapses to revert the tentative change. Bob then calls \textit{claim()} using \(A_1\) on \textit{AB} to redeem Asset\(_A\).

Importantly, Bob is able to contest when either Alice reports inconsistent mutation signatures between contracts or if she preemptively reveals her swap secret. Bob’s ability to call \textit{contestLeader()} ensures that he’s able to maintain his position in the original trade if Alice deviates from the protocol by reporting inconsistent changes.

\textit{III: Replace/Revert Phase}. After the Consistency Phase has ended, in the Replace/Revert Phase, the swap either finalizes to one between Carol and Bob or reverts back to the original swap between Alice and Bob. Once Carol sees consistent changes with no successful \textit{contestLeader()} calls by Bob, she calls \textit{replaceLeader()} to make herself the new leader of the swap. Figure 1 shows an execution of the protocol where all parties are conforming. If Bob sees Carol only

\(^2\)The protocol requires that \textit{startLeader} \(\leq T - 9\Delta\). Otherwise, there may not be sufficient time to complete the replacement.
The BA contract expires at time $T$, the last time for the leader to send her secret to redeem the principal. The AB contract expires at time $T + \Delta$.

Denote the time when $\text{mutateLockLeader()}$ is called on $AB(BA)$ contract as $AB(BA)$. $\text{mutation.start.time}$. After the mutation starts on any contract, Bob is given time $2\Delta$ to contest Alice’s mutation: $AB(BA)$. $\text{contest.timeout} = AB(AB)$. $\text{mutation.start.time} + 2\Delta$.

When Bob is able to contest, and Bob has not called $\text{mutateLockLeader()}$ himself, Carol cannot call $\text{replaceLeader()}$. After the contest period elapses, Carol has $2\Delta$ to call $\text{replaceLeader()}$. Here we have $AB(AB)$. replace.timeout.Carol = $AB(AB)$. $\text{mutation.start.time} + 4\Delta$.

If Carol deviates and only calls $\text{replaceLeader()}$ on one contract, we allow Bob to call $\text{replaceLeader()}$ on the another contract. We give Bob $2\Delta$ more than Carol to call $\text{replaceLeader()}$. That is, $AB(AB)$. replace.timeout._Bob = $AB(AB)$. $\text{mutation.start.time} + 6\Delta$.

We see that it takes $6\Delta$ from the start of a $\text{mutateLockLeader()}$ call to finalize a replacement in the worst case. Thus, $AB(AB)$. $\text{mutation.start.time} + 6\Delta \leq T - \Delta$, which means $AB(AB)$. $\text{mutation.start.time} \leq T - 7\Delta$.

The last person that is able to call $\text{mutateLockLeader()}$ is Bob. He is given one more $\Delta$ than Alice on each contract to call this function in case Alice decides to just call it on one contract. The deadline for Alice to call $\text{mutateLockLeader()}$ should be $T - 7\Delta$. The deadline for Bob to call $\text{mutateLockLeader()}$ should be $T - 6\Delta$. Consider the case when Alice is adversarial by not calling $\text{mutateLockLeader()}$ on $BA$ but does call it on $AB$ by $T - 7\Delta$. If Bob is compliant, then he calls $\text{mutateLockLeader()}$ by $T - 6\Delta$ on the $BA$ contract. Because we require $AB(AB)$. $\text{mutation.start.time} \leq T - 7\Delta$, Bob is guaranteed $6\Delta$ to do a full replacement in the worst case. Conforming Carol’s safety is also guaranteed since she can give up the replacement if she finds there is not sufficient time to call $\text{replaceLeader()}$ then $\text{claim()}$. In that case, due to the timeout of the original swap, adversarial Alice loses her opportunity to sell her position.

For $CA$ edge, after Carol escrows $\text{Asset}_{C}$, in the worst case it takes $7\Delta$ to complete the replacement, ($\Delta$ to start the mutation and $6\Delta$ to complete the replacement). After the replacement, it takes one $\Delta$ for Alice to redeem. Thus, the timeout for $CA$ edge is $\text{start.getEnd} + 9\Delta$.

The reason why we have $2\Delta$ for Bob to contest, and $2\Delta$ more than Carol to call $\text{replaceLeader()}$ is that the start time of mutation can be staggered by $\Delta$ on two contracts, which will be described in detail in proof.

### 4.2.1 An Optimization

Because of the $\text{contestLeader()}$ time window, in the worst case, Carol has to wait $2\Delta$ after a $\text{mutateLockLeader()}$ call to decide whether to release her secret or not. We can speed up the process by adding an $\text{approveLeader()}$ method. Instead of waiting for $2\Delta$ even though Bob does not contest at all, once Bob sees Alice has made consistent mutations on both edges, Bob can call $\text{approveLeader()}$ to approve the pending transfer. In other words, by calling $\text{approveLeader()}$, Bob gives up the right to contest meaning Carol will not have to wait out the contest window in order to call $\text{replaceLeader()}$. In that case, if Bob cooperates, the transfer would be finalized sooner. If Bob doesn’t cooperate, the transfer would be finalized later but not stoppable if Alice and Carol are conforming.
We will now describe a protocol that allows Bob to transfer his flag indicating whether Bob has given up his ability to approve $A_B$. Here we assume Alice has reported consistent signatures in the **Replace/Revert Phase** but has reported them a $\Delta$ apart. Colored blocks represent time periods when functions can be called. The Leader suffix is excluded from the function calls for simplicity.

This can be accomplished by adding to the $AB(BA)$ contracts an approved flag indicating whether Bob has given up his ability to call contestLeader(). Phases II–III are the only ones affected.

We add the following instruction to the Consistency Phase:

1. Once Bob sees Alice has called mutateLockLeader() on both $AB$ and $BA$ with the same change, then before a $\Delta$ elapses, Bob should call approveLeader() on both $AB$ and $BA$.

and to the Replace/Revert Phase:

1. If Carol sees Bob approved the tentative consistent changes on both $AB$ and $BA$, then before a $\Delta$ elapses, she calls replaceLeader() on both $AB$ and $BA$.

If in addition to conforming to the base protocol in Section 5.1, Bob also executes the approve step shown previously, then we call Bob altruistic. This just indicates that Bob is willing to speed up the termination of the protocol even if it doesn’t necessarily change his financial position.

### 4.3 Transfer Follower Position

We will now describe a protocol that allows Bob to transfer his position in a swap with Alice, to a third-party David. The main difference with this protocol and the previous one, is that Bob cannot limit Alice’s optionality. Namely, he cannot unilaterally lock the asset in the $BA$ contract since this would limit the optionality Alice purchased. Effectively Alice must always be able to claim the funds on $BA$ with her secret until the swap itself times out. This weakening of the constraint that we had in the previous protocol, allows for a much simpler protocol. It consists of 2 phases:

1. **Mutate Phase**
2. **Replace/Revert Phase**

#### I: Mutate Phase

1. Bob and David agree on $Asset_D$ at time $startFollower$, similar to the replacing leader protocol. David creates $DB$ contract, escrowing $Asset_D$, setting the swap hashlock as $H(D_1)$ for Bob to redeem $Asset_D$ and the timeout be $startFollower + 5\Delta$.

2. If the previous step succeeds, and Alice has not revealed $A_1$ before $\Delta$ elapses, Bob concurrently calls mutateLockFollower() on $AB$ and mutateLockFreeFollower() on $BA$ contracts, tentatively setting $\text{mutate_hashlock} = H(D_1)$ to both contracts, and setting candidate_sender = David.address on $BA$ and candidate_receiver = David.address on $AB$. We use mutateLockFreeFollower() because on the $AB$ contract, this mutation does not lock the asset on $AB$. This is necessary to ensure Alice does not temporarily lose her optionality.

#### II: Replace/Revert Lock Phase.

1. If David sees Bob successfully call mutateLockLeader(), mutateLockFreeFollower() on both $AB$ and $BA$ respectively, and $\Delta$ then within $\Delta$ he concurrently executes replaceFollower() with $D_1$ on $AB$ and $BA$ to replace Bob.

2. If Bob sees David call replaceFollower() on either $AB$ or $BA$, within $\Delta$ he should call claim() on $DB$ to claim $Asset_D$.

From this point on, we will use Protocol 4.2 to refer the protocol in Section 4.2, Protocol 4.2.1 to the refer to the protocol 4.2.1 with extra steps from Section 4.2.1, Protocol 4.3 to refer to the protocol in Section 4.3, and Protocol 4.4 to refer to the protocol in Section 4.4.

### 4.4 Handling Multiple Candidates

When Alice decides to tentatively transfer her swap option to Carol, it could be the case that Carol is intending to just grief Alice. In the worst case, Alice will have her option locked for $6\Delta$ by Carol that simply doesn’t participate in the Replace/Revert Phase of Protocol 4.2.

To combat this, Alice might want to initiate several concurrent transfers with multiple buyers at once.

A straightforward solution to this is, after Alice gets her position back from a failed transfer with Carol, Alice can call mutateLockLeader() again to indicate she wants to transfer her option to a new option buyer Carol$_{i+1}$. In the worst case, in this protocol, Alice would spend $6\Delta$ with her option locked for each potential buyer who doesn’t comply.

One question then is, can we have Alice potentially transfer her position to multiple buyers at the same time, or with less waiting time than $6\Delta$ in between each new buyer? Although we cannot let Alice transfer her option to multiple buyers simultaneously, we can create enough overlap between each of the potential to reduce Alice’s waiting time for each potential buyer. The protocol description below outlines how Alice is able to reduce her waiting time for each potential buyer to $4\Delta$ rather than $6\Delta$.

The main idea of the protocol is to use to run a concurrent version of Protocol 4.2 for each potential buyer. Alice assigns each potential buyer a ticket (sequence number). A shared counter is synchronized between the $AB$ and $BA$ contracts. The shared counter assigns an ordering to each of the potential buyers to execute their respective versions of Protocol 4.2. In this way, it serves as a first-come first-serve mechanism for Alice to sell her swap option.
We will use Carol\textsubscript{i} to denote the buyer who is assigned the \(i\)-th sequence number by Alice. On each contract \(AB\) and \(BA\), each potential buyer Carol\textsubscript{i} will have their own associated state structure \(state_i\), similar to Protocol 4.2. This keeps track of the state relevant to the 3 different phases from Protocol 4.2.

The protocol is defined as follows:

- Initially \(counter\) on \(AB\) and \(BA\) contracts is initialized to 0.
- For each potential buyer, Alice assigns a unique sequence number \(seq\) (starting from 0 and growing by 1 for each assignment) to them, which represents that party’s position in the queue for a tentative replacement. If Alice assigns the same sequence number to different parties and sends both of them to the \(AB/BA\) contracts, the conflict is resolved via the Consistency Phase from the original Protocol 4.2. It is the same as when Alice deviates by sending inconsistent mutation transactions to both contracts, in which case Bob contests and the tentative transfer is reverted.
- Alice sends mutation transactions as in Protocol 4.2. The main difference is that the mutation transactions now include a sequence number \(seq\) for the candidate replacement. The \(mutate\text{LockLeader}()\), \(contest\text{Leader}()\) functions all take this sequence number as parameters as part of the mutation. If \(counter == seq\), the transaction is accepted. Otherwise, the transaction is rejected. If Alice is conforming, \(counter\) should be synchronized on \(AB/BA\) within \(Δ\). If a tentative transfer is in progress, a new mutation transaction with \(seq == counter + 1\) will be accepted only if 4A has elapsed after the first mutation. A mutation transaction with \(seq == counter\) can serve as mutation transaction for a contest of the current tentative transfer. After the tentative change is reverted, i.e. \(revert\text{Leader}()\) is called, the \(counter\) is incremented by 1. This design can be optimized by accepting the mutation transactions with larger sequence numbers and store them in a queue for future use. These can then take effect immediately after 4A has elapsed after the current tentative mutation happens and \(revert\text{Leader}()\) has been called. The reason why 4A is sufficient interval between the execution of base protocols for two potential buyers is that, after 4A, there is only 2A left for the mutation to be reverted if the previous potential buyer gives up. The later arriving buyer can use this window to finish its consistency phase and after it ends, its replace phase can start without waiting.
- A mutation transaction can take effect (The corresponding candidate can call \(replace\text{Leader}()\) ) only after the previous mutation transaction has expired, meaning it is reverted and Alice regains the position. Then after that the new candidate can execute Protocol 4.2 to replace Alice.

The protocol for transferring the follower position to multiple candidates is similar.

## 5 SECURITY PROPERTIES AND PROOF

### 5.1 General Transfer Properties

Our leader and follower transfer protocols overlap in some properties they satisfy. We outline those properties here.

Recall that assets escrowed in the contracts are called principal. The principals involved in the swap option are always Alice’s principal and Bob’s principal. Here, the original option provider is Bob (follower) and the option owner is Alice (leader). In the leader transfer protocol and the follower transfer protocol, what is transferred is a position in the swap option. If Alice transfers her position it is to Carol, and Bob to David. We say one party owns a position in an option if they are option owner (Alice/Carol): one can release a secret, receiving Bob’s principal by relinquishing Alice’s, or let the option expire and Alice’s principal is refunded to them, or option provider (Bob/David): one provides the option owner the right, but not obligation, for the exchange of Alice’s principal with Bob’s.

From now on, we call a leader/follower who wants to transfer their position as position seller, and the one who wants to replace them as position buyer.

- **Liveness:** In a transfer of positions, if all parties are conforming, then leader/follower transfers their position to a buyer and the position seller gets proper payment from the buyer.
- **Transfer independence:** A compliant position seller (leader/follower) transferring their position to another compliant position buyer, can successfully transfer their position without the cooperation of a third counterparty (follower/leader).  
- **Non-blocking transfer with adversarial counterparty:** If a compliant position seller (leader/follower) is transferring their position to another compliant position buyer, a third counterparty (follower/leader) cannot interfere with the transfer.
- **Transfer atomicity:** If a compliant position seller (leader/follower) loses their position to the buyer, then they receive the expected principal from the position buyer.

**No UNDERWATER for a conforming party (Safety):** UNDERWATER means a party loses their outgoing principal without getting their incoming principal. No UNDERWATER guarantees a conforming party’s safety since it will never end up with losing their principal without getting principals from others.

**Theorem 1.** Protocol 4.2 and 4.3 satisfy No UNDERWATER for Alice, Bob, Carol and David:

- If Alice is conforming, then if she loses her principal, she either gets Bob’s or Carol’s principal, or both.
- If Bob is conforming, then if he loses his principal, he gets Alice’s principal or David’s, or both.
- If Carol is conforming, then if she loses her principal, she gets Alice’s principal or Bob’s principal, or both.
- If David is conforming, then if he loses his principal, he gets Alice’s principal or Bob’s principal, or both.

**Proof.** See Appendix A.2.2. □

### 5.2 Leader Transfer Properties

We first demonstrate how our leader transfer protocol satisfies the general transfer properties from 5.1.

Because the detailed proofs of the following results are long and similar to those that we will see for the follower transfer protocol 5.3, we omit them here. See the Appendix Section A.2 for details.
We now demonstrate how our follower transfer protocol also satisfies the following unique properties.

**Theorem 2.** Protocol 4.2 satisfies liveness: If Alice, Bob, and Carol are all conforming, then Alice gets Carol’s principal, Carol gets Alice’s position, and Bob maintains his position.

**Theorem 3.** Protocol 4.2 satisfies transfer independence: Alice can transfer her position to Carol without Bob’s participation.

**Theorem 4.** Protocol 4.2 satisfies non-blocking transfer: Alice can transfer her position to Carol even if Bob is adversarial.

**Theorem 5.** Protocol 4.2 satisfies transfer atomicity: If Alice loses her position in the swap, then she can claim Carol’s principal.

In addition to the properties from Section 5.1, it is desirable for a leader transfer protocol to have one additional property. Given two protocols $P, P'$ that satisfy non-blocking transfer with adversarial counterparty (Bob), Bob is called altruistic in $P'$ with respect to $P$ if Bob conforming in $P'$ terminates faster than Bob conforming in $P$ in executions where all parties are conforming. Importantly, it is not necessary that Bob should be incentivized to choose to follow $P'$ over $P$.

When Alice and Carol are conforming, even though adversarial Bob cannot block the transfer, Bob can delay the transfer for a few rounds. The following next property we introduce is a slightly stronger version of non-blocking property that is ideal for a leader transfer protocol.

**Theorem 6.** Protocol 4.2.1 satisfies timely transfer with altruistic Bob.

**Proof.** Since Protocol 4.2 is non-blocking by Theorem 4, it is enough to show that Bob is altruistic in Protocol 4.2.1 with respect to Protocol 4.2. By assumption all parties are conforming. In both protocols, Carol will have created MutSwapCA by \texttt{startLeader} + $\Delta$ and Alice will have called \texttt{mutateLockLeader()} on $AB$ and $BA$ by \texttt{startLeader} + 2$\Delta$. In Protocol 4.2.1 Bob now calls \texttt{approveLeader()} on MutSwapAB and MutSwapBA by \texttt{startLeader} + 3$\Delta$. Thus the Replace/Revert Phase for Protocol 4.2.1 begins because Bob skips the contest phase. In Protocol 4.2, since Alice is compliant, her signatures on MutSwapAB and MutSwapBA are consistent. Thus Bob won’t call \texttt{contest()} on MutSwapAB or MutSwapBA during this 2$\Delta$ Consistency Phase. Carol waits 2$\Delta$ after \texttt{startLeader} + 2$\Delta$. In this case, the Replace/Revert Phase begins at \texttt{startLeader} + 4$\Delta$. Thus Bob is altruistic in Protocol 4.2.1 with respect to Protocol 4.2.\qed

**5.3 Follower Transfer Properties**

We now demonstrate how our follower transfer protocol also satisfies the general transfer properties from Section 5.1.

**Theorem 7.** Protocol 4.3 satisfies liveness: If Alice, Bob, and David are all conforming, and Alice doesn’t reveal $A_1$, then Bob gets David’s principal, David gets Bob’s position, and Alice maintains her position.

**Theorem 8.** Protocol 4.3 satisfies transfer independence: Bob can transfer his position to David without Alice’s participation.

**Theorem 9.** Protocol 4.3 satisfies non-blocking transfer: Bob can transfer his position to David even if Alice is adversarial.

Note that Alice is not adversarial by choosing to reveal her secret $A_1$ since this just means she is exercising her option. This is consistent with how she should behave in the original swap protocol.

**Theorem 10.** Protocol 4.3 satisfies transfer atomicity: If Bob loses his swap position, then he can claim David’s principal.

On top of the base properties described in Section 5.1, any protocol transferring Bob’s position should also be:

1. **Optionality preserving:** If Alice is compliant, she never loses her ability to exercise the original swap option.

   Namely, it is unfair for Alice to temporarily lose her right to use her option without her consent.

2. **Theorem 11.** Protocol 4.3 is optionality preserving.

   **Proof.** It’s enough to show that Alice can always claim the principal on $BA$, if she reveals her secret $A_1$ by timeout $T$. The added functions from Protocol 4.3, namely \texttt{mutateLockFreeFollower()}, \texttt{replaceFollower()}, and \texttt{revertFollower()} only modify the state of follower\_mutation. However, any call to \texttt{claim()} on $BA$ doesn’t depend on the state of follower\_mutation. Namely, any call to \texttt{claim()} is independent of the state of the follower transfer protocol. So its enough that Alice reveal $A_1$ before $T$.\qed

There is nothing preventing both Protocol 4.2 and Protocol 4.3 to be run concurrently. That is, if Alice wants to transfer her position to Carol, and Bob wants to transfer his position to David, they can both do so simultaneously. This follows immediately from the fact that the parts of the contracts managing the state for each individual protocol are entirely disjoint and cannot affect each other.

**5.4 Leader Transfer (Multiple Buyers) Properties**

Besides the general properties mentioned above, the protocol that handles multiple candidate buyers (Protocol 4.4) satisfies the following unique properties.

- **First-come first-serve (FCFS):** If Alice is conforming (the buyer who tentatively pays to Alice earlier gets a smaller sequence number), then the earlier arriving buyer has priority to replace Alice’s position.

- **Starvation freedom:** If Alice is conforming and a conforming buyer, say Carol, gets a sequence number $j$, then Carol can replace Alice’s position if she is conforming and all Carol$_i$ where $i < j$ gives up the replacement.

**Theorem 12.** If Alice is conforming, the counter on both contracts are synchronized with inconsistency for at most $\Delta$.

**Proof.** The counter is initialized as 0 and it incremented only after \texttt{revertLeader()} is enabled. The \texttt{revertLeader()} is enabled only when $6\Delta$ elapses after the start time of \texttt{mutateLockLeader()} being...
called. Since the start time of $\text{mutateLock}()$ on both contracts are staggered by at most $\Delta$, when the counter is incremented on one contract, it can be incremented on another contract within $\Delta$. □

**Theorem 13.** If Alice is conforming, Protocol 4.4 satisfies FCFS.

**Proof.** If Alice is conforming, she issues $\text{mutateLockLeader}()$ transactions with ascending sequence numbers to different potential buyers, in an arrive-earlier-smaller-sequence manner. For every potential buyer, a base transfer protocol (Protocol 4.2) is run as normal. Without loss of generality, at time $t$, suppose on AB contract the counter is $i$ and on BA the counter is $j = i - 1$. On AB, Carol has priority. By Theorem 12, we know the counter on BA will become $i$ within $\Delta$ and then Carol has priority. The reason why the counter is incremented from $j$ to $i$ is that Carol gives up the replacement. Then we see on both contracts Carol has priority. It is obvious Carol, where $k \geq i + 1$ cannot start their replacement unless Carol gives up, i.e. the replacement phase ends and the mutation is reverted, since the counter is $i$ now. Therefore the protocol satisfies FCFS.

**Theorem 14.** If Alice is conforming, Protocol 4.4 is starvation free.

**Proof.** Similar to Proof 5.4, if Alice is conforming and a buyer Carol$_j$ gets a sequence number $j$, then the base protocol for Carol$_j$ can start underlying others’ base protocols. If all Carol$_i$ where $i < j$ gives up the replacement, then the counter is incremented to $j$ and then Carol$_j$ can call $\text{replaceLeader}()$ to replace Alice. □

### 6 RELATED WORK

Cross-chain options have their origin in cross-chain atomic swap protocols. A cross-chain atomic swap enables two parties to exchange assets across different blockchains. An atomic swap is implemented via hashed timelock contracts (HTLC) [17]. There are a variety of protocols proposed [1, 8, 10, 24] and implemented [4, 6]. Herlihy et al. proposed protocols for atomic multi-party swaps [13] and more general atomic cross-chain deals [14].

Several researchers [11, 12, 16, 25] have noted that most two-party swap protocols effectively act as poorly-designed options [15], because one party has the power to decide whether to go through with the agreed-upon swap without compensation for its counterpart.

A number of proposals [7, 11, 12, 16, 18, 23, 25] address the problem of optionality in cross-chain atomic swaps by introducing some form of premium payment, where a party that chooses not to complete the swap pays a premium to the counterpart. Robinson [19] proposes to reduce the influence of optionality by splitting each swap into a sequence of very small swaps. Han et al. [11] quantified optionality unfairness in atomic swap using the Cox-Ross-Rubinstein option pricing model [5], treating the atomic swap as an American-style option. The Black-Scholes (BS) Model [3] can be used to estimate the value of European-style options.

Liu [16] proposed an alternative approach where option providers are are paid up-front for providing optionality, as in the conventional options market. In this protocol, Alice explicitly purchases an option from Bob by paying him an nonrefundable premium. Tefaghi et al. [21] proposed a similar protocol which enables Alice to deposit her principal later than Bob. None of these works have considered how to close an option owner’s position by transferring that option to a third party.

There are protocols that allow blockchains to communicate (cross-chain proofs), however they either rely on external third parties [9] or their applicability requires the introduction of centralized services, modifications to existing software, and doesn’t guarantee reliable message delivery [20].

### 7 REMARKS AND CONCLUSIONS

The transferable swap protocols presented here have certain limitations. As described earlier, the protocols require multiple time-out periods. If all parties are responsive, these timeouts should not affect the performance of normal executions, but they could lead to long worst-case executions. The protocols also include a “hard timeout” where Alice and Carol pause to give Bob a chance to object to a malformed transfer. Bob can be paid an incentive to respond quickly, but he could slow down (but not stop) the protocols execution.

The ability for Bob to be able to report inconsistent state changes between multiple blockchains is integral to the design of our protocols. Adding such functionality was not necessary in simpler cross-chain protocols like the two-party swap. In future work, we hope to better understand how the complexity of a cross-chain deal relates to the necessity of this consistency phase.

In practice, our proposed protocols are applicable in contexts where the duration until the option expires is much larger than the execution time of the protocol. This ensures the option can be traded, if desired, between multiple buyers/sellers before the expiration time.

Another practical consideration is the tolerance of the counterparty (Bob) to many potential rounds of protocol interaction where he could incur additional transaction costs. If this is unacceptable to Bob he could choose to only participate in option contracts that are non-tradeable. Alternatively, additional premiums could be added to our proposed protocols to mediate potential future risk Bob would experience as a result of having to contest inconsistent state changes. We expect premiums could be added in a similar fashion to how they were added to the standard two-party swap [25].

Our protocols require Alice to lock her principal first to acquire an option from Bob. Prior work has allowed Alice to delay the deposit of her principal [21]. Our protocols focus on the transferability of options. It is possible to extend our protocols to options as in [21] and make them transferable.

### REFERENCES

[1] AtomicDEX. November, 2019. First AtomicDEX Stress Test Successfully Completed. https://atomicdex.io/first-atomicdex-stress-test-successfully-completed/.

[2] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1986. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[3] Fischer Black and Myron Scholes. 2019. The pricing of options and corporate liabilities. In *World Scientific Reference on Contingent Claims Analysis in Corporate Finance: Volume 1: Foundations of CCA and Equity Valuation*. World Scientific, 3–21.

[4] Sean Bowe and Daira Hopwood. 2017. Flushed Time-Locked Contract transactions. https://github.com/bitcoin/bips/blob/master/bip-0199.medialwiki. As of 9 January 2018.

[5] John C Cox, Stephen A Ross, and Mark Rubinstein. 1979. Option pricing: A simplified approach. *Journal of financial Economics* 7, 3 (1979), 229–263.

[6] Decred. 2018. https://github.com/decred/atomicswap. As of 18 February 2021.
[7] Thomas Eizinger, Lloyd Fournier, and Phillip Hoenisch. 2018. The state of atomic swaps. http://diyhpl.us/wiki/transcripts/scalingbitcoin/tokyo-2018/atomic-swaps/.

[8] Altcoin.io Exchange. October 2017. The First Ethereum <>Bitcoin Atomic Swap. https://blog.altcoin.io/the-first-ethereum-bitcoin-atomic-swap-7b9ef1837388. Altcoin.io Exchange.

[9] Christopher Goes. 2020. The interblockchain communication protocol: An overview. arXiv preprint arXiv:2006.15918 (2020).

[10] Trey Griffith. July, 2018. Sparkswap: Trade across blockchains without custody risk. https://medium.com/sparkswap/sparkswap-trade-across-blockchains-without-custody-risk-8d6f0803e8.

[11] Runchao Han, Haoyu Lin, and Jiangshan Yu. 2019. On the optionality and fairness of Atomic Swaps. In Proceedings of the 1st ACM Conference on Advances in Financial Technologies. ACM, Zurich Switzerland, 62–75. https://doi.org/10.1145/3318041.3355469

[12] Ethan Heilman, Sebastien Lipmann, and Sharon Goldberg. 2019. The arwen trading protocols. https://www.arwen.io/whitepaper.pdf

[13] Maurice Herlihy. 2018. Atomic cross-chain swaps. In Proceedings of the 2018 ACM symposium on principles of distributed computing (PODC ’18). ACM, New York, NY, USA, 245–254. https://doi.org/10.1145/3212734.3212736 Number of pages: 10 Place: Egham, United Kingdom tex.acmid: 3212736.

[14] Maurice Herlihy, Barbara Liskov, and Liuba Shrira. 2019. Cross-Chain Deals and Adversarial Commerce. Proc. VLDB Endow. 13, 2 (Oct. 2019), 100–113. https://doi.org/10.14778/3364324.3364326

[15] Desmond J. Higham. 2009. An introduction to financial option valuation: mathematics, stochastics and computation (4. printing ed.). Cambridge Univ. Press, Cambridge.

[16] James A. Liu. 2020. Atomic Swaptions: Cryptocurrency Derivatives. arXiv:1807.08644 [cs, q-fin] (March 2020). http://arxiv.org/abs/1807.08644

[17] Tier Nolan. May, 2013. Alt chains and atomic transfers. https://bitcoin.talk.org/index.php?topic=193281.0. Bitcoin Forum.

[18] Komodo Platform. July 2019. Advanced blockchain technology, focused on freedom. https://docs.komodoplatform.com/basic-docs/start-here/core-technology-discussions/introduction.html#note-on-changes-since-whitepaper-creation-cr-2019.

[19] Dan Robinson. 2019. HTLCs Considered Harmful. http://diyhpl.us/wiki/transcripts/stanford-blockchain-conference/2019/htlcs-considered-harmful/

[20] Peter Robinson. 2021. Survey of crosschain communications protocols. Computer Networks 200 (2021), 108488.

[21] Mojtaba Tefagh, Fateme Bagheri, Amirhossein Khajehpour, and Melika Abdil. October, 2020. Capital-free Futures Arbitrage. https://doi.org/10.13140/RG.2.2.31609.90729/1

[22] Jiabua Xu, Damien Ackerer, and Alevtina Dubovitskaya. 2021. A Game-Theoretic Analysis of Cross-Chain Atomic Swaps with HTLCs. arXiv:2011.11325 [cs] (April 2021). http://arxiv.org/abs/2011.11325 arXiv: 2011.11325

[23] Yingjie Xue and Maurice Herlihy. 2021. Hedging Against Sore Loser Attacks in Cross-Chain Transactions. In ACM Symposium on Principles of Distributed Computing.

[24] Jake Yocom-Piatt. September, 2017. On-Chain Atomic Swaps. https://blog.decred.org/2017/09/26/On-Chain-Atomic-Swaps/. Decred Blog.

[25] ZmnSCPxj. 2018. An Argument For Single-Asset Lightning Network. https://lists.linuxfoundation.org/pipermail/lightning-dev/2018-December/001752.html. As of 10 January 2021.
A.1 Contracts

```solidity
contract MutSwapAB {
    struct FollowerMutation {
        // Signature by follower to allow candidate to take its position
        Sig voucher;
        // Candidate party to replace follower
        address candidate_follower;
        // Hashlock used to replace follower
        uint replace_hash_lock;
        // Time mutation begins
        uint start_time;
        // Flag for freezing asset when a tentative replacement is happening
        bool mutating;
        // Controls whether asset is locked or not during mutation
        bool can_lock_asset;
    }

    struct LeaderMutation {
        // Signature by follower to allow candidate to take its position
        Sig voucher;
        // Candidate party to replace follower
        address candidate_leader;
        // Hashlock used to replace follower
        uint replace_hash_lock;
        // Hashlock for exercising option
        uint swap_hash_lock;
        // Time mutation begins
        uint start_time;
        // Used for optimistic execution of protocol
        bool approved;
        // Flag for freezing asset when a tentative replacement is happening
        bool mutating;
        // Controls whether asset is locked or not during mutation
        bool can_lock_asset;
    }

    // State information for base swap protocol
    Asset asset; // Reference to preferred token contract
    address sender; // Current sender of escrowed funds
    address receiver; // Current receiver of escrowed funds
    address leader; // Leader of the original swap protocol
    address follower; // Follower of the original swap protocol
    uint swap_hash_lock; // Hashlock of the swap protocol
    uint T_AB; // Timeout for locked asset
    
    // Initial mutation states
    leader_mutation.mutating = false;
    leader_mutation.approved = false;
    follower_mutation.mutating = false;

    // Controls whether leader/follower positions can be changed
    bool mutable;

    function MutSwapAB(Asset _asset, uint start, uint dT, address _sender, address _receiver, uint _swap_hash_lock, address _leader, address _follower, bool _mutable) {
        require(msg.sender == _sender); // Sender can only escrow their own funds
        require(leader != _follower); // Leader and follower should be distinct
        require(leader == _sender); // Leader and follower should be distinct
        require(follower == _receiver); // Leader and follower should be distinct
        this.mutable = _mutable;

        // For AB contract, asset is actively locked during mutation
        asset = _asset;
        follower = _follower;
        leader = _leader;
        swap_hash_lock = _swap_hash_lock; // Hashlock for the initial swap

        // Leader and follower transfer their own funds
        asset.send(address(this));
        T_AB = start + (dT+1)*delta; // Sender is Alice
        if(_mutable){
            // For AB contract, asset is actively locked during mutation
            leader_mutation.can_lock_asset = true;
            follower_mutation.can_lock_asset = true;
        }
        
        function claim(string secret) {
            // If a previous mutation lock was never completed, revert to original swap
            if(_mutable){
                revertLeader();
            }
            revertFollower();
        }

        require(msg.sender == receiver.id); // Only receiver can call claim
        require(now <= T_AB); // Must be before timeout
        require(H(secret) == swap_hash_lock); // Claim conditional on revealing secret
        asset.send(receiver);
    }

    function refund(){
        revertFollower();
    }
}
```

require

171
require

170
}
revertFollower();

167
if

166
165
164
163
require

162
function

160
leader_mutating.start_time =

158
leader_mutating.mutator =

155
leader_mutating.mutating =

151
require

150
require

148
require

146
if

144
if

138
if

132
require

130
require

129
require

128
require

125
require

124
require

121
if

119
if

118
//If a previous mutation lock was never completed, revert
to original swap

120
if(mutable){

121
if(leader_mutation.mutating){

122
revertLeader();

123
}

124
if(follower_mutation.mutating){

125
revertFollower();

126
}

127
}

128
require(msg.sender == sender); //Only sender can call refund

129
require(now > T_AB); //After lock has timed out

130
asset.send(sender);

131
}

132

133
function mutateLockLeader(Sig sig,address _candidate_sender, uint _replace_hash_lock, uint _swap_hash_lock){

134
require(mutable);

135
//If a previous mutation lock is stale, then call revert
to allow for a new mutation lock to be made

136
if(leader_mutation.mutating){

137
revertLeader();

138
}

139
require(!leader_mutation.mutating); //Only one mutate lock at a time

140
require(msg.sender == sender || msg.sender == receiver); //Only sender or receiver can mutate

141
if(msg.sender == leader){ //Alice has less time to call mutateLock

142
require(T_AB > now + 8*delta);

143
}

144
else if(msg.sender == follower){ //Bob is given more time to call mutateLock in response to Alice

145
require(T_AB >= now + 7*delta);

146
}

147
require(sig.valid(leader)); //Requires Alice's sig of Carol's hashlocks

148
require(sig.msg() == [_replace_hash_lock,_swap_hash_lock]); //The msg has to just be the candidate's hashlocks

149
leader_mutation.mutating = true;

150
leader_mutation.voucher = sig; //Proof that candidate was approved

151
leader_mutation.candidate_sender = _candidate_sender;

152
leader_mutation.mutator = msg.sender;

153
leader_mutation.replace_hash_lock = _replace_hash_lock;

154
leader_mutation.swap_hash_lock = _swap_hash_lock;

155
leader_mutation.start_time = now; //Used for flexible timeouts in the transfer

156
}

157
function mutateLockFollower(Sig sig,address _candidate_receiver, uint _replace_hash_lock){

158
require(mutable);

159
//If a previous mutation lock is stale, then call revert
to allow for a new mutation lock to be made

160
if(follower_mutation.mutating){

161
revertFollower();

162
}

163
require(msg.sender == follower); //Only Bob can call

164
require(!follower_mutation.mutating);

165
require(follower_mutation.lock_asset); //Only can call

166
mutateLockFollower on AB contract

167
require(sig.valid(follower)); //Bob's valid signature

168
require(sig.msg() == [_replace_hash_lock]);

169
require(T_AB >= now + 3*delta); //Need enough time for David to call claim

170
follower_mutation.mutating = true;

171
follower_mutation.voucher = sig; //Proof that candidate was approved by previous owner of the position

172
follower_mutation.candidate_receiver = _candidate_receiver;

173
follower_mutation.replace_hash_lock = _replace_hash_lock;

174
follower_mutation.start_time = now; //Used for flexible timeouts in the transfer

175
}

176
function replaceLeader(string secret){

177
require(mutable);

178
require(leader_mutation.mutating);

179
bool candidate_round = false;

180
bool follower_round = false;

181
//If Bob mutated, can skip contesting phases since he is implicitly approving of the signature he called mutateLock with

182
if(leader_mutation.mutator == follower ||

183
leader_mutation.mutator == leader &

184
leader_mutation.approved)){ //Can immediately call replace once Bob has mutated

185
//Carol can replace once Bob has called mutate

186
candidate_round = (msg.sender == leader_mutation.candidate_sender) & (4*delta >= now - leader_mutation.start_time > 0);

187
//Bob can call replace

188
follower_round = (msg.sender == follower) & (6*delta >= now - leader_mutation.start_time > 0);

189
}

190
else{ //Contesting (pessimistic case)

191
//Carol can replace in round right after last chance at contesting

192
candidate_round = (msg.sender == leader_mutation.candidate_sender) & (4*delta >= now - leader_mutation.start_time > 2*delta);

193
//Bob can replace in the 3 rounds right after last chance at mutating round

194
follower_round = (msg.sender == follower) & (6*delta >= now - leader_mutation.start_time > 0);

195
}

196
require(candidate_round || follower_round);

197
require(H(secret) == leader_mutation.replace_hash_lock);

198
sender = leader_mutation.candidate_sender; //Carol takes refund optionality from Alice

199
swap_hash_lock = mutation.swap_hash_lock;

200
leader_mutation.approved = false;

201
leader_mutation.mutating = false;

202
leader_mutation.mutation.clear();

203
}

204
require(candidate_round || follower_round);

205
require(H(secret) == leader_mutation.replace_hash_lock);

206
sender = leader_mutation.candidate_sender; //Carol takes refund optionality from Alice

207
swap_hash_lock = mutation.swap_hash_lock;

208
leader_mutation.approved = false;

209
leader_mutation.mutating = false;

210
leader_mutation.mutation.clear();

211
}

212
//If no contesting has occurred, transfer can be complete

213
function replaceFollower(string secret){

214
require(mutable);

215
require(follower_mutation.mutating);
221 require(2*delta >= now - follower_mutation.start_time > 0); //2 rounds given for replacement
222 require(H(secret) == follower.mutation.replace_hash_lock);
223 receiver = follower.mutation.candidate_receiver; // David takes refund optionality from Bob
224
225 function contestLeader(Sig sig, string secret){
226 require(mutable);
227 require(msg.sender == follower); //Only allow Bob to call this
228 require(leader.mutation.mutating); //Can only contest a mutation if one has occurred
229 require(leader.mutation.approved); //Cannot contest after approval
230 require(leader.mutation.mutator == leader); //Can only contest a mutation when Alice called it, not Bob
231 require(2*delta >= now - leader.mutation.start_time > 0); //Can contest only in the 2 rounds after mutation
232 //Can contest if Alice creates two inconsistent signatures or tries to reveal preimage of hashlock too early
233 require(sig.valid(leader) || H(secret) == swap_hash_lock); //Checks if Alice reported inconsistent sigs
234 leader.mutation.clear();
235 leader.mutation.mutating = false;
236 }
237
238 function approveLeader(){
239 require(mutable);
240 require(msg.sender == leader); //Can only approve after approval
241 require(leader.mutation.mutator == leader); //Can only call approve when Alice was the one who called mutate
242 require(2*delta >= now - leader.mutation.start_time > 0); //Can approve only in the 2 rounds after mutation
243 leader.mutation.approved = true;
244 }
245
246 //Issue is really in the sequential consistent case
247 //Bob can call approve in any next two rounds after a mutation
248 function revertLeader(){
249 require(mutable);
250 require(msg.sender == leader); //Only allow Bob to call this
251 require(leader.mutation.mutating); //Can only approve after a mutation has begun
252 require(leader.mutation.mutator == leader); //Can only call approve when Alice was the one who called mutate
253 require(2*delta >= now - leader.mutation.start_time > 0); //Can approve only in the 2 rounds after mutation
254 leader.mutation.approved = false;
255 }
// If a previous mutation lock was never completed, revert
if (leader_mutation.mutating) {
  revertLeader();
} 

} 

require(msg.sender == receiver.id); // Only receiver can call claim 
require(now <= T_BA); // Mint be before timeout 
require(H(secret) == swap_hash_lock); // Claim conditional on revealing secret 
asset.send(receiver); 

} 

function refund() {

} 

if (mutable) {
  if (leader_mutation.mutating) {
    revertLeader();
  } 
  if (follower_mutation.mutating) {
    revertFollower();
  }

} 

require(msg.sender == sender); // Only sender can call refund 
require(now > T_BA); // After lock has timed out 
asset.send(sender); 

} 

function mutateLockLeader(Sig sig, address _candidate_receiver , uint _replace_hash_lock,uint _swap_hash_lock) {

} 

require((leader_mutation.mutating); // Only one 
mutate_lock at a time 
require(msg.sender == sender || msg.sender == receiver); // Only sender or receiver can mutate 
if (msg.sender == leader) { // Alice has less time to call 
  require(T_BA >= now + 7*delta); 
} else if (!msg.sender == follower) { 
  // Bob is given more time to call mutateLock in 
  response to Alice 
  require(T_BA >= now + 6*delta); 
}

require(sig.valid(leader)); // Requires Alice's sig of 
Carol's hashlocks 
require(sig.msg() == [replace_hash_lock,_swap_hash_lock] ); // The msg has to just be the candidate's 
hashlocks 
leader_mutating.mutating = true; 
leader_mutating.voucher = sig; // Proof that candidate was approved 
leader_mutating.candidate = _candidate_receiver; 
leader_mutating.mutator = msg.sender; 
leader_mutating.replace_hash_lock = _replace_hash_lock; 
leader_mutating.swap_hash_lock = _swap_hash_lock; 
leader_mutating.start_time = now; // Used for flexible 
timeouts in the transfer

}
function mutateLockFreeFollower($sig sig, address _candidate_sender, uint _replace_hash_lock)
    require(mutable);
    require(msg.sender == follower); // Only Bob can call
    require(!follower.mutation.mutating);
    require(!follower.mutation.lock_asset);
    require(!leader_mutation.mutating);
    require(_replace_hash_lock);
    require(T_BA >= now + 2*delta); // Used for flexible timeouts in the transfer
    follower.mutation.mutating = true;
    follower.mutation.voucher = sig; // Proof that candidate was approved
    follower.mutation.candidate_receiver = _candidate_sender;
    follower.mutation.replace_hash_lock = _replace_hash_lock;
    follower.mutation.start_time = now;
    // Need enough time for David to call claim

function replaceLeader(string secret){
    require(mutable);
    require(leader.mutation.mutating);
    bool candidate_round = false;
    bool follower_round = false;
    // If Bob mutated, can skip contesting phases since he is implicitly approving of the signature he called
    if(leader.mutation.mutator == follower)
    {
        leader.mutation.mutator = leader && leader.mutation.approved(); // Can immediately call replace once Bob has mutated
        candidate_round = msg.sender == leader.mutation.mutator;
        candidate_receiver &\& (4*delta >= now - leader.mutation.start_time > 0);
        // Bob can call replace
        follower_round = (msg.sender == follower) &\& (6*delta >= now - leader.mutation.start_time > 0);
        // If Bob mutated, can skip contesting phases since he is implicitly approving of the signature he called
    } else // Contesting (pessimistic case)
    {
        candidate_round = msg.sender == leader.mutation.mutator;
        candidate_receiver &\& (4*delta >= now - leader.mutation.start_time > 2*delta);
        // Bob can replace in the 3 rounds right after last chance at mutating round
        follower_round = (msg.sender == follower) &\& (6*delta >= now - leader.mutation.start_time > 0);
    }

function approveLeader()
{
    require(mutable);
    require(msg.sender == follower); // Only Bob can approve
    require(!leader.mutation.mutating); // Can only approve after a mutation has begun
    require(leader.mutation.mutator == leader); // Can only approve when Alice called it, not Bob
    follower.mutation.mutating = false;
    follower.mutation.mutating = false;
    leader.mutation.approved = false;
    leader.mutation.approved = false;
    leader.mutation.approved = true;
}
A.2 Proofs

A.2.1 Misc Proofs. First, we provide proofs for the properties of the leader transfer protocol.

Lemma 15. If Alice and Carol are both conforming in the Mutate Lock Phase, then both AB and BA are mutate locked with \((H(C_1), H(C_2))\) by startLeader + 2Δ.

Proof. If Carol is conforming, she escrows her principal to Alice by time \(R + \Delta\). Alice, after observing the creation of CA, will then call mutateLockLeader() within \(R\) on both AB and BA, passing \((H(C_1), H(C_2))\) as the transfer hashlock and the new swap hashlock. For any contract, if it is not mutate locked, then the mutateLockLeader() sent by Alice can be included in the blockchain by startLeader + 2Δ. If the mutateLockLeader() transaction sent by Alice can not be included in the blockchain due to someone else calling it first, it must be that Bob called mutateLockLeader() on the contract with \((H(C_1), H(C_2))\) before her. In any case, within startLeader + 2Δ, both contracts are mutate locked by \((H(C_1), H(C_2))\). □

If there is a successfully call to mutateLockLeader() on AB with \((H(C_1), H(C_2))\), let the first time that this transaction is called be \(t_{AB}\). Similarly, let \(t_{BA}\) be the first time mutateLockLeader() is called on BA with \((H(C_1), H(C_2))\).

Theorem 2. Protocol 4.2 satisfies liveness: If Alice, Bob, and Carol are all conforming, then Alice gets Carol’s principal, Carol gets Alice’s position, and Bob maintains his position.

Proof. If Alice starts the transfer procedure, and all parties are conforming, then Carol owns the option and paid her principal to Alice. The argument is as follows.

By Lemma 15, we know mutateLockLeader() has been called by \(R + 2\Delta\) on AB and BA with \((H(C_1), H(C_2))\). After max\(\{t_{AB}, t_{BA}\} + 2\Delta \leq \text{startLeader} + 4\Delta\), Carol will know that Bob cannot successfully call contest() on either AB or BA. Carol then releases her secret \(C_1\) to AB and BA by max\(\{t_{AB}, t_{BA}\} + 3\Delta \leq \text{startLeader} + 5\Delta\). Then AB and BA are transferred to a new state where the swap hashlock is \(H(C_2)\), the sender of AB contract becomes Carol, and the receiver of BA contract becomes Carol. At this point, these contracts can be more appropriately renamed as \(CB\) and \(BC\) respectively. That is to say, Carol has set up a swap with Bob using her swap hashlock \(H(C_2)\). Carol now owns the option. Because the timeout on CA is startLeader + 10\(\Delta\), Alice can use \(C_1\) to redeem Carol’s principal on CA by revealing \(C_1\) by startLeader + 6\(\Delta\). □

Theorem 3. Protocol 4.2 satisfies transfer independence: Alice can transfer her position to Carol without Bob’s participation.

Proof. If Alice and Carol are both conforming, then by Lemma 15, we know mutateLockLeader() will be called with \((H(C_1), H(C_2))\) on both AB and BA by startLeader + 2\(\Delta\). In the worst case, after max\(\{t_{AB}, t_{BA}\} + 2\Delta\), Carol should release \(C_1\) to replace Alice on both AB and BA. This results in the replacement of the old swap hashlock \(H(A_1)\) with \(H(C_2)\). Bob does not have to call any functions to help Carol replace Alice’s role. Thus, conforming Alice and Carol can trade without Bob’s cooperation.

We have proven that conforming Alice and Carol can trade the option without Bob cooperating. Now we look at this property from another perspective. Can Bob actively prevent Alice selling her option to Carol? The following result guarantees Bob cannot block Alice and Carol.

Theorem 4. Protocol 4.2 satisfies non-blocking transfer: Alice can transfer her position to Carol even if Bob is adversarial.

Proof. The only way for Bob to block the transfer of an option is to contest. To contest, Bob needs to show a different mutation signed by Alice or by showing Alice’s secret to the old hashlock. As long as Alice is conforming to the protocol and does not collude with Bob, which is the case in this context, an adversarial Bob cannot forge her signature and successfully call contest(). □

Theorem 5. Protocol 4.2 satisfies transfer atomicity: If Alice loses her position in the swap, then she can claim Carol’s principal.

Proof. Conforming Carol only loses her principal if she releases her secret \(C_1\). She only releases her secret \(C_1\) when she sees both AB and BA are both mutate locked by \((H(C_1), H(C_2))\) and at least one of the conditions are met: (1) Bob has approved both AB and BA; (2) both of his contest windows on AB and BA have ended. Bob can no longer contest this result either due to timeout of his contesting window or Bob has approved this result. Then when Carol sends her secret \(C_1\) to both AB and BA, she replaces Alice’s role and changes the swap hashlock to \(H(C_2)\) thus she owns the option.

Conforming Alice loses her option only if Carol releases her secret \(C_1\) on some contract. We know by Lemma 15 that in the worst case mutateLockLeader() will be called on AB and BA by startLeader + 2\(\Delta\). Thus, the latest time any party can call replace() is startLeader + 8\(\Delta\). So if Alice loses her option there must have
been a replace() call by startLeader + 8Δ. Since CA has timeout startLeader + 10Δ, Alice is guaranteed enough time to claim Carol's principal. □

The remaining proofs are for the properties of the follower transfer protocol.

Lemma 16. In Protocol 4.3, if Bob, David are both compliant, and Alice does nothing, then Bob gets David's principal, and David gets Bob position.

Proof. By assumption, we know the Mutate Phase ends startFollower + 2Δ. At this point, candidate_sender = David.address on BA and candidate_receiver = David.address on AB. By startFollower + 3Δ we know David will reveal D1 to AB and BA. At this point sender = David on BA and receiver = David on AB. Thus David has successfully taken Bob's position. When Bob forwards D1 to the DB, it will arrive no later than startFollower + 4Δ < startFollower + 5Δ. So Bob gets David's principal. □

Theorem 7. Protocol 4.3 satisfies liveness: If Alice, Bob, and David are all conforming, and Alice doesn't reveal A1, then Bob gets David's principal, David gets Bob's position, and Alice maintains her position.

Proof. Because Alice is compliant and doesn't reveal A1, she does nothing according to Protocol 4.3. Since Bob and David are both compliant, the result follows from Lemma 16. □

Theorem 8. Protocol 4.3 satisfies transfer independence: Bob can transfer his position to David without Alice's participation.

Proof. Assume Alice does nothing and Bob, David are compliant. By Lemma 16, the result follows immediately. □

Theorem 9. Protocol 4.3 satisfies non-blocking transfer: Bob can transfer his position to David even if Alice is adversarial.

Proof. Alice's only ability during Protocol 4.3, is to choose to reveal her secret A1. Since Bob and David are compliant, David creates DB by startFollower + Δ. In the worst case, Bob calls mutateLockFreeFollower() on BA at the last available opportunity, at T = 2Δ. Since Bob is compliant, he also calls mutateLockFollower() on AB by T = 2Δ. Since David is compliant, he calls replace() on AB, BA by T = Δ, thereby revealing D1 to Bob. Both replace calls are then completed by startFollower + 3Δ. Bob then has enough time to reveal D1 on DB since its timeout is startFollower + 5Δ. If Alice reveals A1 by T on BA, then David will have Bob's position so he on AB and BA. Thus he can reveal A1 on AB since it has timeout T + Δ. □

Theorem 10. Protocol 4.3 satisfies transfer atomicity: If Bob loses his swap position, then he can claim David's principal.

Proof. Suppose a conforming Bob loses his swap position. By Protocol 4.3, we know mutateLockFollower() and mutateLockFreeFollower() are called on AB and BA respectively by startFollower + 2Δ by Bob. By assumption Bob loses his swap position. Thus David must have revealed D1 on AB or BA before the replaceFollower() timeout. Since each replaceFollower() timeout is 2Δ, the latest David could have revealed D1 is startFollower + 4Δ. Conforming Bob sends D1 to DB which arrives by startFollower + 5Δ. Since the timeout on DB is startFollower + 5Δ, this ensures Bob gets David's principal. □

A.2.2 No UNDERWATER. The goal of this section is to prove Theorem 1.

If the reselling is started by Alice calling mutateLockLeader() on AB and/or BA contract, the state of the corresponding contract will be changed. We first define states of contracts. These are referenced from A.1. A smart contract's state is defined by the following elements:

- sender: sender can call refund() if the receiver does not redeem it before swap hashlock times out.
- receiver: receiver can call claim() and pass the preimage of swap hashlock to get the asset escrowed in the contract.
- mutating: mutating is a bool where mutating=true means the asset is temporarily not redeemable due to some active mutation in progress. In other words, the right to claim the asset is locked. On the other hand, mutating=false means the asset can be claimed if the preimage to the hashlock is sent to the contract.
- replace_hash_lock: The swap hashlock is the hashlock used to claim assets. If the preimage of the hashlock is sent to claim(), then the receiver can claim the asset.
- replace_hash_lock: If replace_hash_lock != nil, that means Alice would like to transfer her option. replace_hash_lock stores the hashlock for the transfer.
- candidate_sender/candidate_receiver: denotes the new sender or receiver when transfer is finalized, i.e. Carol replaces Alice's role.

We take AB as an example to illustrate the states. The states on BA are symmetric which can be inferred from the context.

- Ready2Claim(H(A1), AB) denotes state (sender = Alice, receiver = Bob, mutating = false, swap_hash_lock = H(A1)), meaning if A1 is sent before the contract expires, Bob can claim Alice's principal. We will use Ready2Claim(H(A1)) for simplicity when which contract we are referring to is obvious from the context.
- MutateLockContestable(Carol, H(C1), H(C2), AB) denotes the state (sender = Alice, receiver = Bob, mutating = true, replace_hash_lock = H(C1), swap_hash_lock = H(C2), candidate_receiver = Carol), which means Alice tentatively locks the asset and transfers her role to Carol. In this state Bob is able to call contest(). We will use MutateLockContestable(H(C1), H(C2)) for simplicity when other parts we are referring to are obvious from the context. MutateLockContestable is just a preparation for Carol to replace Alice's role. The replaceLeader() cannot be called until the state defined below.
- MutateLockNonContestable(Carol, H(C1), H(C2), AB). In this state Bob cannot call contest() because his contest window has passed or he approved the mutation. At this state, Carol can release her secret C1 to replace Alice. The state when Carol can replace Alice is denoted as MutateLockNonContestable(Carol, H(C1), H(C2), AB). MutateLockNonContestable(H(C1), H(C2)) is the short name.
Figure 3: State transition until Carol replaces Alice on AB contract

- **Ready2Claim(H(C2), CB)**: After `MutateLockNonContestable(Carol, H(C1), H(C2), AB)` is reached, Carol can call `replace()` to replace Alice’s role by sending C1. After the replacement, the state becomes `(sender = Carol, receiver = Bob, mutation_lock = false, swap_hash_lock = H(C2))`, denoted as `Ready2Claim(H(C2), CB)`. `Ready2Claim(H(C2))` is the short name.

- **Claimed(H(S), AB)**: The asset on AB contract is claimed by a secret S which is the preimage to H(S) and Bob gets the asset. In this context, S can be A1 or C2. `Claimed(H(S))` is the short name.

The state transition is depicted in Fig 3.

There are two stages of changes of state on the AB and BA contracts.

1. **First stage.** Alice and Bob aim to change the state of both contracts from `Ready2Claim(H(A1))` to a mutation state `MutateLockNonContestable(H(C1), H(C2))`, during which state `MutateLockContestable(H(C1), H(C2))` may be reached temporarily, corresponding to Mutate Lock Phase and Consistency Phase.

2. **Second stage.** Starting from `MutateLockNonContestable(H(C1), H(C2))`, Bob and Carol are involved in an atomic change from `MutateLockNonContestable(H(C1), H(C2), Carol)` to `Ready2Claim(H(C2))` by Carol releasing a secret C1. It is similar to a standard atomic swap where Carol is the leader and Bob is the follower, corresponding to the Replace/Revert Phase.

**Theorem 17.** Conforming Alice will never end up in UNDERWATER.

**Proof.** If Carol never creates CA or creates CA with different conditions than what Carol and Alice agreed upon, then Alice does nothing and so she doesn’t lose her option.

Otherwise, consider the case when Carol creates CA with the conditions Alice and Carol had agreed upon.

If Alice is conforming, by Lemma 15 then both AB and BA are mutate locked with \((H(C_1), H(C_2))\) by `startLeader + 2\Delta`. This means AB and BA both have reached the state `MutateLockContestable(H(C_1), H(C_2))` by `startLeader + 2\Delta`. Since Alice is conforming, by `startLeader + 4\Delta`, the state will transition to `MutateLockNonContestable(H(C_1), H(C_2))` since Bob cannot forge Alice’s signature and contest.

After `MutateLockNonContestable(H(C_1), H(C_2))`, if Carol releases C1, Alice observes it in the Replace/Revert phase, and can get Carol’s principal as shown in Theorem 5. If Carol does not release C1, then eventually, after \(t_{AB} + 6\Delta\) and \(t_{BA} + 6\Delta\) respectively, the states on both contracts can be reverted to `Ready2Claim(H(A1))` which means Alice still owns the option and it is unlocked. If Alice finally owns the option, then she does not end up in UNDERWATER by the guarantee of atomic swap. If Alice loses her option (maybe only lose a role on one contract), she gets Carol’s principal, with/without losing her principal escrowed in her old option, which is acceptable for her.

\[\square\]

**Theorem 18.** Conforming Carol will never end up in UNDERWATER.

**Proof.** After Carol escrows her principal to Alice on CA and sends her hashlocks \(H(C_1), H(C_2)\) to Alice, Carol does not do anything in the first stage but observe. She only joins in the protocol after she observes that both contracts are in `MutateLockNonContestable(H(C_1), H(C_2))` state. Otherwise, she is silent and she will not end up in UNDERWATER since her principal will be refunded eventually. After she observes `MutateLockNonContestable(H(C_1), H(C_2))` state on both contracts, she releases C1, moving `MutateLockNonContestable(H(C_1), H(C_2))` state to `Ready2Claim(H(C_2))`. Then she owns the option provided by Bob and takes over Alice’s option. In that case, she may lose her principal to Alice due to the release of C1. Say, in the worst, she loses her principal to Alice.

Then, if Carol decides to exercise the option, she gets Bob’s principal. That means, her principal is exchanged with Bob’s. If she does not exercise the option and let it expire, she can get Alice’s principal. That means, her principal is exchanged with Alice’s. In any case, Carol never ends up in UNDERWATER.

\[\square\]

To prove that conforming Bob does not end up in UNDERWATER, we need to to prove the consistency of states on two contracts. Specifically,

- In the first stage, if Alice calls `mutateLockLeader()` in any contract, either both contracts eventually gets `MutateLockNonContestable(H(C_1), H(C_2))` or reverted back to `Ready2Claim(H(A_1))`.

- In the second stage, if Carol releases C1, both contracts are eventually at state `Ready2Claim(H(C_2))`. If Carol does not releases C1, both contracts are eventually are in state `Ready2Claim(H(A_1))`.

Suppose, without loss of generality, that AB contract is the first to update its state from `Ready2Claim(H(A_1))` to a new state by Alice calling `mutateLockLeader()` at some time t, assuming Bob is conforming.

**Theorem 19.** If BA is not claimed by \(t + \Delta\), then BA will be mutate locked by \(t + \Delta\).
Proof. Since Bob is conforming, when Alice calls `mutateLockLeader()` on the AB contract, Bob can send `mutateLockLeader()` transaction on the BA within $\Delta$. Recall that the start time that `mutateLockLeader()` is called is denoted as $t_{AB}$ and $t_{BA}$ respectively. Then in BA `mutateLockLeader()` is called on BA by $t_{AB} + \Delta$.

If Bob’s `mutateLockLeader()` is included on BA, then $t_{BA} \leq t_{AB} + \Delta$. If Bob’s `mutateLockLeader()` on BA is not included, then it must be Alice was able to call it before he was. In either case, $t_{BA} \leq t_{AB} + \Delta$.

The case that by $t + \Delta$, the other contract’s state changes to $\text{Claimed}(H(A_1))$ will be analyzed in Theorem 22.

Theorem 20. Suppose two contracts both eventually reach $\text{MutateLockNonContestable}(H(C_1), H(C_2))$. Then, either both contracts are eventually $\text{Ready2Claim}(H(A_1))$ or $\text{Ready2Claim}(H(C_2))$.

Proof. Since we know the start time of mutation between two contracts does not stagger beyond $\Delta$. Denote the start time on two contracts as $t_{first}$ and $t_{second}$ respectively where $t_{first} \leq t_{second}$. Since Bob is conforming, $t_{second} - t_{first} \leq \Delta$. We denote the timeout for Carol to release $C_1$ on AB and BA contracts as $t_1$,$t_2$ respectively, and the timeout for Bob to release $C_1$ as $t_3$,$t_4$. Without loss of generality, assume $t_1 = t_{first} + 4\Delta$ and $t_2 = t_{second} + 4\Delta$. $t_3 = t_{first} + 6\Delta$ and $t_4 = t_{second} + 6\Delta$. We see that the latest timeout for Carol to release $C_1$ is $t_2$. The earliest timeout for Bob to send $C_1$ satisfies $t_3 \geq t_2 + \Delta$ since we have $t_3 = t_{first} + 6\Delta \geq t_{second} + 5\Delta = t_2 + \Delta$.

That means, if Carol releases $C_1$, then it is eventually sent to both contracts and the state is $\text{Ready2Claim}(H(C_2))$. If Carol does not release $C_1$, then both contracts are reverted to $\text{Ready2Claim}(H(A_1))$ after timeout $t_3$ and $t_4$, respectively. □

The atomic changes between AB and BA are a bit more complicated, since starting from $\text{Ready2Claim}(H(A_1))$, there are multiple possible state changes available on both contracts: $\text{Claimed}(H(A_1))$, $\text{MutateLockContestable}(H(C_1), H(C_2))$, and $\text{MutateLockNonContestable}(H(C_1), H(C_2))$. The states $\text{Claimed}(H(A_1))$ and $\text{MutateLockNonContestable}(H(C_1), H(C_2))$ are not contestable. Assuming Bob is conforming and relay whatever he sees from one contract to another contract.

Lemma 21. If a contract is transferred to $\text{MutateLockContestable}(H(C_1), H(C_2))$ at $t$, then within $t + 2\Delta$, it learns the other contract’s update and then agree on either reverting back if there is a conflict or agree on the same mutation if no conflict.

Proof. If a contract, say AB, is transferred to $\text{MutateLockContestable}(H(C_1), H(C_2))$ at time $t$, since the compliant Bob relays it, whatever happens before $t - \Delta$ to AB contract, that means, at time $t - \Delta$, the other chain is in $\text{Ready2Claim}(H(A_1))$. If there is any change happening on BA contract, it would happen between $t - \Delta$ to $t + \Delta$. Then, by $t + 2\Delta$, AB contract will learn what happens on BA contract between $t - \Delta$ to $t + \Delta$ by Bob. If AB contract does not receive any transaction from Bob, then no conflicting changes happened on BA from $t - \Delta$ to $t + \Delta$. Bob can update BA contract to $\text{MutateLockContestable}(H(C_1), H(C_2))$ by relaying Alice’s mutation to BA contract, and AB contract can be updated to $\text{MutateLockNonContestable}(H(C_1), H(C_2))$ after $t + 2\Delta$. If there is any state change between $t - \Delta$ to $t + \Delta$ on BA contract, then by $t + 2\Delta$, AB contract would be informed, and its own state update is also sent to BA contract. Then the conflicting state update would revert back their state change to $\text{Ready2Claim}(H(A_1))$ when it receives the conflicting change. □

Theorem 22. Conforming Bob never ends up in $\text{UNDERWATER}$.

Proof. Starting from $\text{Ready2Claim}(H(A_1))$,

1. If a contract is transferred to $\text{MutateLockContestable}(H(C_1), H(C_2))$ at $t$, and the other contract is not updated to $\text{Claimed}(H(A_1))$, then by Lemma 21, we know that either both contracts eventually reach $\text{MutateLockNonContestable}(H(C_1), H(C_2))$ or are reverted back to $\text{Ready2Claim}(H(A_1))$.

2. If a contract BA is transferred to $\text{Claimed}(H(A_1))$ state at $t$, then, if the other contract makes temporary $\text{MutateLockContestable}(H(C_1), H(C_2))$, it will be revert back to $\text{Ready2Claim}(H(A_1))$ by $t + \Delta$, and then transfer to $\text{Claimed}(H(A_1))$. Bob does not end up with $\text{UNDERWATER}$.

3. If a contract is transferred directly to $\text{MutateLockNonContestable}(H(C_1), H(C_2))$ at $t$, then the other contract must be $\text{MutateLockContestable}(H(C_1), H(C_2))$ at $t' \in [t - \Delta, t]$, and then after $t + 2\Delta$ or sooner, the other contract will become $\text{MutateLockNonContestable}(H(C_1), H(C_2))$.

When $\text{MutateLockNonContestable}(H(C_1), H(C_2))$ is reached on both contracts, by Theorem 20, Bob will either be involved in the swap with Alice by Alice’s secret $A_1$ or involved in the swap with Carol by Carol’s secret $C_2$. By the guarantee of atomic swap, Bob will not end up with $\text{UNDERWATER}$.

□

The details for showing Protocol 4.3 satisfies $\text{No UNDERWATER}$ are omitted since they mirror the previous arguments shown for 4.2.