Transformable Discreet Log Contracts

by

Shwetark Patel

S.B., Computer Science and Engineering and Mathematical Economics, Massachusetts Institute of Technology (2021)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2022

Author . . . .
Department of Electrical Engineering and Computer Science
May 13, 2022

Certified by . . . .
Neha Narula
Director, Digital Currency Initiative
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Transformable Discreet Log Contracts

by

Shwetark Patel

Submitted to the Department of Electrical Engineering and Computer Science
on May 13, 2022, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Since Bitcoin and the Unspent Transaction Output (UTXO) model were introduced by Satoshi Nakamoto over a decade ago, there have been many important issues identified with the UTXO model; the most important being that it is hard to extend the model to accommodate more complex use cases, such as those related to decentralized finance. Currently, Ethereum has many decentralized exchanges which allow users to seamlessly make trades. Performing a trade on chain on Bitcoin is quite difficult; currently, the most elegant way is to set up a Discreet Log Contract (DLC) between you and your counter-party. However, this currently have many downsides; for example they are not transferable (i.e. once Alice and Bob sign up for the DLC, they are stuck in the DLC until settlement or they both interactively agree to leave). We fix this by introducing the Transformable Discreet Log Contract (TDLC), which allows a third party, Carol, to swap in for either Alice or Bob midway through the contract with reduced interaction and the Truly Transformable Discreet Log Contract (TTDLC), which allows multiple parties to seamlessly trade the contract around between them. With both the TDLC and the TTDLC, the party swapping into the contract only has to interact with the single party swapping out. The end goal for the work presented in this thesis is to help improve the usability of Bitcoin for advanced use cases such as those relevant to decentralized finance.

Thesis Supervisor: Neha Narula
Title: Director, Digital Currency Initiative

Acknowledgments

Thanks to Dr. Neha Narula and Tadge Dryja for their continued help and support with this project. I couldn't have done it without them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

First introduced in 2008, Bitcoin was the world's first decentralized cryptocurrency. The original idea of cryptocurrencies was to allow payments to be processed in a decentralized manner rather than relying on a central entity, though it has evolved to much more than this in recent times [6].

However, at the moment, Bitcoin is victim to many weaknesses. On Ethereum and similar chains, decentralized exchanges exist, which allow users to perform trades and swaps with ease. Due to the way Bitcoin is structured, the solutions existing on Ethereum are not viable on Bitcoin. There are some existing ideas on Bitcoin, which will be mentioned in the related work, that address some of the issues, but are much weaker than the solutions on Ethereum due to the existence of smart contracts on Ethereum. We aim to introduce additional ideas to improve some of the existing solutions on Bitcoin surrounding making swaps and trades.

1.1 Intro to the UTXO Model

The Unspent Transaction Output (UTXO) model is the precursor to all of the other cryptocurrency models that exist today. Nakomoto first introduced the idea in 2008 in the Bitcoin Whitepaper. In order to understand the UTXO model, we must first understand what transactions in Bitcoin are composed of. Each transaction is composed of a certain number of inputs and a certain number of outputs, where the sum of the input amounts needs to be at least as much as the sum of the output amounts (so you can't create Bitcoin out of thin air).

Each input is composed of the following components:

1. Transaction ID that references an unspent output and index
2. Script Signature – This is a way to prove that you can satisfy the conditions to spend the unspent output that is being spent.

Each output is composed of the following components:

1. Amount
2. Output index
3. Script PubKey – This indicates how the output can be spent by a later input.

Once the transaction is created, the Transaction ID (TXID) is generated by double hashing (using SHA256) the transaction contents (minus witnesses, which are a special case which will be discussed later). This TXID will be used by future transactions to reference a created output; for example, say my transaction has two outputs and a TXID X. Then, future transactions can use X as well as the relevant output index (either 0 or 1, as there are 2 outputs) in order to spend the unspent output [6].

1.1.1 Segregated Witness V0

A small detail that must be noted is that, in pre-segwit Bitcoin, signatures that prove you are able to spend a specific unspent output are part of the transaction data that is double hashed in order to create the TXID. This is annoying primarily because you must commit to a transaction by signing it before knowing how you might want to spend the outputs, which makes many use cases impossible. In more detail, there are use cases where we want to spend the output of an unsigned transaction. For example, if we want to be able to revert a transaction at some point if certain conditions are met, we can take the output of the unsigned transaction, create a transaction spending it back to the original owners, get their signatures on the transaction, and then have it put on chain later if necessary. This guarantees that refunding will be possible in the event that we commit to the unsigned transaction. Such use cases were originally impossible on Bitcoin, but developers soon came up with a solution to this. This solution is known as Segregated Witness, and it allows one to put signatures in the witness data, which is not included in data which is hashed to produce the TXID. This opens up many opportunities – you can, for example, create a transaction's format without signing it and still know its TXID so that you can spend its output with other transactions [5].

This will lay the groundwork that we need to understand Discreet Log Contracts.

1.1.2 Other Terminology

Before we dive into DLCs, we need to go over some other basic terminology related to Bitcoin.

1. M-of-N Multi-signature Outputs – This is an output whose Script PubKey indicates that it can only be spent if we have M signatures out of the possible set of N determined keys. For example, a 2-of-3 Multi-signature Output would involve 3 predetermined keys and the output can only be spent if at least 2 of those keys provide a signature in the spending input's scriptSig.

2. Pay to Public Key Hash (PKH) vs Pay to Script Hash (PSH) – Pay to public key hash means creating a UTXO which only requires a signature from a specific public/private key pair. Pay to Script Hash means creating a UTXO which can be spent by anyone who provides data (most often including signatures) that meets the specifications of the script.

3. Block height – This refers to the current block that the Bitcoin blockchain is

on.

4. Check Locktime Verify (CLTV) – This is an opcode that can be included in a script to require that the current block height that the transaction is being executed on is at least some height.

5. Check Sequence Verify (CSV) – While CLTV is an op code that acts an absolute time lock (i.e. it checks that an output is being spent on at least some height), CSV is a relative time lock. It checks that the height difference between when the output is being spent and when the output is created is at least some fixed value.

6. SIGHASH flags – Usually, when we sign off on a transaction, we sign off on all the contents. However, certain SIGHASH flags allow us to sign off on only specific parts of the transaction contents rather than all of it.

7. SIGHASH ALL – This is the standard SIGHASH type, where we essentially just sign on the entire transaction contents.

8. SIGHASH SINGLE | ANYONECANPAY – This is a specific type of SIGHASH flag that allows us to create a signature that only signs off on the first input and the first output. This will allow for arbitrary other inputs and outputs to be added later. This sighash type is generally used in the event that someone only cares about what the first input and output of a transaction is but doesn't care what happens with the rest of the transaction. As an example, say that there's a party of friends that is attempting to settle a bet. The first friend, Bob, may only care that he gets sent a certain amount of coin using a specific UTXO as input, and can use SIGHASH SINGLE | ANYONECANPAY to represent this [6].

9. SIGHASH ANYPREVOUT | ANYONECANPAY | SINGLE – This is a specific type of SIGHASH flag that allows us to create a signature that only signs off on the spending condition of the first input and the entire first output. This also will allow for arbitrary other inputs and outputs to be added later [9].

1.2 Intro to the Account Model

In the Ethereum account model, there are two different types of addresses: externally owned accounts and contract accounts. Externally owned accounts do not maintain any state storage, but do have an account balance associated with them. This is somewhat distinct from the UTXO model, in which UTXOs can be spent as long as you can prove you can spend them. In the account model, however, the private key just owns the balance in the associated externally owned account.

Contract accounts, on the other hand, have associated storage, which is extremely powerful and is the reason that making trades and swaps is so much easier in the account based model. This storage is non-volatile; it is persistent across transactions. Contracts are essentially just some bytecode that is deployed to the blockchain, and this bytecode is executed whenever the contract is called. It is important to note that contracts themselves cannot send transactions; they can only be called as a result of a transaction sent by an externally owned account. Because these contracts have associated persistent storage, it is much easier to store data necessary to operate a decentralized exchange, such as amount of liquidity on both sides for example.

On account based block-chains, there are two main types of decentralized exchanges. One are automated market makers, which essentially have liquidity providers provide liquidity on both sides. Traders are then able to swap coin from one side to the other side for some small fee by trading against liquidity providers.

The other type of exchange is order book based exchanges, which operate similar to existing centralized exchanges except they process the order book in a decentralized manner [8].

1.3 Automated Market Makers

Automated Market Makers (AMMs) works in a fairly straightforward fashion. AMMs each have a bunch of pools, each of which allows trades between two tokens. For a specific pool, the AMM holds the amount of the first token and the amount of the second token in the pool.

There are two main types of operations on a pool: trades and liquidity additions/deletions. With trades, the user requests to trade a certain amount of token A for some amount of token B. The AMM uses the amount of tokens on both sides of the pool and the amount of token A that the user is putting in to calculate how much of token B the user should get out. The pool then sends this amount of token B to the user and receives the promised amount token A from the user. The AMM usually levies a transaction fee (generally a percentage of the amount of token A the user is putting into the pool) for each trade.

The formula for most standard AMMs is very simple. Let's say that A_0 and B_0 are the amount of reserves in the AMM of the first and second token respectively before any trades happen. Similarly, A_1 and B_1 are the amount of reserves in the AMM of the first and second token after a trade happens. For the following analysis, **we will assume the transaction fee is 0**. We must have:

$$A_0 \cdot B_0 = A_1 \cdot B_1 \tag{1.1}$$

Let's work through an example. Let's say that $A_0 = B_0 = 2$ initially. If a user trades 2 of the first token for the second token, then the amount they will get back is fairly easy to calculate:

First, we see that $A_1 = A_0 + 2 = 2 + 2 = 4$. Then, from 1.1, we find that:

$A_0 \cdot B_0 = A_1 \cdot B_1 \Rightarrow 2 \cdot 2 = 4 \cdot B_1 \Rightarrow B_1 = 1$

This means that the user receives $B_0 - B_1 = 2 - 1 = 1$ of the second token back from the AMM. This also shows us why having liquidity in the pool is important. For example, if initially we had $A_0 = B_0 = 10$ and again the user traded 2 of the first token for the second token, we can redo the analysis:

We see that $A_1 = A_0 + 2 = 10 + 2 = 12$. Then, from 1.1, we find that:

$A_0 \cdot B_0 = A_1 \cdot B_1 \Rightarrow 10 \cdot 10 = 12 \cdot B_1 \Rightarrow B_1 = \frac{25}{3}$

This means that the user receives $B_0 - B_1 = 10 - \frac{25}{3} = \frac{5}{3}$ of the second token. With higher liquidity, users can make larger trades while maximizing the amount that they receive out.

With liquidity additions, a user can add in a certain amount of token to both sides of the pool in order to mint liquidity provider (LP) tokens, which entitle the user to receive a share of the entire pool (including transaction fees collected). Let C_0 and D_0 be the amount of the first and second token the user adds into the pool. Furthermore, let S be the number of currently existing LP tokens. It should be the case that $\frac{C_0}{A_0} = \frac{D_0}{B_0}$. Furthermore, the number of LP tokens that the user will receive back is $\frac{C_0}{A_0} \cdot S$. As an example, say that $A_0 = B_0 = 2$ and $S = 6$. Then, the user could potentially send $C_0 = D_0 = 1$ tokens to the AMM, and would receive back $\frac{C_0}{A_0} \cdot S = \frac{1}{2} \cdot 6 = 3$ additional newly minted LP tokens.

With liquidity deletions, the user withdraws their liquidity from the pool, and receives some amount of token A and some amount of token B as a result. If the user sends X LP tokens to the pool, they will receive back $\frac{X}{S} \cdot A_0$ of the first token and $\frac{X}{S} \cdot B_0$ of the second token. If we have $A_0 = B_0 = 3$, $S = 9$, and $X = 3$, then the user would receive back $\frac{3}{9} \cdot 3 = 1$ of the first token and $\frac{3}{9} \cdot 3 = 1$ of the second token.

In order to execute both trades and liquidity additions/deletions, the funds that the user needs for the transactions need to be sent to the smart contract at some point. For example, if the user is making a trade, the smart contract would give the user some token in exchange for some token of another type. The smart contract would assume ownership of the user's input funds. Because smart contract code is public and verifiable on the blockchain, anyone can verify that the smart contracts will execute these operations fairly and reliably [1].

The account model allows for much more flexibility than the UTXO model, especially due to the ability to deploy smart contracts which can execute code in a publicly known and verifiable manner. However, there are ways to get around some of the issues with the UTXO model.

1.4 How can we get around what the UTXO Model Lacks?

In the UTXO model, the reason that it's so hard to create a decentralized exchange is because of the lack of persistent, mutable storage capabilities on Bitcoin. Furthermore, there are no other types of programmable tokens on Bitcoin like there are on Ethereum, so there is nothing to exchange. Because Bitcoin requires a spender to prove that they can spend the inputs, but does not indicate how the inputs can be spent, it is impossible to enforce the generalized state changes that these decentralized exchanges require without building on top of Bitcoin. Without building on top of bitcoin, we can't dynamically keep track of data needed to operate the exchange, such as the order book or liquidity on both sides [6]. There are ways to build on top of Bitcoin to introduce these capabilities, but they either lead to lack of decentralization, are inefficient, or require specific restricting assumptions. The Extended UTXO model that Cardano uses, for example, succeeds in extending Bitcoin script and allowing more complex validation scripts to be safely used on Cardano. However, it suffers from the same issue as the UTXO model in that it only allows users to spend a specific UTXO once per block, which is particularly disadvantageous compared to Ethereum. If multiple users want to make a DEX trade on Ethereum, that would be fine since they can all interact with the smart contract, but on Cardano, all state changes besides the first one cannot go through since they would rely on the first state change [2].

However, even on Bitcoin, creating some type of contract whose terms are predetermined that simulates a trade would still be possible, because the parties can essentially pre-sign transactions that settle the contract rather than doing so dynamically. This is the basic idea behind discreet log contracts.

1.5 Intro to Futures Contract Trading

On centralized exchanges, the idea of a futures contract is fairly straightforward. Essentially, you are attempting to predict the price of an underlying asset at some point in the future.

There are two sides to a futures contract: long and short. If you buy (i.e go "long") on a futures contract, then, assuming you hold until expiration, your profit will be the price of the contract at expiration minus the price you bought the contract at. On the other hand, if you sell (i.e. go "short") and hold until expiration, your profit will be the price of the contract at expiration minus the price you bought at.

There are two other extremely important properties of futures contracts:

1. You can trade futures contracts around with other people (i.e you do not necessarily need to hold until expiration).

2. Physical delivery of the underlying item is usually enforced by the contract

3. You don't actually need to put in the maximum possible required amount of capital in order to enter into a futures position; you can usually trade on leverage. For example, say that you want to go long on 5 futures contracts at 50 dollars each. You don't need to actually put down $50 \cdot 5 = 250$ dollars; rather, you just put down a percentage and if the market moves against you, your broker will likely liquidate you. This makes futures contracts less capital constraining. Neither the DLC, Transformable Discreet Log Contract (TDLC), nor the Truly Transformable Discreet Log Contract (TTDLC) will support this use case; all of them will require both parties to commit to the maximum required amount of capital beforehand. However, because the payout structure does not need to be 1:1, the participants can get amplified sensitivity to volatility by committing different amounts of money. It is important to note that the participants can never lose more money than the amount committed. [4].

1.6 Intro to DLCs

The Discreet Log Contract (DLC) was invented by Dryja as a means to allow trading on Bitcoin. Essentially, two parties, after performing some pre-computation, can enter into a contract that can be settled at some point in the future based on the outcome of some event. The payout structure is potentially different for every single outcome.

In order to achieve this, we need to rely on a trusted oracle model (Note: This is different from the standard Oracle Model present in cryptography), where we trust the oracle to produce a different signature for every outcome of the event (this is how we know which payout structure is valid on chain). However, the oracle itself does not even know about the existence of the contracts. The oracle could be offering a service with a formatted price feed, for example, that many people might read and use. The only step where the oracle is involved is publishing the right signature for the outcome that the participants are betting on [3].

Let's go through an example to explain how the discreet log contract works: Let's say that Alice and Bob are betting on the outcome of a game which has two outcomes: win or lose. Then, before the discreet log contract can actually be put on chain, we need to perform some precomputation:

1.6.1 Pre-computation

The pre-computation involves several steps: First, we must create the layout of some transactions. It is important to note that we do not actually put any transactions on chain while creating transaction layouts. Furthermore, we use SegWit for all transactions here, so that we can determine the TXID of a transaction without actually putting it on chain.

We create transaction layouts in the following order:

1. The funding transaction. Essentially, the inputs to this are just a UTXO from Alice and a UTXO from Bob of the same amount (though practically, Alice

and Bob will just use UTXOs of arbitrary amounts and send some money back to their PKH as change). The output is a 2-of-2 multi-signature output between Alice and Bob. Note that no signatures have been created or broadcasted yet. The funding transaction would look as shown in 1-1.

2. The contract execution transactions (CET). Because we know the TXID of the funding transaction, we can use it as an input to our CETs. Both Alice and Bob create one CET for each possible outcome of the contract which represents the payout structure. Say that Alice is creating a CET for the "WIN" outcome. She creates the layout for a transaction which has the funding transaction UTXO as an input and pays to her own PKH as the first output (she is essentially retrieving the amount that she won from the contract by doing this) and a second P2SH output. Essentially, this script will allow Bob to spend the UTXO if he can prove that the oracle signed on the "WIN" outcome. If, after some time, Bob has not tried to sped the UTXO (which likely implies that he does not have the correct oracle signature), the script allows Alice to spend the UTXO. The CETs that Bob creates are symmetric (i.e. essentially Alice and Bob swap roles). The CET associated with the "WIN" outcome that Alice would sign look as shown in 1-2.

The next step is signing. Alice will take the CETs that she has created, sign them, and send them to Bob. Bob will do the same.

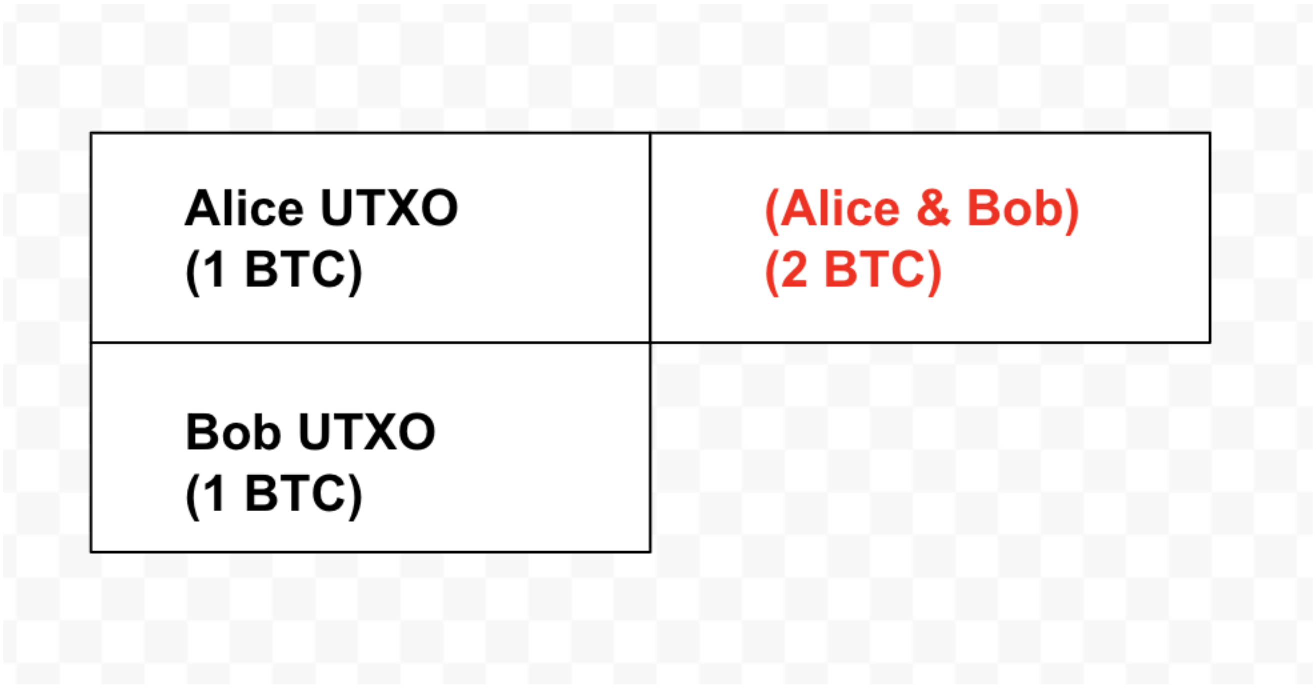

Figure 1-1: Funding transaction

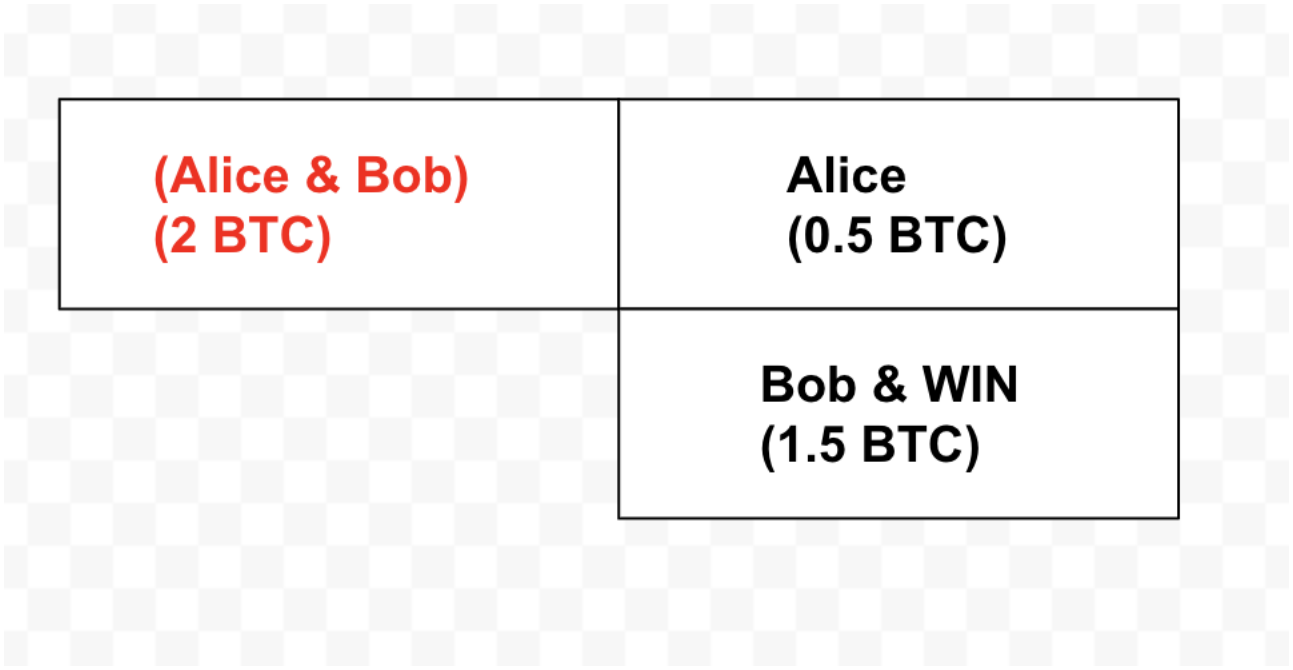

Figure 1-2: Contract Execution Transaction

1.6.2 Execution

Once all the CETs have been signed, if both parties are satisfied, they can sign the funding transaction (in any order) and put it on chain. It is important that these CETs are signed before the funding transaction is signed; if the funding transaction is signed first, then it can be put on chain without any guarantees that the contract can be settled through a CET. This means that one party can hold the other party's funds hostage. Now, after the settlement time has been reached, the oracle will publish the correct signature. Either party can find the CET corresponding to the outcome, which the counter-party has already signed, sign, and put it on chain. After the CET is put on chain, the party who put the CET on chain must still spend the PSH output using the oracle signature. This enforces that the party put the correct CET on chain; otherwise, they will not be able to spend the PSH output.

Note that cooperation between the two parties is still possible before the CET is put on chain. Furthermore, in the event of oracle failure, there is a timeout transaction created at the beginning which sends both parties their funds back. Also, if the parties don't want to go through the hassle of finding and putting a CET on chain, they can mutually agree on and sign a transaction that sends them both the correct amount of money [3].

1.7 Disadvantages of DLCs

The core idea of DLCs is to be able to model futures contracts, except in a decentralized fashion. However, as we've referenced earlier, there are a few major issues with DLCs:

1. They are non-transferable. This means that, once you enter into a DLC, you cannot exit it until the time you agreed to exit without cooperation.

2. They require both sides to commit the maximum amount of capital necessary before entering into the contract.

None of the solutions we propose solve problem 2. However, the Transformable Discreet Log Contract partially solves problem 1 while the Truly Transformable Discreet Log Contract fully solves it.

1.8 Related Work

1.8.1 The Lightning Network: Payment Channels

Introduction

One major problem that faces the blockchain today is throughput. With so many incoming transactions, each of these transactions must be put into a block in an efficient fashion. However, traditional methods to do this, such as increasing block size, lead to decreased decentralization.

One way to tackle this problem is to fight it at the protocol level. Imagine that persons A and B want to pay each other a multitude of times. Normally, they would have to create a transaction on chain whenever one wants to pay the other.

However, with the lightning network, we can avoid putting all these transactions on chain. The two will simply create a payment channel, continuously pay each other through the channel, and then put the final result on chain [7].

We discuss the lightning network because it connects to the idea of using Bitcoin script and segregated witness in a clever manner, which enables us to support more operations than originally thought on Bitcoin. Let's explore how this works in more detail.

It's important to note that the lightning network uses more than just payment channels; it also does routing over a network. It makes a network of these channels and routes payments through these channels. However, we will only focus on an individual payment channel.

Payment Channels

Initially, one party funds the payment channel. Say that Alice initially owes Bob 0.5 BTC. This funding transaction (let's call it **Transaction A**) would look like this:

Transaction A

Inputs:

1. Alice UTXO (1 BTC)

Outputs:

1. Alice and Bob Multisig (1 BTC)

Before this is signed by either party, however, an "initial commitment" transaction must be created. Let's call this initial commitment transaction **Transaction B**. This commitment transaction would look different for both parties. The commitment transaction given to Bob by Alice would look like the following:

Inputs:

1. Alice and Bob Multisig (1 BTC)

Outputs:

1. Alice PKH (0.5 BTC)
2. (Bob PKH AND OP CSV) OR (Alice PKH AND Revocation Key) (0.5 BTC)

Alice signs this transaction and gives it to Bob, who can then sign and broadcast whenever he wishes.

The idea behind this revocation key is straightforward: Bob holds a key-pair and in order to nullify the above transaction, he can simply reveal the private key.

If Bob proceeds to sign and send transaction B on chain afterwards, Alice can spend the first output as well as the second output with the revocation key.

Similarly, the commitment transaction given to Alice by Bob looks like the following:

Inputs:

1. Alice and Bob Multisig (1 BTC)

Outputs:

1. Bob PKH (0.5 BTC)
2. (Alice PKH AND OP CSV) OR (Bob PKH AND Revocation Key) (0.5 BTC)

Bob signs this transaction and gives it to Alice, who can then sign and broadcast whenever she wishes.

Transaction A is then broadcasted on chain.

Now, Alice and Bob can start sending each other Bitcoin through the payment channel. If Alice wants to send Bob some amount of coin, then they would calculate their new balances. Say Alice has 0.4 Bitcoin and Bob has 0.6 has Bitcoin after the transfer. They would first both reveal their revocation keys for the previous commitment transaction (i.e. Transaction B in this case).

They would then create and sign a new pair of commitment transactions. Let's call these new commitment transactions **Transaction C**.

The new commitment transaction that Alice gives to Bob looks like the following:

Inputs:

1. Alice and Bob Multisig (1 BTC)

Outputs:

1. Alice PKH (0.4 BTC)

2. (Bob PKH AND OP CSV) OR (Alice PKH AND Revocation Key) (0.6 BTC)

Alice signs this transaction and gives it to Bob, who can broadcast it whenever he wants.

Similarly, the new commitment transaction that Bob gives to Alice looks like the following:

Inputs:

1. Alice and Bob Multisig (1 BTC)

Outputs:

1. Bob PKH (0.6 BTC)

2. (Alice PKH AND OP CSV) OR (Bob PKH AND Revocation Key) (0.4 BTC)

Bob signs this transaction and gives it to Alice, who can broadcast it whenever she wants.

Every time one party wants to send the other money, they repeat this process of broadcasting the revocation keys and then recreating the commitment transactions.

Any party can close the channel at any time, regardless of their balance, by broadcasting the most recent commitment transaction. For example, if either Alice or Bob wanted to close the channel after the latest commitment transaction, either of them can sign their version of Transaction C and broadcast it to the chain. If they broadcast an old commitment transaction, the other party can just take all their money within the CSV time. They can also cooperatively create a transaction that spends the final amounts to both their addresses; this saves time because the party broadcasting does not need to wait before receiving their funds.

1.8.2 Other related topics

There are other projects that aim to extend programmability on Bitcoin, some of which we have taken intuition from [2] [10] [12]. There are also some other solutions which aim to extend the world of decentralized finance on Bitcoin [11]. The most interesting and relevant of these is SNICKER, which aims to make privacy protecting coin-join transactions non-interactive [10].

Chapter 2

The Transformable Discreet Log Contract

2.1 Introducing the TDLC

One of the biggest disadvantages of DLCs is that, once you are in a contract, you can't get out until expiration, unless both parties agree to leave. This is because you shouldn't publish one of the CETs without knowing the oracle outcome (otherwise you stand to lose a lot of money if you're wrong when the oracle publishes the outcome!). This is very different from traditional finance – if you are in a futures contract, then you can sell that futures contract to someone else if you want, but only if it has not yet reached expiration time.

The TDLC allows the DLC to be partially transferable. In particular, we choose another party, Carol, whose public key is known beforehand. Carol will be able to swap into the contract mid-way through, with either Alice or Bob swapping out. Of course, both the party swapping in and the party swapping out must agree to this. Note that the party staying in the contract should not care about this swap at swap time, as the terms of the contract stay exactly the same. However, both Alice and Bob need to agree to potentially allow Carol to swap in for one of them in the pre-computation phase (before any transactions are put on chain). Note that if

Carol ends up swapping into the contract, no further swaps are possible without interaction between Carol and the party who stays in the contract.

Two other important properties that we maintain is that only the party swapping in and the party swapping out must be online when the swap happens. This is an important non-interactivity related property because the party who wants to substitute out of the contract and Carol don't want to rely on anyone else being online to make the substitution. Furthermore, Carol should not need to be online at the beginning while Alice and Bob are performing the pre-computation. Again, this is important because Alice and Bob don't want to rely on Carol being online to enter into a contract.

Also, there is a slight downside to our protocol. If Alice ends up swapping out while Carol ends up swapping in, then Carol must be the one to put the CET on chain. The one exception to this final property is, in the event that after some amount of time Carol does not put the CET on chain, Bob will be able to come back and claim all the funds (though he needs to be present to do so). The reason Carol is required to put the final CET on chain in this case is in order to help maintain the property that only Alice and Bob should need to be online during the pre-computation phase.

2.1.1 Pre-computation

The pre-computation step is very similar to regular Discreet Log contracts. Alice and Bob sign the same exact transactions between themselves 1-1 1-2. However, they also have to sign additional transactions that will allow Carol to swap in for either party later on. Right now, we will assume that there is only one Carol, and will generalize this to multiple Carols later.

First, Alice will create and sign a single transformation transaction (let's call it **Transformation Transaction 1**) using SIGHASH SINGLE | ANYONECANPAY. The transaction will look like the following:

Inputs:

1. Alice and Bob Multisig (This is spending the output of the funding transaction 1-1)

Outputs:

1. (Alice and Carol Multisig) OR (Alice and CLTV)

The purpose of this transaction is to allow transferability of the contract between Alice and Bob to one between Alice and Carol (though importantly note that an additional CLTV is part of the new script as well!). It's important to note that this is not what this transformation transaction will look like on chain; when Bob is swapping out and Carol is swapping in, they will add some inputs/outputs to make this transaction more complete.

Bob must create and sign a similar symmetric transformation transaction (let's call it **Transformation Transaction 2**) that will potentially allow Alice to swap out. This transformation transactions looks like the following:

Inputs:

1. Alice and Bob Multisig (This is spending the output of the funding transaction 1-1)

Outputs:

1. (Bob and Carol Multisig) OR (Bob and CLTV)

Furthermore, for each possible outcome, Alice will create and sign the following additional CET (let's call it **Additional CET 1**) in order to allow Carol to potentially close a DLC between Alice and Carol later on:

Inputs:

1. (Alice and Carol Multisig) OR (Alice and CLTV) (This is spending the first output of Transformation Transaction 1)

Outputs:

1. Alice
2. (Carol and Oracle Outcome) OR (Alice and CLTV)

Similarly, for each possible outcome, Bob will create and sign the following additional CET in order to allow Carol to potentially close a DLC between Bob and Carol later on:

Inputs:

1. (Bob and Carol Multisig) OR (Bob and CLTV) (This is spending the first output of Transformation Transaction 2)

Outputs:

1. Bob
2. (Carol and Oracle Outcome) OR (Bob and CLTV)

In less technical terms, our first input in Additional CET 1 is essentially just a DLC between Alice and Carol (note that it was originally Alice and Bob, and has now been modified) with an additional CLTV clause. The outputs are just a payout to Alice and a payout to Carol if she can provide the correct oracle signature (otherwise, Alice can come back and claim all the funds after some time delay!)

This essentially forces Carol to be the one to put the final CET on chain; otherwise, Alice will be able to come back and claim all the funds after some amount of time. Carol can always do this since she already has Alice's signature on the CET.

Furthermore, the CLTV should be flexible to the point where Carol has enough time to put the CET on chain after the transformation transaction has been put on chain.

All of the transactions that Bob signs are given to Alice and all of the transactions that Alice signs are given to Bob. Carol has no steps required from her in this pre-computation phase and is not necessarily online to receive any transactions or signatures. This allows us to maintain the property that Alice/Bob are the only ones who should be required to be online during precomputation.

Once all of this signing is complete, Alice and Bob can sign the funding transaction and put it on chain.

2.1.2 Execution

There are two cases with the execution; either a swap happens or it does not.

If a swap does not happen, then the contract just settles like a regular DLC (i.e. a CET is put on chain by one of the parties).

If a swap happens, then without loss of generality let's say Alice swaps out and Carol swaps in. Then, Alice gives Carol the following: Bob's signatures on the transformation transaction and the CETs. Carol knows that she must pay Alice some amount to swap into the transaction. They discuss and decide on an amount. Carol then creates the following modified Transformation Transaction:

Inputs:

1. Alice and Bob Multisig
2. Carol UTXO

Outputs:

1. (Bob and Carol Multisig) OR (Bob and CLTV)
2. Carol Change (She will likely send to her own PKH)

3. Alice PKH

Essentially, Carol adds inputs and outputs that include the payment from Carol to Alice. She then signs the transaction (with SIGHASH ALL) and sends the signature to Alice. If Alice would like to proceed, she can sign the transaction and put it on chain.

Finally, after settlement time, Carol must find the relevant CET for the oracle outcome and put it on chain, thus settling the contract with Bob. In the event that Carol puts the wrong CET or takes too long to put a CET, Bob can claim all of the funds.

It is important to note that cooperation between Bob and Carol is still possible at the end. If Carol doesn't want to go through the hassle of finding and putting a CET on chain, she can just contact Bob and they can mutually agree on and sign a transaction that sends them both the correct amount of money. This could potentially be advantageous to Bob as well since the transaction fee for the CET may have been higher initially.

2.2 Extension to Multiple Carols

This protocol can be extended with ease to multiple Carols as well. Alice and Bob must initially know the public keys of all the Carols that they are planning on potentially involving in the contract. All of the transformation transactions and additional CETs that were created for one Carol in the case with one Carol can now be created independently for each Carol. In the execution step, say that Bob wants to swap out (and he potentially asks multiple Carols to swap with him). When Bob and each of the Carols interact, although multiple Carols may sign and send the signatures to Bob, Bob can only put one on chain as there is only one funding transaction output. Similarly, the Carols are still able to move their input UTXOs if they believe that Bob is taking too long putting their transformation transaction on chain.

The downside, however, is that the extension to more Carols does require significantly more signatures to be created. Before, with just one Carol, the Big-O complexity of the number of signatures required was the same as regular DLCs. However, now the complexity is multiplied by the number of Carols involved, as we need separate signatures for each Carol. Furthermore, if multiple Carols try to swap in, then only 1 of them will be successful. Though this could potentially be a waste of time for Carols who failed to swap in, they did not make any on chain transactions and thus did not lose any money.

2.3 Example

We provide an example of what the on-chain transactions would look like in the event of a successful swap. First, assume that the funding transaction looks like the basic DLC funding transaction we've shown 1-1.

Say that Bob wants to swap out of the contract and Carol wants to swap in. Bob and Carol would first interact; Say that Bob and Carol agree that Carol will pay Bob 0.5 BTC in order to be able to swap into the contract. Bob would then put a transformation transaction on chain, which is shown in 2-1.

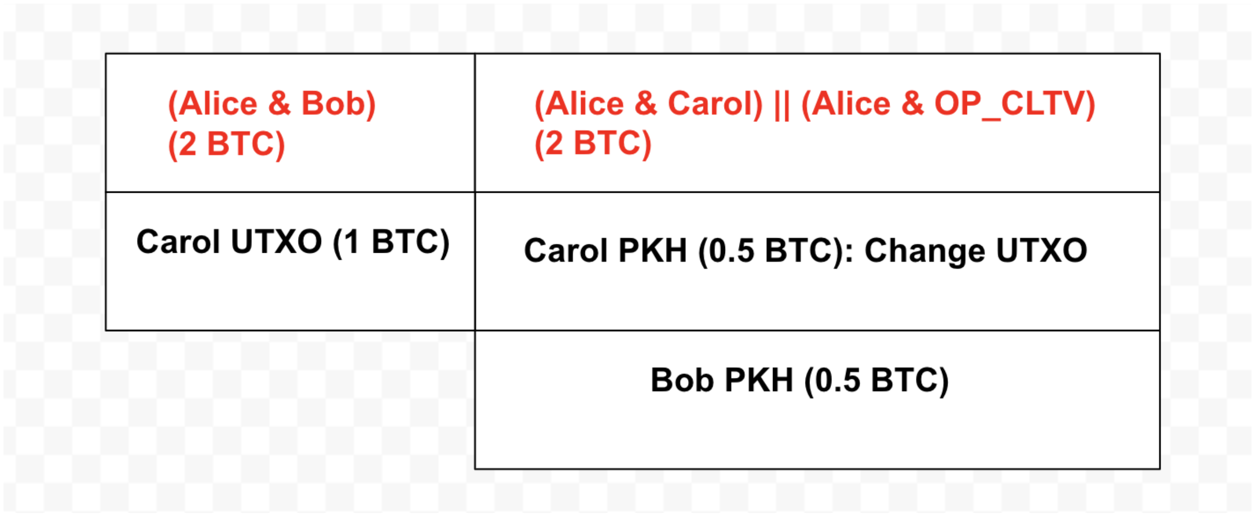
(Alice & Bob)
(2 BTC)
(Alice & Carol) || (Alice & OP_CLTV)
(2 BTC)
Carol UTXO (1 BTC)
Carol PKH (0.5 BTC): Change UTXO
Bob PKH (0.5 BTC)

Figure 2-1: TDLC Transformation Transaction

Finally, say that the oracle publishes a signature. Carol would then be the one to find the right CET and put it on chain, as shown in 2-2.

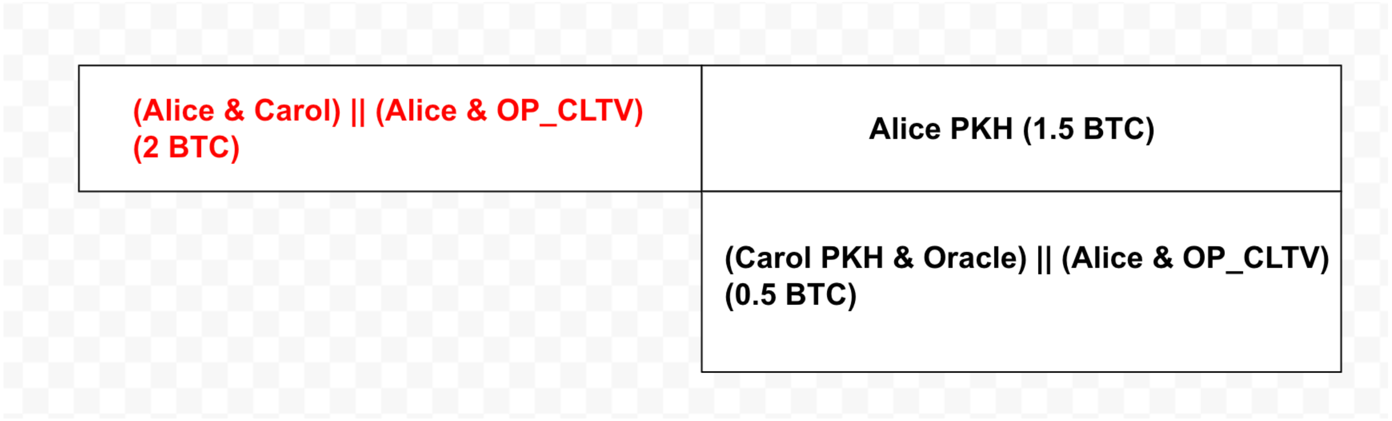
(Alice & Carol) || (Alice & OP_CLTV) (2 BTC)
Alice PKH (1.5 BTC)
(Carol PKH & Oracle) || (Alice & OP_CLTV) (0.5 BTC)

Figure 2-2: Carol's TDLC Contract Execution Transaction

Carol can simply spend the second output of this to her own PKH assuming she put the right CET on chain, as she has the oracle signature.

2.4 Security Considerations

2.4.1 State Machine Representation

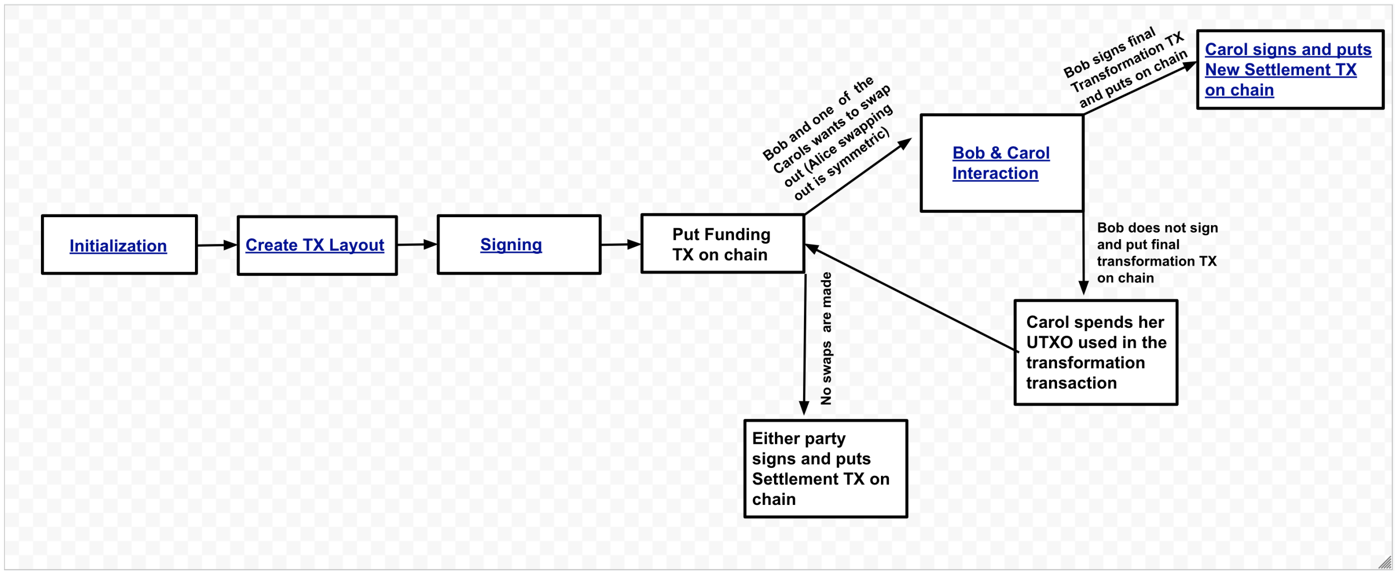
Initialization
Create TX Layout
Signing
Put Funding TX on chain
Bob and one of the Carols wants to swap out (Alice swapping out is symmetric)
Bob & Carol Interaction
Bob signs final Transformation TX and puts on chain
Carol signs and puts New Settlement TX on chain
Bob does not sign and put final transformation TX on chain
Carol spends her UTXO used in the transformation transaction
No swaps are made
Either party signs and puts Settlement TX on chain

Figure 2-3: TDLC State Diagram

This state machine representation, shown in 2-3, describes exactly what happens throughout the TDLC process. The first step is essentially initialization, where Alice and Bob meet and decide on all the potential Carols as well as the contracts that they'll be betting on. Then, the transactions necessary for the TDLC process are created. After this, the relevant transactions are signed and sent between Alice and Bob. Alice and Bob will then send these signed transactions to the Carols when they come online as necessary. The funding transaction is then put on chain. There are then two cases: either the transformation transaction is put on chain or not. In the latter case, the process proceeds like in a regular DLC. In the former case, Bob and Carol start interaction and reach agreement to swap Bob and Carol. Once Carol signs the transformation transaction and sends it to Bob, there are two possibilities: either Bob signs and puts it on chain or Carol gets tired of waiting for Bob to do this and decides to spend her input UTXO, thus negating the entire transaction. If the transformation transaction is put on chain, Bob is out and it is up to Carol to later put the relevant CET on chain. Alice can no longer put the CET on chain since she doesn't have Carol's signatures on the CETs.

Some of these state machine steps lead to interesting security considerations. For example:

1. Right after Carol sends her signature to Bob, Bob has some time to decide whether he really wants to sign and put it on chain. The caveat is that Carol can move her UTXO at any point in time, so Bob has to be careful with how long she takes. The interesting point is that there can be a race condition (called a "swaption") where Bob tries to put the signed transaction on chain at the same time that Carol tries to move her UTXO, which may lead to both parties bidding up the transaction fee [3].

2. If there are hundreds of Carols that are allowed to participate in the process, and they all sign transformation transactions and send them to Bob, Bob has quite a bit of leverage. He can wait longer to see if the odds are moving against him; he doesn't care if a few Carols drop out. In the worst case, this

can lead to many on chain transactions where Carols decide to move their input UTXOs because they see that Bob is waiting to see if the odds end up in his favor.

3. Setting CLTV times is an important question as well. In order to be perfectly safe, there needs to be enough time between the point at which the oracle releases the outcome and the CLTV time for the CETs (otherwise neither party can publish a CET without the other person racing to steal the funds). Furthermore, the CLTV on the transformation transaction output needs to be generous enough for Carol to be able to put a CET on chain afterwards.

Chapter 3

The Truly Transformable Discreet Log Contract

3.1 Introducing the TTDLC

The Truly Transformable DLC is a continuation of the idea described in the Transformable DLC. The assumptions of the TTDLC are the following:

1) The TTDLC assumes that there are N parties who could potentially be involved in the contract. Furthermore, we want to allow these parties to trade around the contract freely, rather than allowing at most one swap without interaction. The TTDLC transforms are very similar to the TDLC transform, except they can be repeated multiple times instead of just once. The parties that end up holding the contract at the end isn't relevant; in any case, either of the two parties will be able to put a transaction on chain and settle.

2) The public keys of all of the N parties need to be known beforehand.

3) We also enforce that only the two parties that are trading the contract between each other (i.e. one of them is swapping out and the other is swapping in) need to be present at the time that the swap is made. All of the other $N-2$ parties do not need to be online for this, though everyone does need to be online at setup time.

It makes sense why this type of solution would be more practical than the TDLC; this allows much more liquidity with the contract. Our major goal here is to simulate cash settled futures contract trading as much as possible, except in a decentralized manner.

3.2 Signing

We need to do some signing initially (just like regular DLCs require us to sign CETs before we put the funding transaction on chain). We have to sign two types of transactions here:

3.2.1 Transformation Transactions

For this part, we use SIGHASH_ANYPREVOUT | ANYONECANPAY | SINGLE to sign off on the spending condition and amount of the 0th input and the entire 0th output.

For every pairwise distinct, unordered triple of people (i, j, k), person i must sign the following CET using SIGHASH_ANYPREVOUT | ANYONECANPAY | SINGLE:

Input 0 script pubKey (well it's the script pubKey of the previous output to be more precise..): (Person i & j multi-sig)

Input 0 amount: Same as funding TX amount

Output 0 script pubKey: (Person i & k multi-sig)

Output 0 amount: Same as funding TX amount

The output that Input 0 is spending from here may not actually exist yet (which is why we use SIGHASH_ANYPREVOUT!). We are essentially creating this transformation transaction preemptively so that in the event that there ends up being a multi-sig between Persons i and j as a result of the contract being handed around, Person j can substitute out and Person k can substitute in.

The total number of transactions created here is $O(N^3)$ (where N is the number of people).

3.2.2 CETs

For every pairwise distinct, unordered pair of people (i, j) multiplied by every oracle outcome, person i must sign the following transaction using SIGHASH_ANYPREVOUT:

Input 0 script pubKey: (Person i & j multi-sig)

Input 0 amount: Same as funding TX amount

Output 0: Pay to Person i's PKH (amount based on oracle outcome)

Output 1: Pay to (Person j + Oracle) || (Person i & OP_CLTV) (amount based on oracle outcome)

Note that all signatures can be made public to everyone.

The total number of transactions created here is $O(N^2 \cdot O)$ (where N is the number of people and O is the number of outcomes).

3.3 Put Funding TX on chain

The next step is to create a single initial DLC funding transaction between two arbitrary parties (WLOG let's say Persons 0 and 1) and put it on chain.

3.4 Creating Transformation Transactions

Let's say Persons i and j are currently in the contract (which implies a Funding UTXO between two exists at the moment). Let's say that person j wants to swap out and person k wants to swap in; in this case, both person j and k need to be online and interact, but person i does not need to be there. Person j and k will construct the following transaction by adding inputs and outputs to the base Transformation Transaction:

Input 0 script pubKey: (Person i & j multi-sig)

Input 0 amount: Same as funding TX amount

Input 1: Any UTXO spendable by Person k

Output 0 script pubKey: (Person i & k multi-sig)

Output 0 amount: Same as funding TX amount

Output 1: Some agreed upon amount is sent to Person j's PKH.

Person k will then insert in Person i's relevant Transformation Transaction signature (during the signing phase, Person i already created and published this signature) and sign themselves. They will then send the signed transaction to Person j, who can then sign and send the transaction into the mempool.

Any number of these transformation transactions can be put on chain, one after another. As a result, ownership of the contract can constantly change. Note that transformation transactions signatures cannot be reused since the TXID of the UTXO that is being passed around is constantly changing. However, it is possible for the same pair of parties to hold the contract at multiple points throughout the process.

3.5 Final CET

Once the oracle publishes an outcome, a CET can be put on chain by either party. WLOG assume that persons i and j are currently in the contract. Let's say that person j wants to be the one to publish the final settlement transaction. Person j will create and sign the following transaction:

Input 0 script pubKey: (Person i & j multi-sig)

Input 0 amount: Same as funding TX amount

Output 0: Pay to Person i's PKH (amount based on oracle outcome)

Output 1: Pay to (Person j + Oracle) || (Person i & OP_CLTV) (amount based on oracle outcome)

They will use Person i's signature which already exists from the signing phase to complete the transaction. They will then send this to the mempool and the contract will be settled.

3.6 Example

Let's take a look at an example of an on-chain view of the transactions that go into making swaps with the TTDLC.

First off, let's assume that the funding transactions looks like 3-1.

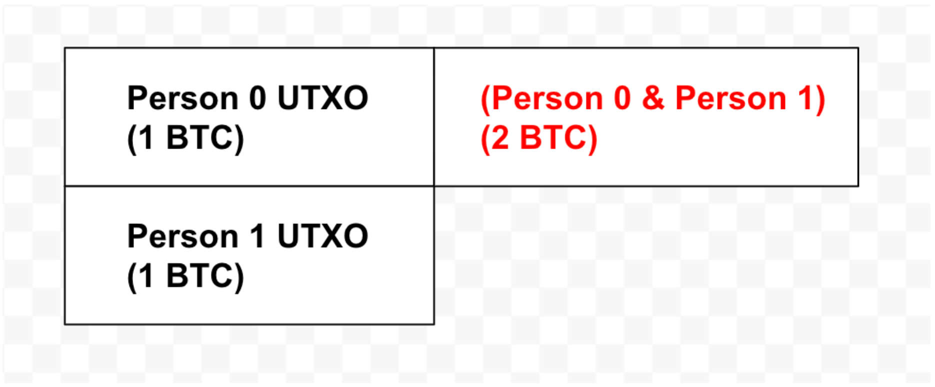

Figure 3-1: TTDLC Funding transaction

Let's say that Person 1 wants to be swapped out and replaced with Person 2. Also, Person 1 and Person 2 interact and agree that Person 2 should transfer 0.5 BTC to Person 1 in order for Person 2 to take Person 1's place in the contract. When put on chain, the transformation transaction would look like 3-2.

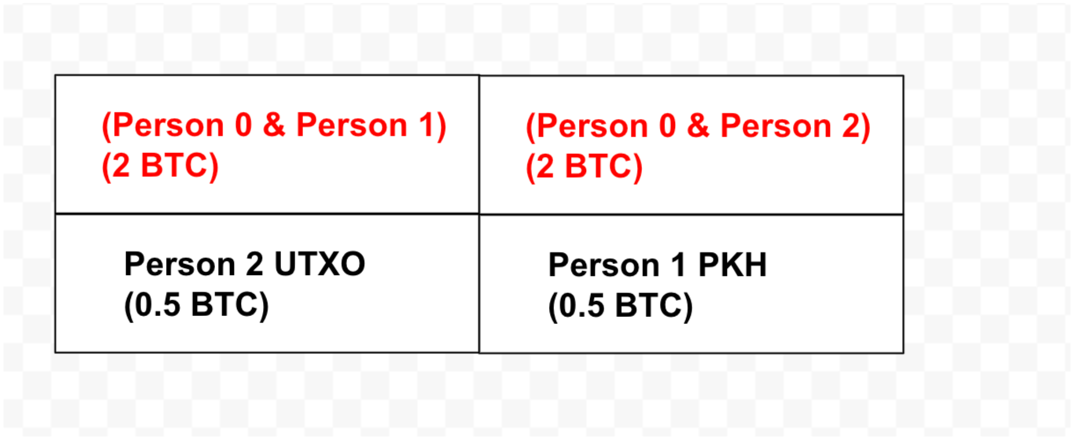

Figure 3-2: TTDLC Transformation transaction

Say that we reach expiration time and there are no more swaps. Either person 0 or person 2 can settle the contract. Let's say that Person 0 ends up settling the contract. The final CET would look like 3-3.

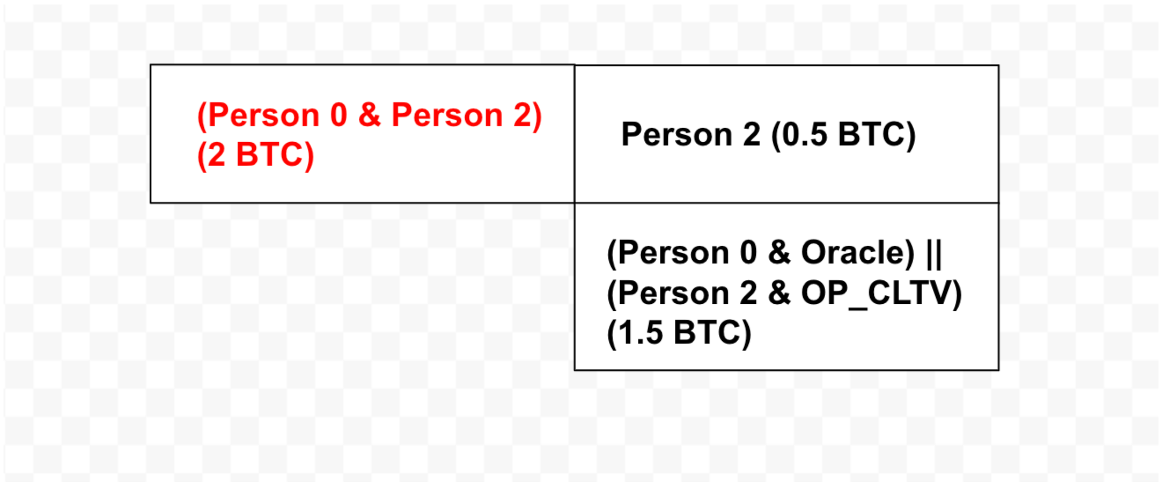

Figure 3-3: TTDLC CET

Presuming Person 0 put the correct CET on chain, Person 0 can then spend the second output using the correct oracle signature.

3.7 Security Considerations

There exist at least two interesting security considerations:

1. Say Person i and j are currently in the contract, and k wants to swap in for j. Person k signs a transformation transaction and sends it to j. Person j can now wait a while before they send the transaction into the mempool (to see if it is advantageous to do so). Person k has the option to respond to this by spending the UTXO they are using as an input, but it could become some type of race if they try to do this simultaneously. These races do not break correctness, but rather just make it unclear where k will get into the contract or not.

2. Say the oracle has already published a signature and the person who stayed in the contract at expiration and lost wants to be a sore loser. They can keep sending transformation transactions to prevent a CET from being put on chain. The party who lost can constantly observe the mempool and send a transformation transaction with a higher transaction fee than the party who is trying to settle. The deterrent here is that they have to do this every single block and they lose

the transaction fee, which adds up to pretty significant sums of money. One attack is for the sore loser to continue doing this until the CLTV on the CETs expire. In order to combat this, we can either make sure to put the CLTV far enough into the future to the point where this attack is always unprofitable or we can use CSV, which is a relative time lock and thus not susceptible to this attack, in the output script of the CET instead of CLTV, which is an absolute time lock.

3.8 Practical Use Cases

Right now, DLCs are rarely used on Bitcoin, though there are companies aiming to implement software and specifications to use them [15]. One of the reasons DLCs are currently rarely used might be because transferrability is not really present; the fact that the two parties at the beginning are forced to stay in the contract until the end unless both of them agree to leave is not particularly appealing. In fact, most proficient futures contract traders on centralized exchanges (which is what we're attempting to simulate to an extent, except we want the process to be entirely decentralized) generally aim to buy the contract low and sell it off at a higher price later on, before expiration. In simpler terms, waiting until expiration is generally not something that top traders do. The TTDLC will, therefore, enable DeFi applications on Bitcoin where a group of N traders can buy and sell the "futures contract" (really a DLC) as they please. This innovation will perhaps shift some favor towards Bitcoin in the DeFi landscape, as other blockchains, such as Ethereum, Avalanche, Solana, etc. are considered to be much stronger for DeFi at the moment.

The TTDLC is also useful as a theoretical result; we hope that our work will inspire others to innovate in the DLC space either by improving our research or coming up with a completely new idea. Any progress we make will improve the usability of Bitcoin and other UTXO models, which are key to maintaining high levels of decentralization and privacy on chain.

Chapter 4

Evaluation

4.0.1 TDLC Transaction Size Approximations

If we assume only one Carol, we're still only creating $O(o)$ signatures. However, if we assume that there are C carols, we need to create $O(C * o)$ signatures overall, which can be quite large. For example, if there are 100000 Carols and 100000 outcomes, the pre-computation phase would require 10^{10} signatures, which is much too large.

For each of these transactions, we only require signatures, each of which is 64 bytes. Therefore, the total size in bytes would be around $640 \cdot 10^9$, which is 640 GB. This is not impossible to store, but very large.

4.0.2 TTDLC Transaction Size Approximations

With the regular DLC and the TDLC, the number of transactions that need to be created and signed is quite low (on the order of $O(o)$, where o is the number of outcomes. For the TDLC, this becomes $O(C * o)$ if we assume multiple Carols). The TTDLC, because it allows for a much stronger use case, require many more signatures to be created. In particular, we must create $O(N^2 \cdot o + N^3)$ signatures in total; this suggests that $N > 2000$ will be extremely difficult (and even $N > 1000$ is quite difficult). In terms of size in bytes, $2000^3 = 8 \cdot 10^9$ signatures. Each signature is 64 bytes, so this translates to 512 GB, which is huge. Therefore, to some extent, the algorithm assumes that the quantity $N^2 \cdot o + N^3$ is low enough for the process to be practical. The time to compute these signatures would also pose a hurdle.

Chapter 5

Conclusion

5.0.1 Future Work

It was noted that even TTDLCs require the full amount of capital to be committed by both participants beforehand. Interesting future work would be to allow some kind of leveraged trading on Bitcoin; essentially both participants would only have to commit to some percentage of the maximum amount of capital and could be liquidated if the price moves against them enough.

Furthermore, throughout this thesis, we have only described the TDLC and TTDLC using Segwit V0. Both of these protocols would perhaps be better implemented with Taproot, Adaptor Signatures, or Segwit V1; this could be a point of future work [13] [14].

5.0.2 Recap

In this thesis, we discussed the Transformable Discreet Log Contract (TDLC) as well as the Truly Transformable Discreet Log Contract (TTDLC), both of which assist with DLC transferability. The core idea behind the former is to allow potentially multiple Carols to substitute into a DLC between Alice and Bob, though the public keys of all potential Carols must be known beforehand. At most one of these Carols can then swap in for either Alice or Bob midway through the contract. Only the Carol swapping in and the party swapping out need to be present to make the substitution. The Carol swapping in can then put a CET on chain to close the contract. The TTDLC is much stronger; it allows for many parties to essentially arbitrarily trade around a contract as long as the party swapping out and the party swapping are present to make the substitution. However, the TTDLC requires a potentially much larger setup phase.

We hope that our work will improve DLC transferability and general programmability on Bitcoin going forward.

Bibliography

[1] Adams et al.

"Uniswap v2 Core."

March 2020.

Uniswap.

`https://uniswap.org/whitepaper.pdf`

[2] "Concurrency and Cardano."

Jan 2022.

https://builtoncardano.com/blog/concurrency-and-cardano-a-problem-a-challenge-or-nothing-to-worry-about

[3] Dryja, Thaddeus.

"Discreet Log Contracts."

`https://adiabat.github.io/dlc.pdf`

[4] Hayes, Adam.

"How Do Futures Contracts Work?"

Oct 2021.

Investopedia.

`https://www.investopedia.com/terms/f/futurescontract.asp`

[5] Lombrozo et al.

“BIP 0141."

Dec 2015.

https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki

[6] Nakamoto.

“Nakamoto, Satoshi. Bitcoin: A Peer-to-Peer Electronic Cash System."

October 2008.

https://bitcoin.org/bitcoin.pdf

[7] Poon and Dryja.

“The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments."

2016.

https://lightning.network/lightning-network-paper.pdf

[8] Wood, Gavin.

“Ethereum: A secure decentralised generalised transaction ledger Berlin version."

Mar 2022.

https://ethereum.github.io/yellowpaper/paper.pdf

[9] Decker, Christian and Towns, Anthony.

“SIGHASH ANYPREVOUT for Taproot Scripts."

Feb 2017.

https://bips.xyz/118

[10] Gibson, Adam.

“SNICKER BIP draft"

https://gist.github.com/AdamISZ/2c13fb5819bd469ca318156e2cf25d79

[11] “Exchange, Decentralized"

Bisq.

`https://bisq.network/`

[12] Rubin, Jeremy.

“Designing Bitcoin Contracts with Sapio"

Sapio.

`https://learn.sapio-lang.org/`

[13] Wuille, Pieter.

“Taproot proposal"

May 2019.

`https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2019-May/016914.html`

[14] Ruffing et al.

“Adaptor Signatures and Atomic Swaps from Scriptless Scripts"

`https://github.com/ElementsProject/scriptless-scripts/blob/master/md/atomic-swap.md`

[15] “Bitcoin Settled Derivatives"

Suredbits.

`https://suredbits.com/`