Bitmatrix: A Constant Product Market Maker Based on Recursive Covenants

Burak Keceli
burak@bitmatrix.app

Bitmatrix is a constant product market maker built on Liquid Network, which utilizes the power of bitcoin opcodes to enable automated liquidity provision across L-BTC and other Liquid assets.

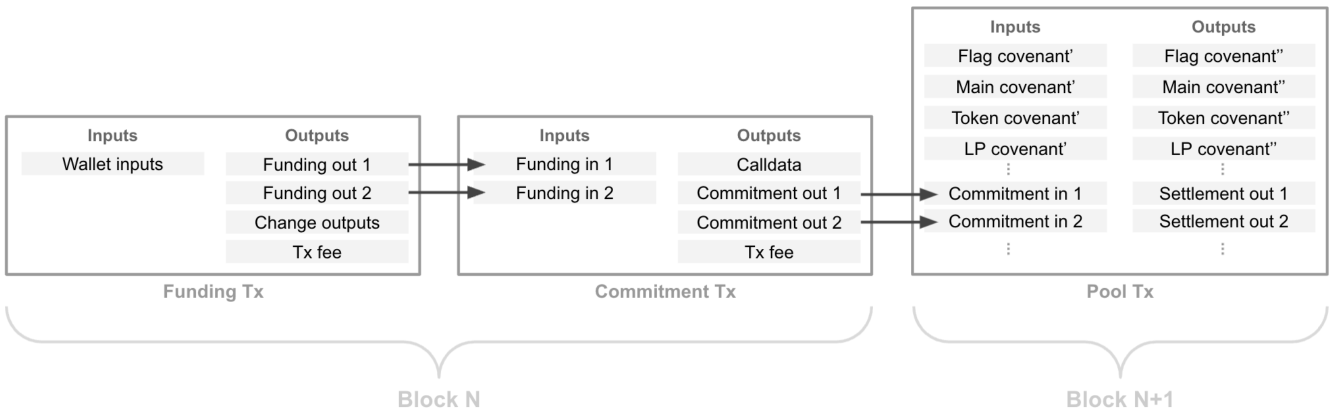
1. Introduction

Bitmatrix is a covenant-based AMM protocol where users can create their pool, add liquidity to an existing pool, or swap two assets. A Bitmatrix pool is made of covenants that interoperate and enforce constraints, resulting in a fully-functional AMM based on the UTXO model. Bitmatrix users can interact with pools across multiple front-ends, such as the web-based Bitmatrix interface, a mobile device, or an Elements node. Whether swapping two assets or adding or removing liquidity, numerous users can asynchronously transact regardless of the type of interaction.

A Bitmatrix interaction involves three subsequent transactions;

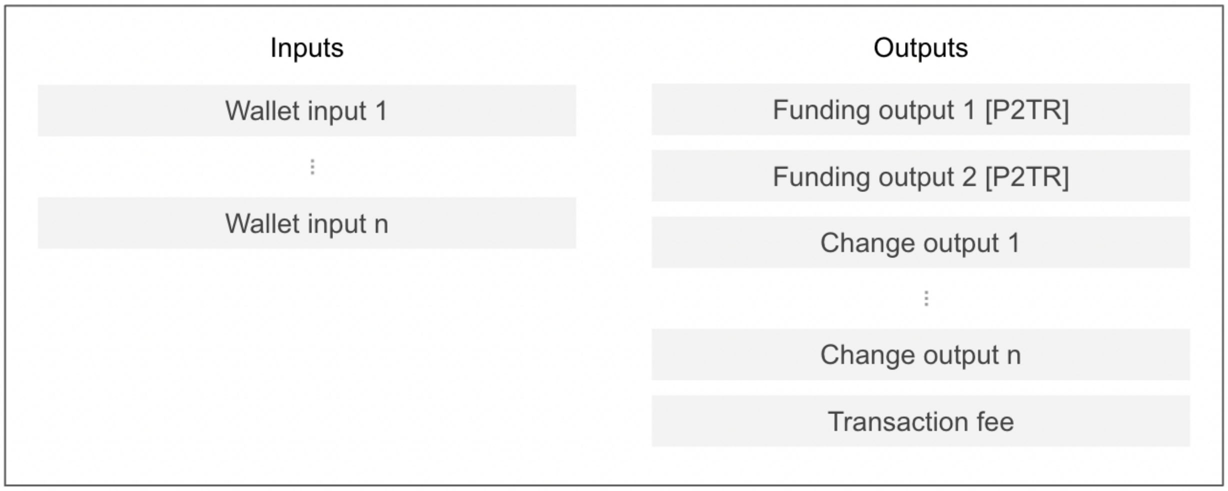
1. Funding transaction
2. Commitment transaction
3. Pool transaction

The *funding and commitment transactions* are created and broadcasted altogether, forming an unconfirmed parent-child chain. Outputs from *commitment transactions* are constrained to one block relative locktime, making them spendable in *pool transactions* once they are confirmed. This measure is taken to avoid attack vectors related to transaction pinning. Given deterministic one-minute blocks in Liquid, a Bitmatrix interaction can take one to two minutes to finalize, depending on when the interaction is attempted.



2. Funding Transaction

A Funding Transaction is signed and broadcasted by the user's wallet upon attempting to swap two assets or add/remove liquidity over a Bitmatrix pool. A Funding Transaction can come with various wallet inputs and change outputs but must construct two exact funding outputs to spend in the subsequent commitment transaction.



Funding Tx

2.1 Commitment Transaction Output Template

A template of Commitment Transaction outputs is mapped out before constructing a Funding Transaction. It's a 309-byte construction made of 11 items concatenated end to end. A single SHA256 of the entire bytedata is named "Commitment Transaction Output Template Hash".

Items	**Bytes**
Calldata [OP_RETURN] scriptPubKey	81
Calldata [OP_RETURN] amount	8
Calldata [OP_RETURN] asset ID	32
Commitment output 1 [P2TR] scriptPubKey	34
Commitment output 1 [P2TR] amount	8
Commitment output 1 [P2TR] asset ID	32
Commitment output 2 [P2TR] scriptPubKey	34
Commitment output 2 [P2TR] amount	8
Commitment output 2 [P2TR] asset ID	32
Transaction fee amount	8
Transaction fee asset ID	32

2.2 Funding Outputs

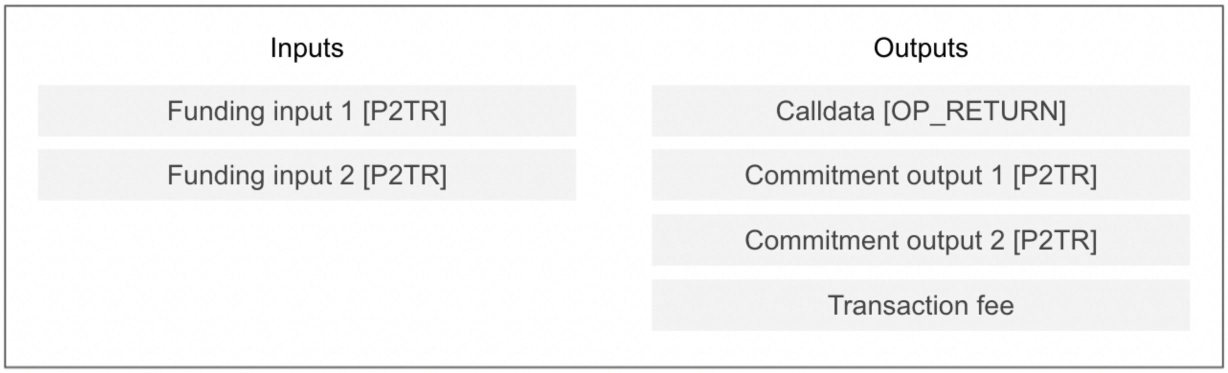
Funding output 1 and Funding output 2 are P2TR constructions whose tapscript constrain both outputs to a pre-defined *commitment transaction output template hash*:

```
//Inspect 1st output
<0> INSPECTOUTPUTSCRIPTPUBKEY <0> INSPECTOUTPUTVALUE <0> INSPECTOUTPUTASSET

//Inspect 2nd output
<1> INSPECTOUTPUTSCRIPTPUBKEY <1> INSPECTOUTPUTVALUE <1> INSPECTOUTPUTASSET

//Inspect 3rd output
<2> INSPECTOUTPUTSCRIPTPUBKEY <2> INSPECTOUTPUTVALUE <2> INSPECTOUTPUTASSET

//Inspect fee output
<3> INSPECTOUTPUTVALUE <3> INSPECTOUTPUTASSET

//Compute Commitment Transaction Output Template Hash
CAT CAT CAT SHA256

//Compare computed hash to stored hash
<Commitment Transaction Output Template Hash> EQUAL
```

The script TapLeaf should be tweaked with users' recovery key in case a *commitment transaction template* is somehow non-constructible. scriptPubKeys of *funding output 1* and *funding output 2* are accordingly created to contain the final tweaked result. The funding transaction is signed and broadcasted afterward.

3. Commitment Transaction

A commitment transaction commits to reserving a slot in a pool transaction. It's made of a standardized construction of 2 inputs and 4 outputs. The two funding outputs from the *funding transaction* are spent in the *commitment transaction* while no additional inputs are permitted. A *commitment transaction* is made of two commitment outputs, namely *commitment output 1* and *commitment output 2*. Plus an OP_RETURN and a fee output.



Commitment Tx

Commitment output 1 carries L-BTC supply at all times, in contrast to *commitment output 2* whose asset type depends on the type of interaction:

Case	Commitment Output 1 Asset	Commitment Output 2 Asset
Swap L-BTC for Token	L-BTC (fees + supply)	L-BTC (dust)
Swap Token for L-BTC	L-BTC (fees)	Token (supply)
Add Liquidity	L-BTC (fees + supply)	Token (supply)
Remove Liquidity	L-BTC (fees)	LP (supply)

3.2 Calldata

The first output of every *commitment transaction* is an OP_RETURN followed by a 78-byte PUSHDATA called Calldata. Aggregator nodes parse this data to interpret how to spend commitment outputs in a *pool transaction*. The 78-byte push is a concatenation of 5 items:

Calldata Items	Bytes
Target Flag Asset ID	32
Method Call	1
Recipient Public Key	33
Slippage Tolerance	8
Ordering Fees	4

3.2.1 Target Flag Asset ID

The first item in Calldata is a 32-byte Target Flag Asset ID indicating the destination *pool transaction* where commitment outputs are made only spendable. As each Bitmatrix trading pair runs on different *pool transactions*, and each *pool transaction* is identifiable via a unique flag asset, commitment outputs are constrained to the target pool's flag asset ID.

3.2.2 Method Call

The second item in Calldata is a single-byte Method Call indicating the type of interaction the user is aiming for with the Bitmatrix pool. Associated bytecodes are as follows:

Method Types	Bytecode
Swap L-BTC for Token	0x01
Swap Token for L-BTC	0x02
Add Liquidity	0x03
Remove Liquidity	0x04

3.2.3 Recipient Public Key

The third item in Calldata is a 33-byte compressed public key used to represent;

1. Settlement Output Witness Program

Recipient Public Key is encoded in a P2WSH construction where Settlement Outputs in *pool transaction* live in:

```
<0x01000000> CHECKSEQUENCEVERIFY <Recipient Public Key> CHECKSIG
```

The script enforces one block relative locktime to avoid reverse transaction pinning attacks that will later be explained in the paper. A v0 P2WSH witness program is preferred since a much less efficient script-path must be followed if this otherwise was a v1 P2TR witness program.

2. Recovery key

Recipient Public Key is encoded in a timeout path where Commitment Outputs in *commitment transactions live* (in section 3.3).

3.2.4 Slippage Tolerance

The fourth item in Calldata is an 8-byte Slippage Tolerance indicating the minimum amount of swap amount users can receive where anything below gets programmatically canceled. Slippage tolerance must be a valid script number and is set to 0x0000000000000000 if the method call is a non-swap type (0x03 or 0x04).

3.2.5 Ordering Fees

The sixth item in Calldata is a 4-byte ordering fees field indicating the amount of sats users are offering to reserve a slot in *pool transaction*. Commitment outputs in *pool transactions* are ordered according to a fee market which is determined by *ordering fees*. Ordering fees is a signed 32-bit integer and must be greater than 1.

3.3 Commitment Outputs

Commitment output 1 and output 2 are P2TR constructions whose Tapscript constrain both outputs to a Target Flag Asset ID in a success case and a Recipient Public Key in a timeout case. The following TapScript is given to commitment outputs:

```
//Success path:
IF
<0x01000000> CHECKSEQUENCEVERIFY
<0> INSPECTINPUTASSET <Target Flag Asset ID> EQUAL

//Timeout path:
ELSE
<0> INSPECTINPUTASSET <Target Flag Asset ID> EQUAL NOT VERIFY
<0x3c000000> CHECKSEQUENCEVERIFY
<Recipient Public Key> CHECKSIG
ENDIF
```

The script compiles to bytecode (variables surrounded with square brackets):

```
0x630401000000b200c820[TargetFlagAssetID]876700c820[TargetFlagAssetID]879169043c000000b221[
RecipientPublicKey]ac68
```

The script TapLeaf is tweaked with a point with an unknown discrete logarithm to eliminate a key-path spend:

```
0x021dae61a4a8f841952be3a511502d4f56e889ffa0685aa0098773ea2d4309f624
```

A P2TR witness program is created from the final tweaked key result and is given to both Commitment output 1 and Commitment output 2 scriptPubkeys.

3.1 Tapscript Mask

In order to inspect user-provided P2TR commitment inputs further in *pool transaction*, tapscript bytedata for each input is provided in witness. Inside the main covenant script, each tapscript bytedata from the witness is tweaked with a pre-determined internal key with an unknown discrete logarithm to match the tweaked keys in previous scriptPubkeys.

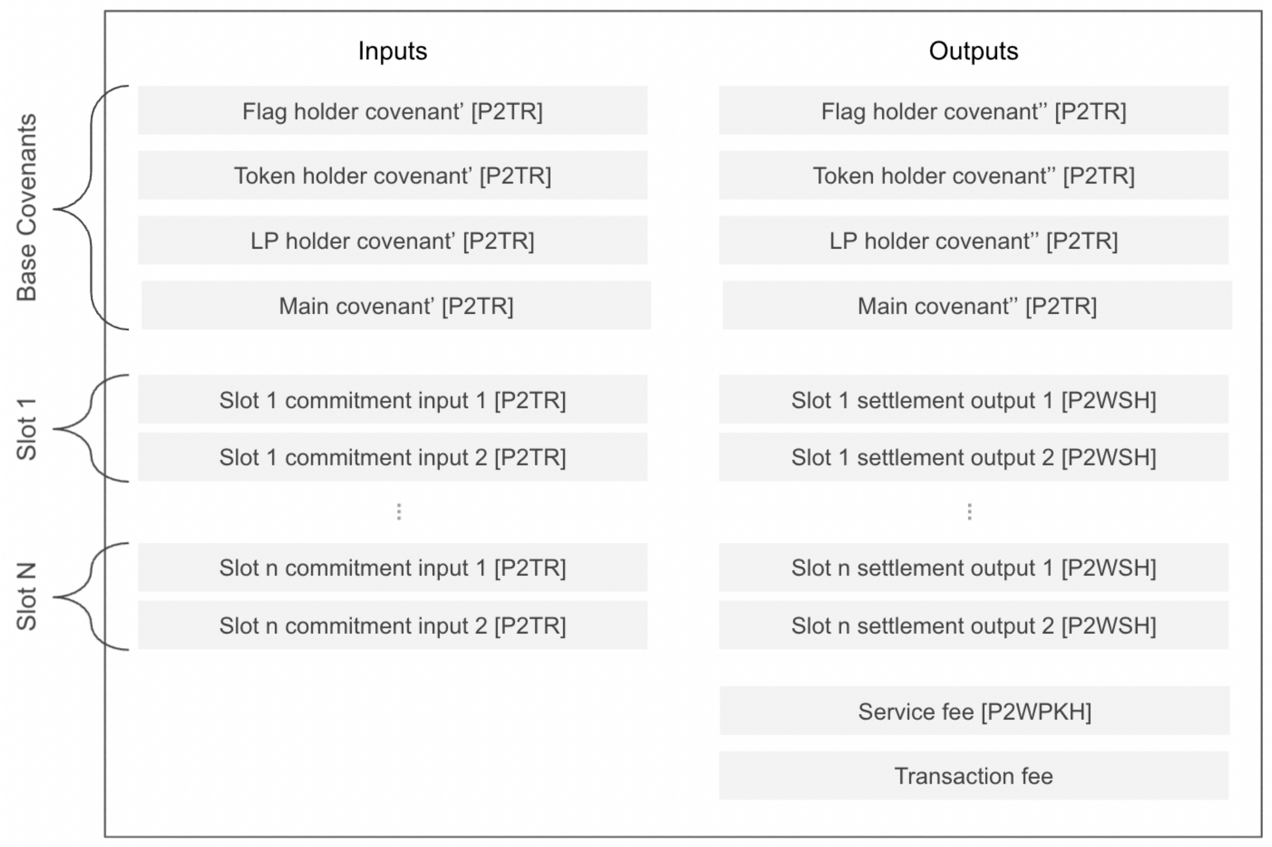
The witness script applies a bitwise AND between the encoded *tapscript Mask* and each tapscript bytedata and compares the result to tapscript bytedata themselves.

Replacing [RecipientPublicKey] of script bytedata with 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff yields a *tapscript Mask* to be harcoded in main covenant script:

```
0x630401000000b200c820[FLAG_ASSET_ID]876700c820[FLAG_ASSET_ID]879169043c000000b221fff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffac68
```

4 Pool Transaction

Pool Transaction is the main transaction where each interaction is executed in a global state. User commitment outputs are spent in this transaction as they commit to reserving a slot. These outputs are of the type anyone-can-spend and can collectively be spent.


Inputs
Outputs
Base Covenants
Flag holder covenant' [P2TR]
Token holder covenant' [P2TR]
LP holder covenant' [P2TR]
Main covenant' [P2TR]
Flag holder covenant'' [P2TR]
Token holder covenant'' [P2TR]
LP holder covenant'' [P2TR]
Main covenant'' [P2TR]
Slot 1
Slot 1 commitment input 1 [P2TR]
Slot 1 commitment input 2 [P2TR]
Slot 1 settlement output 1 [P2WSH]
Slot 1 settlement output 2 [P2WSH]
Slot N
Slot n commitment input 1 [P2TR]
Slot n commitment input 2 [P2TR]
Slot n settlement output 1 [P2WSH]
Slot n settlement output 2 [P2WSH]
Service fee [P2WPKH]
Transaction fee

Pool Tx

A Pool Transaction includes 4 covenant spends on the input side and 4 new covenant constructions on the output side, each from index #0 to index #4. A transaction fee and service fee output are placed at the bottom, with varying slots in between.

Each slot is made of two commitment inputs and two settlement outputs. Slots are ordered according to a fee market but are ordered lexicographically when they offer the same fee.

In order to avoid unconfirmed transaction chain forks in mempool to achieve a global state of a *pool transaction*, all covenants inside the *pool transaction* enforce one state transition per block. This is so that a new pool state transition is not valid until one block has elapsed since the previous pool state transition confirms.

4.1 Flag Holder Covenant

Flag holder covenant is a special type of covenant carrying a unique asset called Flag Asset with one unit supply and 0 precision. A flag asset is issued before pool creation and given to the Flag holder covenant at the point of pool creation. All three other covenants, Main covenant, Token holder covenant, and LP holder covenant, are constrained to Flag Asset ID and therefore made only spendable in the presence of Flag holder covenant. Flag holder covenant is spent at the input index #0 and re-constructed at the output index #0. The covenant TapScript is as follows:

```
//Make sure it's spent at the input index #0
PUSHCURRENTINPUTINDEX <0> EQUALVERIFY

//Make sure its re-constructed at the output index #0
PUSHCURRENTINPUTINDEX INSPECTINPUTASSET <0> INSPECTOUTPUTASSET EQUALVERIFY

//Make sure Token holder covenant, LP holder covenant, and Main covenant are included
<0> INSPECTINPUTOUTPOINT <0x00000000> EQUALVERIFY DUP DUP
<1> INSPECTINPUTOUTPOINT <0x01000000> EQUALVERIFY EQUALVERIFY
<2> INSPECTINPUTOUTPOINT <0x02000000> EQUALVERIFY EQUALVERIFY
<3> INSPECTINPUTOUTPOINT <0x03000000> EQUALVERIFY EQUALVERIFY
```

4.2 Token Holder Covenant

Token holder covenant carries Token liquidity as a pair to L-BTC liquidity held by main covenant. It’s always spent at the input index #1 and re-constructed at the output index #1. Its TapScript constrain funds to target asset ID:

```
//Make sure it’s spent at the input index #1
PUSHCURRENTINPUTINDEX <1> EQUALVERIFY

//Make sure its constrained to a target asset ID
<0> INSPECTINPUTASSET <0x01> EQUALVERIFY <Target Asset ID> EQUALVERIFY
```

4.3 LP Holder Covenant

LP holder covenant carries a pool of pre-minted LP tokens. When liquidity is added, LP tokens are given to liquidity providers proportional to added liquidity. When liquidity is removed, LP tokens are redeemed back to the pool proportional to removed liquidity. LP holder covenant is always spent at input index #2 and re-constructed at output #2. It’s TapScript constraining funds to flag asset ID:

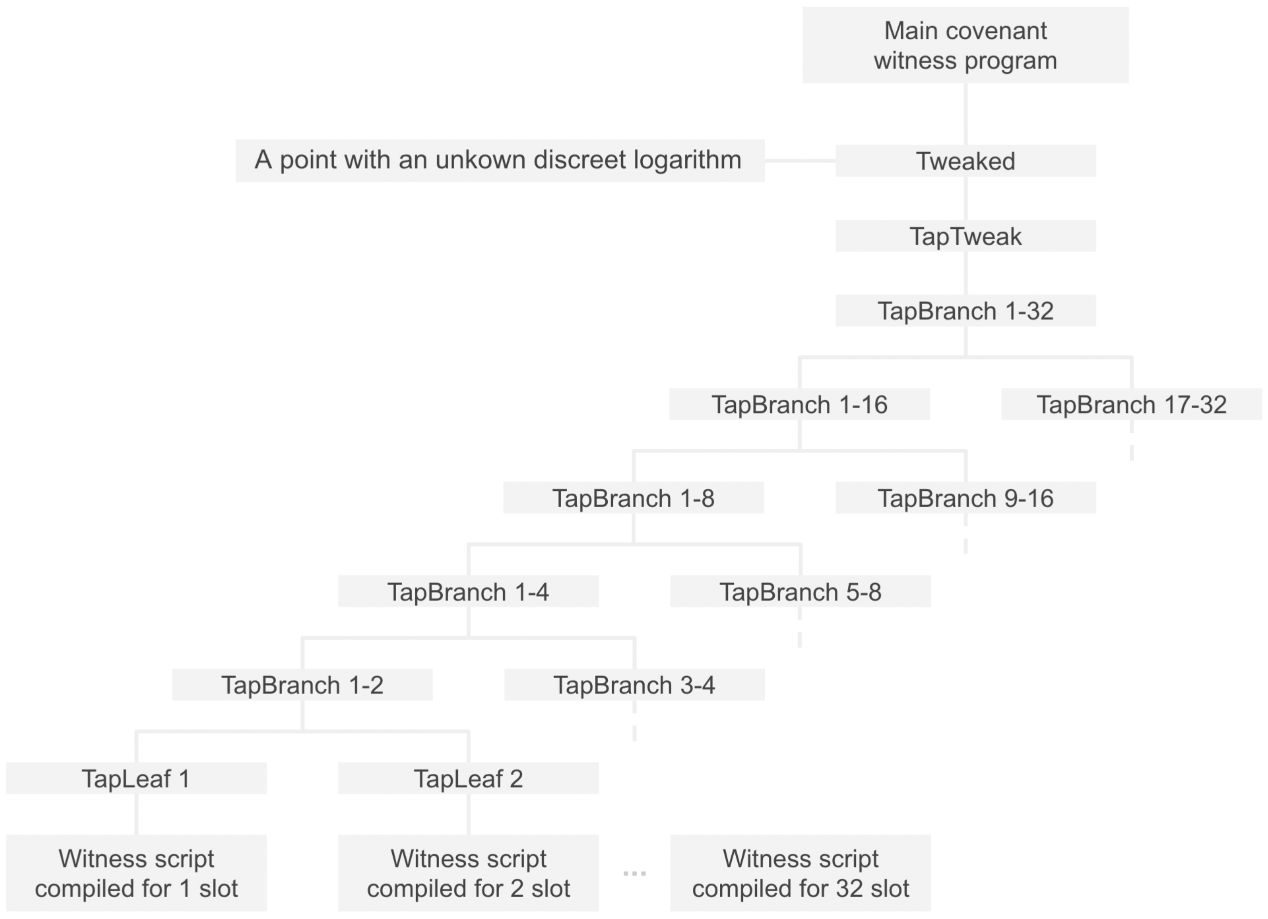
```
//Make sure it's spent at the input index #2
PUSHCURRENTINPUTINDEX <2> EQUALVERIFY

//Make sure its constrained to a target asset ID
<0> INSPECTINPUTASSET <0x01> EQUALVERIFY <Target Asset ID> EQUALVERIFY
```

Flag holder covenant, Token holder covenant, and LP holder covenant TapScripts are all tweaked with a point with an unknown discrete logarithm to eliminate a key-path spend:

```
0x021dae61a4a8f841952be3a511502d4f56e889ffa0685aa0098773ea2d4309f624
```

4.4 Main Covenant

The main covenant inside a pool transaction is a custom P2TR witness program that commits to a 32-leaf MAST construction. Each leaf index reflects the number of concurrent user calls in that leaf spend. As the number of leaves is bounded by 32 and given one-minute blocks in Liquid, a Bitmatrix pool can process up to 32 interactions per minute or 0.53 interactions per second.


Main covenant witness program
A point with an unkown discreet logarithm
Tweaked
TapTweak
TapBranch 1-32
TapBranch 1-16
TapBranch 17-32
TapBranch 1-8
TapBranch 9-16
TapBranch 1-4
TapBranch 5-8
TapBranch 1-2
TapBranch 3-4
TapLeaf 1
TapLeaf 2
Witness script compiled for 1 slot
Witness script compiled for 2 slot
...
Witness script compiled for 32 slot

When constructing MAST, script bytedata for a single slot is compiled 32 times, each time re-adding itself endwise the times of leaf index. When spending the main covenant, a 193-byte control block is mapped out for a leaf that accommodates the number of concurrent interactions that exist in that period. Lower index leaves can be replaced with higher index leaves when the number of concurrent user calls is less than 32. When the number of concurrent user calls hits 32 or greater, an internal fee market occurs, and therefore original versions of a 32nd leaf spend can be replaced with newer versions of a 32nd leaf spend.

4.4.1 Concurrency

By leveraging Tapscript (BIP 342), Relative lock-time (BIP 68), and Replace-by-fee (BIP 125), multiple users can asynchronously interact with a Bitmatrix pool in a fully on-chain logic.

4.4.1.1 Replacement Logic

A built-in replacement logic enforced by the Bitcoin script when leveraged with BIP-125 allows concurrent users to compete over a fee market to reserve a slot in a 32-seat pool transaction.

The four base covenants in a pool transaction and all user-provided inputs signal replaceability. This is so that anyone can submit newer versions of a pool transaction within a minute as long as paying a greater absolute fee plus their bandwidth according to BIP-125. Since commitment inputs are of-type anyone-can-spend, any Elements node can monitor the network and order user calls according to a fee market based on how much fee each offers.

Based on parameters;

I. b is the base fee amount that must be paid by every user set to 1200 sats.

II. o is the ordering fee a user can offer varying from 1 to 2147482647 sats.

III. m is the maximum number of slots available in a Pool Transaction set to 32 seats.

IV. c is the present number of reserved slots (user calls) in a Pool Transaction bounded by m

V. g is the vByte-length gap between two subsequent leaves in witness and base txn, set to 560 vBytes = 230 base bytes + 193 script body vBytes + 136 stack element vBytes.

VI. s is the number of vBytes single-leaf pool spend consumes, set to 1377 vBytes.

VII. r is the default Elements minimum relay fee set to 10 vByte/sat (0.1 sat/vByte).

Fees paid by the *transaction fee output* is calculated as $fp = \frac{\sum_{n=o1}^{oc} n+b}{3} + \frac{\left(\sum_{n=o1}^{oc} n+b\right) * 2 * c}{3 * m}$. The formula is based on allowing newer versions of pool transactions to be replaced with original versions. Given the total fees offered by all users $ft = \sum_{n=o1}^{oc} n + b$, the remaining fees are paid extra to service fee output $fr = ft - fp$. .

The fourth rule of BIP-125 requires that the replacement transaction must pay for its bandwidth at or above the rate set by the node's minimum relay fee setting.

A replacement Pool Transaction at index c bounded by m;

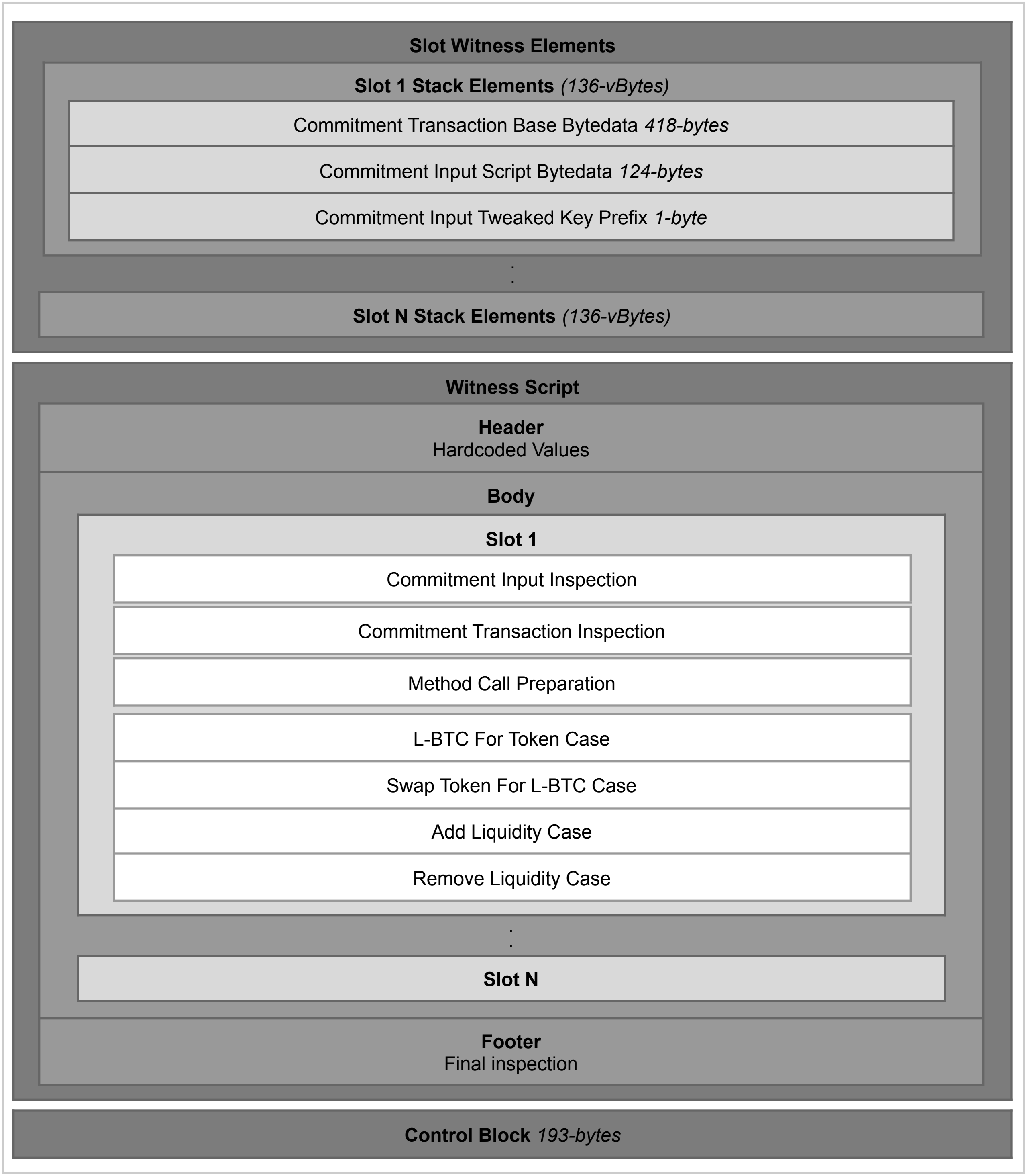
- Pays for bandwidth $\Delta b = \frac{\sum_{n=o1}^{oc} n+b}{3} - \frac{\sum_{n=o(c-1)}^{o(c-1)} n+b}{3} + \frac{\left(\sum_{n=o1}^{oc} n+b\right) *2 *c}{3 * m} - \frac{\left(\sum_{n=o(c-1)}^{o(c-1)} n+b\right) *2 *c}{3 * m}$

- Must pay for bandwidth $bp = \frac{s + (c-1)*g}{r}$

Given hardcoded values $m = 32, r = 10$, and $b = 1000$the statement $\Delta b > bp$ is true at all times when $c < m$ is true. An internal fee market competition occurs when $c = m$ becomes true.

4.4.2 Spend Structure

A single main covenant spend takes a 193-byte control block at the bottom, a varying-size witness script, and a varying number of slot inputs that satisfy the witness script, each taking 3 stack elements. Therefore the main covenant spend has a witness element count of $2 + 3n$ for a value of n that is an integer between 1 and 32.



Slot Witness Elements
Slot 1 Stack Elements *(136-vBytes)*
Commitment Transaction Base Bytedata *418-bytes*
Commitment Input Script Bytedata *124-bytes*
Commitment Input Tweaked Key Prefix *1-byte*
Slot N Stack Elements *(136-vBytes)*
Witness Script
Header
Hardcoded Values
Body
Slot 1
Commitment Input Inspection
Commitment Transaction Inspection
Method Call Preparation
L-BTC For Token Case
Swap Token For L-BTC Case
Add Liquidity Case
Remove Liquidity Case
Slot N
Footer
Final inspection
Control Block *193-bytes*

4.4.2.2 Slot Witness Elements

Each slot in the main covenant spend takes 3 stack elements, summing at 135.75-vBytes:

Hardcoded values	Bytes
Commitment Transaction Base Bytedata	418
Commitment Input Tapscript Bytedata	124
Commitment Input Tweaked Key Prefix	1

```
- - - - - - - - - - - - - - - - - - - - - BEGIN SLOT N STACK ELEMENTS - - - - - - - - - - - - - - - - - - -
.
.
- - - - - - - - - - - - - - - - - - - - - - END SLOT N STACK ELEMENTS - - - - - - - - - - - - - - - - - - - -
.
.
- - - - - - - - - - - - - - - - - - - - - BEGIN SLOT 1 STACK ELEMENTS - - - - - - - - - - - - - - - - - - -
//Commitment Transaction Base Bytedata
<418-bytes>
//Commitment Input Tapscript Bytedata
<124-bytes>
//Commitment Input Tweaked Key Prefix
<0x02 or 0x03>
- - - - - - - - - - - - - - - - - - - - - - END SLOT 1 STACK ELEMENTS - - - - - - - - - - - - - - - - - - - -
```

4.4.2.2.1 Commitment Transaction Base Bytedata

Non-witness bytedata of a standardized 419-base-byte *commitment transaction* is provided in witness. The witness script inspects transaction bytedata to read *calldata* within the serialization.

4.4.2.2.2 Commitment Input Script Bytedata

The tapscript bytedata of a commitment input is provided in witness. The witness script inspects the tapscript bytedata to avoid potential pinning attacks.

4.4.2.2.3 Commitment Input Tweaked Key Prefix

A TWEAKVERIFY operation is used to verify the authenticity of inspected tapscript. In order to input the tweaked key into TWEAKVERIFY, the commitment input scriptPubkey is inspected, and the 32-byte witness program is parsed. However, tweaked keys' one-byte prefix is unknown over introspection and is therefore provided in witness. The 32-byte witness program is concatenated with Commitment Input Tweaked Key Prefix to construct the full 33-byte tweaked key.

4.4.2.3 Witness Script

The main covenant witness script is a varying-size expressive tapscript that validates the correct execution of each slot. Witness script is divided into three parts for better conceptualizing:

1. Witness Script Header
2. Witness Script Body
3. Witness Script Footer

4.4.2.3.1 Witness Script Header

The script header starts with six values hardcoded at the beginning of the witness script. The values are used in certain operations taking place in the script body.

Hardcoded values	Bytes
L-BTC Asset ID	32
Token Asset ID	32
Commitment Input TapScript Mask	124
A point with an unknown discrete logarithm	32
The string “TapLeaf”	7
The string “TapTweak”	8

```
- - - - - - - - - - - - - - - - - - - - - - - - - - BEGIN SCRIP HEADER - - - - - - - - - - - - - - - - - - - - - - - - -

//L-BTC Asset ID
<0x6d521c38ec1ea15734ae22b7c46064412829c0d0579f0a713d1c04ede979026f>

//Token Asset ID
<0x[TOKEN_ASSET_ID]>

//Commitment Input TapScript Mask
<0x630401000000b200c820[FLAG_ASSET_ID]876700c820[FLAG_ASSET_ID]879169043c000000b221
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffac68>

//A point with an unkown discreet logarithm
<0x1dae61a4a8f841952be3a511502d4f56e889ffa0685aa0098773ea2d4309f624>

//The string “TapLeaf” in bytes
<0x5461704c656166>

//The string “TapTweak” in bytes
<0x546170547765616b>
```

.

```
//Inspect number of inputs
INSPECTNUMINPUT
//Constrain number of inputs
<Number of inputs in respective leaf spend>  EQUALVERIFY
//Inspect number of outputs
INSPECTNUMOUTPUT
//Constrain number of outputs
<Number of outputs in respective leaf spend> EQUALVERIFY
//Constrain main covenant to a target asset ID
<0> INSPECTINPUTASSET <0x01> EQUALVERIFY <Target Asset ID> EQUALVERIFY
//Inspect pool L-BTC supply
<3> INSPECTINPUTVALUE
//Verify the prefix(1)
<0x01> EQUALVERIFY
//Send pool L-BTC supply to altstack
TOALTSTACK
//Inspect pool Token supply
<1> INSPECTINPUTVALUE
//Verify the prefix(1)
<0x01> EQUALVERIFY
//Send pool Token supply to altstack
TOALTSTACK
//Inspect pool LP supply and verify the prefix(1)
<2> INSPECTINPUTVALUE <0x01> EQUALVERIFY
//Send pool LP supply to altstack
TOALTSTACK
//Send int 0 (total tx fees) to altstack
<0> TOALTSTACK
//Send max 32-bit signed int to altstack (to compare 1.st index ordering fee)
<2147483647> TOALTSTACK

 - - - - - - - - - - - - - - - - - - - - - - - - - END SCRIP HEADER - - - - - - - - - - - - - - - - - - - - - - - - - -
```

4.4.2.3.2 Witness Script Body

The witness script header is followed by a varying-size witness script body making the most critical part of the overall covenant design. The witness script body hosts script enforcements for each slot. It can accommodate a minimum of 1 and a maximum of 32 slots which are serialized end-to-end and executed in order. Each slot block consists of 4 parts:

1. Commitment Input TapScript Inspection
2. Commitment Transaction Calldata Inspection
3. Method Call Preparations
4. Method Call Cases

4.4.2.3.2.1 Commitment Input Inspection

The witness-provided commitment input tapscript bytedata is inspected to match a pre-defined script pattern (Tapscript mask). A TWEAKVERIFY operation is used to verify that commitment input tapscript bytedata really commits to scriptPubKeys of commitment inputs:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - BEGIN SCRIP BODY - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - - - - - - BEGIN SLOT 1 - - - - - - - - - - - - - - - - - - - - - - - - - - -

//Calculate TapLeaf given script bytedata
<1> PICK SHA256 DUP CAT <0xc0> CAT <7> PICK SIZE SWAP CAT CAT SHA256

//Calculate TapTweak given hardcoded internal key and TapLeaf
<1> PICK SHA256 DUP CAT <4> PICK <2> ROLL CAT CAT SHA256

<3> PICK SWAP <8> ROLL

//Make sure commitment inputs share the same scriptPubkey, and inspect their tweaked key
<3+N> INSPECTINPUTSCRIPTPUBKEY DUP <4+N> INSPECTINPUTSCRIPTPUBKEY EQUALVERIFY

//Verify witness version 1
<1> VERIFY

//TweakVerify given hardcoded internal key, TapLeaf, and inspected tweaked key
TWEAKVERIFY

//A bitwise AND between TapScript Mask and each tapscript bytedata from the witness are computed,
and the result is compared to each tapscript bytedata.
<6> ROLL DUP <5> PICK AND EQUALVERIFY

//Fetch commitment input 1 outpoints and verify their output index
<N> INSPECTINPUTOUTPOINT <0x01000000> EQUALVERIFY DUP

//Fetch commitment input 2 outpoints and verify their output index
<N+1> INSPECTINPUTOUTPOINT <0x02000000> EQUALVERIFY

//Verify commitment input 1  and commitment input 2 are from the same previous transaction ID
EQUALVERIFY
```

4.4.2.3.2.2 Commitment Transaction Inspection

The witness-provided commitment transaction base bytedata is hashed to match the inspected commitment input outpoints. Then the whole transaction bytedata is parsed to extract individual calldata elements within the transaction serialization:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - CONTINUE SLOT 1 - - - - - - - - - - - - - - - - - - - - - - - - - -

///////Fetch commitment transaction base bytedata, and make 4 copies
<7> ROLL DUP 2DUP

///////Hash base tx bytedata and compare to commitment input prvtxid
HASH256 <4> ROLL EQUALVERIFY

///////Enforce size 418-bytes
SIZE <418> EQUALVERIFY

//Parse 2nd sequence field and EQUALVERIFY ffffffff
<84> <4> SUBSTR <0xffffffff> EQUALVERIFY

//Parse number of inputs and make sure its 4
<88> <1> SUBSTR <0x04> EQUALVERIFY

//Parse the first output (OP_RETURN)
<133> <81> SUBSTR

//Make sure its an op_return and parse pushdata
DUP <0> <1> SUBSTR <0x6a> EQUALVERIFY <3> <78> SUBSTR

//Parse target flag asset ID and verify
DUP <0> <32> SUBSTR

//Inspect flag holder covenant asset ID and verify the prefix(1)
<0> INSPECTINPUTASSET

//Verify the prefix(1)
<0x01> EQUALVERIFY

//Constrain user call to Target Asset ID
EQUALVERIFY

//Parse calldata to extact method call
DUP <32> <1> SUBSTR

//Parse calldata to extact recipient public key
SWAP DUP <33> <33> SUBSTR

//Parse calldata to extact slippage tolerance
SWAP DUP <66> <8> SUBSTR

//Parse calldata to extact ordering fees
SWAP <74> <4> SUBSTR
```

4.4.2.3.2.3 Method Call Preparation

A set of preparations made to get the stack ready for final method call cases:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - CONTINUE SLOT 1 - - - - - - - - - - - - - - - - - - - - - - - - - -

//Get lower index ordering fees from the stack
FROMALTSTACK

//Get total ordering fees from the stack
FROMALTSTACK

//Make a copy of current-index ordering fees
<2> PICK

//Update total ordering fees
ADD

//Send updated total ordering fees to altstack
TOALTSTACK

//Send lower index ordering fees to altstack
TOALTSTACK

//Make a copy of current ordering fees
DUP

//Make sure current ordering fees is greater than 1 sat
<1> GREATERTHANOREQUAL VERIFY

//Inspect commitment input 1 asset and verify the prefix(1)
<3+N> INSPECTINPUTASSET <0x01> EQUALVERIFY

//Make a copy of the L-BTC asset ID (copying from the script header)
<10> PICK

//Verify commitment input 1 assetID(32) is L-BTC
EQUALVERIFY

//Make a copy of current-index ordering fees and get lower index ordering fees from altstack
DUP FROMALTSTACK

//Make a copy of current-index ordering fees lower-index ordering fees
2DUP

//Compare current-index ordering fees to lower-index ordering fees
LESSTHAN

//If current-index ordering fees is smaller than the lower-index ordering fees
IF

//Ordering is based on a valid fee market. Drop copies of current-index and lower-index ordering fees
2DROP

//If current-index ordering fees is not smaller than the lower-index ordering fees
ELSE
```

```
//Both ordering fees must be equal; otherwise, fail. Lexicographical ordering will be enforced.
EQUALVERIFY

//Inspect current-index slot commitment input 1 outpoint
<3+N> INSPECTINPUTOUTPOINT

//Remove vout(4) and outpoint_flag(1) to keep txid(32)
2DROP

//Splice the first 8 bytes of current-index commitment input txid(32)
<8> LEFT

//Inspect lower-index slot commitment input 1 outpoint
<1+N> INSPECTINPUTOUTPOINT

//Remove vout(4) and outpoint_flag(1) to keep txid(32)
2DROP

//Splice the first 8 bytes of lower-index commitment input txid(32)
<8> LEFT

//Enforce lexicographical ordering by comparing the two previous txid(32)
LESSTHAN64 VERIFY

//Drop copies of current-index and lower-index ordering fees
2DROP

ENDIF

//Make a copy of current ordering fees
DUP

//Send the copy of current ordering fees to altstack for higher-index slot comparisons
TOALTSTACK

//Add base fee 1200 sats to current ordering fees, making up total tx fees
<1200> ADD

//Convert total tx fees to LE64
SCIPTNUMTOLE64

//Inspect commitment input 1 L-BTC value and verify the prefix(1)
<3+N> INSPECTINPUTVALUE <0x01> EQUALVERIFY

SWAP
//Subtract total L-BTC provided from tx fees, making up available user L-BTC amount remaining
SUB64

//Make sure available user L-BTC remaining is greater than 500 sats
DUP <500> SCIPTNUMTOLE64 GREATERTHAN64 VERIFY

//Get current-index ordering fees from altstack
FROMALTSTACK
//Get up-to-latest-index total accumulated fees from altstack
FROMALTSTACK
```

```
//Get current-index ordering fees from altstack
FROMALTSTACK
//Update total accumulated fees by adding the current-index base fee
<1200> ADD
//Get LP token pool supply from altstack
FROMALTSTACK
//Get Token pool supply from altstack
FROMALTSTACK
//Get L-BTC pool supply from altstack
FROMALTSTACK
//Calculate P2WSH witness program given receipent public key
<0x0401000000b221> <8> ROLL <Recipient Public Key> <0xac> CAT CAT SHA256
//Inspect commitment output1 scriptPubkey
<3+N> INSPECTOUTPUTSCRIPTPUBKEY
//Constrain commitment output1 witness version to v0:
<0> EQUALVERIFY
//Constrain commitment output1 witness program to calculated witness program
EQUALVERIFY
//Fetch method call extracted from calldata
<7> ROLL
//If the method call is either 1, 2, or 3:
DUP <3> LESSTHANOREQUAL
IF
//Inspect settlement output 2 asset
<4+N> INSPECTOUTPUTASSET
//Verify the prefix(1)
<0x01> EQUALVERIFY
//Verify the assetID(32) is L-BTC
<14> PICK EQUALVERIFY
//Inspect settlement output 2 value
<4+N> INSPECTOUTPUTVALUE
//Verify the prefix(1)
<0x01> EQUALVERIFY
//Verify the value(8) is 0
<0> SCIPTNUMTOLE64 EQUALVERIFY
//Verify scriptPubKey is an empty OP_RETURN
INSPECTOUTPUTSCRIPTPUBKEY <-1> EQUALVERIFY <0x6a0100> SHA256 EQUALVERIFY

ENDIF
```

4.4.2.3.2.4 Method Call Cases

One of four code blocks is executed based on the method call parsed from the calldata:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - CONTINUE SLOT 1 - - - - - - - - - - - - - - - - - - - - - - - - - - -

//If the method call is 1:
DUP <1> EQUAL IF DROP
 - - - - - - - - - - - - - - - - - - - - - - - BEGIN CASE 1 : Swap L-BTC for Token - - - - - - - - - - - - - - - - - - - - - - -
                                        .
                                        .
 - - - - - - - - - - - - - - - - - - - - - - - END CASE 1 : Swap L-BTC for Token - - - - - - - - - - - - - - - - - - - - - - -
ENDIF

//If the method call is 2:
DUP <2> EQUAL IF DROP
 - - - - - - - - - - - - - - - - - - - - - - - BEGIN CASE 2 : Swap Token for L-BTC - - - - - - - - - - - - - - - - - - - - - - -
                                        .
                                        .
 - - - - - - - - - - - - - - - - - - - - - - - END CASE 2 : Swap Token for L-BTC - - - - - - - - - - - - - - - - - - - - - - -
ENDIF

//If the method call is 3:
DUP <3> EQUAL IF DROP
 - - - - - - - - - - - - - - - - - - - - - - - - - - - BEGIN CASE 3 : Add Liquidity - - - - - - - - - - - - - - - - - - - - - - - -
                                        .
                                        .
- - - - - - - - - - - - - - - - - - - - - - - - - - - - END CASE 3 : Add Liquidity - - - - - - - - - - - - - - - - - - - - - - - - -
ENDIF

//If the method call is 4:
DUP <4> EQUAL IF DROP
 - - - - - - - - - - - - - - - - - - - - - - - - - BEGIN CASE 4 : Remove Liquidity - - - - - - - - - - - - - - - - - - - - - -
                                        .
                                        .
- - - - - - - - - - - - - - - - - - - - - - - - - - END CASE 4 : Remove Liquidity - - - - - - - - - - - - - - - - - - - - - - -
ENDIF
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - END SLOT 1 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                                        .
                                        .
                                        .
                                        .
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - BEGIN SLOT N - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                                        .
                                        .
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - END SLOT N - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - END SCRIPT BODY - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

4.4.2.3.2.4.1 Swap L-BTC for Token

A swap-in L-BTC amount, a base fee of 1200, an ordering fee varying from 1-2147483647, and a dust 650 sats are supplied in the commitment transaction. The pool transaction constrains settlement output 1 to the Token swap-out amount based on the CPMM formula in a success case and to the initial L-BTC swap-in amount in a slippage failure case.

Inputs	**Asset**	**Amount**
Commitment Input 1	L-BTC	Swap-in Amount + Base Fee (1200) + Ordering Fee (1-2147483647)
Commitment Input 2	L-BTC	Dust (650)
Outputs (Success)	**Asset**	**Amount**
Settlement Output 1	Token	Swap-out Amount
Settlement Output 2	L-BTC	0 [OP_RETURN]
Outputs (Out of Slippage)	**Asset**	**Amount**
Settlement Output 1	L-BTC	Swap-in Amount
Settlement Output 2	L-BTC	0 [OP_RETURN]

Based on parameters;

I. $sl$is a state of L-BTC pool supply held by the main covenant
II. $st$is a state of Token pool supply held by the Token holder covenant
III. si is the swap-in L-BTC amount that the user commits
IV. lf is the liquidity provider fee charged from the swap-in amount
V. so is the swap-out Token amount that the user receives
VI. s is the Token slippage limit

1. A %0.2 LP fee is calculated $lf = \frac{si}{500}$
2. New tradable swap-in amount $si' = si - lf$
3. L-BTC pool supply for constant $sl' = sl - si$
4. L-BTC pool supply for constant (downgraded 32x) $sl'' = \frac{sl'}{32}$
5. Token pool supply downgraded (downgraded 4194304x) $st' = \frac{st}{4194304}$
6. Tradable L-BTC swap-in amount downgraded (downgraded 32x) $si'' = \frac{si'}{32}$
7. The new state of Token pool supply $st''= \frac{sl''*st'}{si''}$
8. Final Token pool supply (magnified 4194304x) $st''' = st'' * 4194304$
9. Token user received $so = st - st'''$
10. Final L-BTC pool supply $sl''' = st + si$
11. Success if $s > so'$

```
- - - - - - - - - - - - - - - - - - - BEGIN CASE 1 : Swap L-BTC for Token - - - - - - - - - - - - - - - - - -
//Inspect commitment input 2 asset
<4+N> INSPECTINPUTASSET

//Verify the prefix(1)
<0x01> EQUALVERIFY

//verify the assetID(32) is L-BTC
<13> PICK EQUALVERIFY

//Inspect commitment input 2 value
<4+N> INSPECTINPUTVALUE

//Verify the prefix(1)
<0x01> EQUALVERIFY

//Verify the value(8) is dust
<650> SCIPTNUMTOLE64 EQUALVERIFY

//Fetch commitment input 1 tradable L-BTC supply
<5> ROLL

//Make two copies send one to altstack
DUP DUP TOALTSTACK

//Calculate LP fee %0.2
<500> SCIPTNUMTOLE64 DIV64

//Verify no overflow and  remove remainder
VERIFY NIP

//Subtract LP fees from the L-BTC amount and verify no overflow
SUB64 VERIFY

//Make a copy of the total L-BTC pool supply
<1> PICK

//Add existing pool supply with new pool supply to calculate new token pool supply and verify no overflow
ADD64 VERIFY

//Make a copy of the total L-BTC pool supply and downgrade precision (Minimum unit to 32 sats)
<1> PICK <32> SCIPTNUMTOLE64 DIV64

//Verify no overflow
VERIFY

//Make a copy of the total Token pool supply and downgrade precision (Minimum unit to 4194304)
<3> PICK <4194304> SCIPTNUMTOLE64 DIV64

//Verify no overflow
VERIFY

//Multiply the two downgraded pool supply to calculate the constant value and verify no overflow
MUL64 VERIFY
```

```
//Make a copy of L-BTC pool supply for constant and downgrade precision (Minimum unit to 32 sats)
<1> PICK <32> SCIPTNUMTOLE64 DIV64

//Verify no overflow
VERIFY

//Divide constant with L-BTC pool supply to calculate the new Token pool supply and verify no overflow
DIV64 VERIFY

//Magnify the new Token pool supply and verify no overflow
<4194304> SCIPTNUMTOLE64 MUL64

//Verify no overflow
VERIFY

SWAP

//Make a copy of the previous Token pool supply
<3> PICK

//Make a copy of the new Token pool supply
<2> PICK

//Subtract the previous Token pool supply with the new token pool supply, making Tokens user received
SUB64

//Verify no overflow
VERIFY

//Remove L-BTC pool supply for constant
NIP

//Make a copy of L-BTC initial pool supply
<2> PICK

//Get tradable L-BTC supply from altstack, make one copy, and sand the copy to altstack
FROMALTSTACK DUP TOALTSTACK

//Add initial L-BTC pool supply with tradable user L-BTC supply to calculate the final L-BTC pool supply
ADD64

//Verify no overflow
VERIFY

SWAP

//Fetch slippage limit parsed from calldata
<8> ROLL

//Make a copy of the amount received
PICK

//Check slippage overflow
GREATERTHANOREQUAL64 2DROP <1> <1> EQUAL
```

```
//If the slippage is not overflowed:
IF

//Inspect settlement output 1 asset
<3+N> INSPECTOUTPUTASSET

//Verify the prefix(1)
<0x01> EQUALVERIFY

//Get Token asset ID
<13> PICK

//Verify that settlement output 1 holds the Token assetID(32)
EQUALVERIFY

//Inspect settlement output  1 amount and verify the prefix(1)
<3+N> INSPECTOUTPUTVALUE <0x01> EQUALVERIFY

//Verify that settlement output 1 holds the same value(8)
EQUALVERIFY

//Fetch old token pool supplies and drop
<3> ROLL <3> ROLL 2DROP

//If the slippage is overflow
ELSE

//Inspect settlement output 1 asset
<3+N> INSPECTOUTPUTASSET

//Verify the prefix(1)
<0x01> EQUALVERIFY

//Make a copy of Token asset ID and verify that settlement output 1 holds the L-BTC assetID(32)
<14> PICK EQUALVERIFY

//Drop overflown token received amount and the two new failed pool supply
DROP 2DROP

//Get tradable user L-BTC amount from altstack
FROMALTSTACK

//Inspect settlement output  1 amoun
<3+N> INSPECTOUTPUTVALUE

//Verify the prefix(1)
<0x01> EQUALVERIFY

//Verify that settlement output 1 holds the same value(8)
EQUALVERIFY

ENDIF

//Send top 5 stack elements to altstack for further higher index slot interpretations
TOALTSTACK TOALTSTACK TOALTSTACK TOALTSTACK TOALTSTACK

- - - - - - - - - - - - - - - - - - - - END CASE 1 : Swap L-BTC for Token - - - - - - - - - - - - - - - - - - -
```

4.4.2.3.2.4.2 Swap Token for L-BTC

A swap-in Token amount, a base fee 1200, an ordering fee varying from 1-2147483647, and a dust 650 sats are supplied in the commitment transaction. The pool transaction constrains settlement output 1 to the L-BTC swap-out amount based on the CPMM formula in a success case and to the initial Token swap-in amount in a slippage failure case.

Inputs	**Asset**	**Amount**
Commitment Input 1	L-BTC	Base Fee (1200) + Ordering Fee (1-2147483647) + Dust (650)
Commitment Input 2	Token	Swap In Amount
Outputs (Success)	**Asset**	**Amount**
Settlement Output 1	L-BTC	Swap Out Amount
Settlement Output 2	L-BTC	0 [OP_RETURN]
Outputs (Out of Slippage)	**Asset**	**Amount**
Settlement Output 1	Token	Swap In Amount
Settlement Output 2	L-BTC	0 [OP_RETURN]

Based on parameters;

I. $sl$is a state of L-BTC pool supply held by the main covenant
II. $st$is a state of Token pool supply held by the Token holder covenant
III. si is the swap-in Token amount that the user commits
IV. lf is the liquidity provider fee charged from the swap-in amount
V. so is the swap-out L-BTC amount that the user receives
VI. s is the L-BTC slippage limit

1. A %0.2 LP fee is calculated $lf = \frac{si}{500}$
2. New tradable swap-in amount $si' = si - lf$
3. Token pool supply for constant $st' = st - si$
4. Token pool supply for constant (downgraded 4194304x) $st'' = \frac{st'}{4194304}$
5. L-BTC pool supply downgraded (downgraded 32x) $sl' = \frac{sl}{32}$
6. Tradable Token swap-in amount downgraded (downgraded 4194304x) $si'' = \frac{si'}{4194304}$
7. The new state of L-BTC pool supply $sl'' = \frac{st''*sl'}{si''}$
8. Final L-BTC pool supply (magnified 32x) $sl''' = sl'' * 32$
9. L-BTC user received $so = sl - sl'''$
10. Final Token pool supply $st''' = st + si$
11. Success if $s > so$

```
- - - - - - - - - - - - - - - - - - - BEGIN CASE 2 : Swap Token for L-BTC - - - - - - - - - - - - - - - - - - -

//Fetch commitment input 1 tradable L-BTC supply
<5> ROLL

//Push a dust value 650 onto stack and verify that commitment input 1 holds dust L-BTC
<650> SCIPTNUMTOLE64 EQUALVERIFY

//Inspect commitment input 2 asset and verify the prefix(1)
<4+N> INSPECTINPUTASSET <0x01> EQUALVERIFY

//Get Token asset ID and verify that commitment input 2 holds the Token assetID(32)
<11> PICK EQUALVERIFY

//Inspect commitment input 2 amount and verify the prefix(1)
<4+N> INSPECTINPUTVALUE  <0x01> EQUALVERIFY

//Make two copies of Token swap-in amount and send one to altstack
DUP DUP TOALTSTACK

//Calculate LP fee %0.2
<500> SCIPTNUMTOLE64 DIV64

//Verify no overflow
VERIFY

//Remove remainder
NIP

//Subtract LP fees from the Token amount and verify no overflow
SUB64 VERIFY

//Make a copy of the total Token pool supply
<2> PICK

//Add existing pool supply with new pool supply to calculate new L-BTC pool supply
ADD64 2DROP

//Verify no overflow
VERIFY

//Make a copy of the total L-BTC pool supply and downgrade precision (Minimum unit to 32 sats)
<1> PICK <32> SCIPTNUMTOLE64 DIV64

//Verify no overflow
VERIFY

//Make a copy of total Token pool supply downgrade precision. (Minimum unit to 4194304)
<3> PICK <4194304> SCIPTNUMTOLE64 DIV64

//Verify no overflow
VERIFY

//Multiply the two downgraded pool supply to calculate the constant value
MUL64

//Verify no overflow
VERIFY
```

```
//Make a copy of Token pool supply for constant and downgrade precision (Minimum unit to 4194304)
<1> PICK <4194304> SCIPTNUMTOLE64 DIV64

//Verify no overflow
VERIFY
//Divide constant with Token pool supply to calculate the new L-BTC pool supply
DIV64

//Verify no overflow
VERIFY

//Magnify the new L-BTC pool supply
<32> SCIPTNUMTOLE64 MUL64

//Verify no overflow
VERIFY

SWAP

//Make a copy of the previous pool L-BTC supply
<2> PICK

//Make a copy of the new pool L-BTC supply
<2> PICK

//Subtract the pool supply states to calculate the L-BTC user received
SUB64

//Verify no overflow
VERIFY

//Remove Token pool supply for constant
NIP

//Make a copy of Token initial pool supply
<3> PICK

//Get tradable Token supply from altstack, make one copy, and send the copy to altstack
FROMALTSTACK DUP TOALTSTACK

//Add initial Token pool supply with tradable user Token supply to calculate the final Token pool supply
ADD64

//Verify no overflow
VERIFY

SWAP

//Fetch slippage limit parsed from calldata
<8> ROLL

//Make a copy of the amount received
<1> PICK

//Check slippage overflow
GREATERTHANOREQUAL64 2DROP <1> <1> EQUAL
```

```
//If the slippage is not overflow
IF

//Inspect settlement output 1 asset and verify the prefix(1)
<3+N> INSPECTOUTPUTASSET <0x01> EQUALVERIFY

//Make a copy of Token asset ID and
<14> PICK

//Verify that settlement output 1 holds the L-BTC assetID(32)
EQUALVERIFY

//Inspect settlement output  1 amount
<3+N> INSPECTOUTPUTVALUE

//Verify the prefix(1)
<0x01> EQUALVERIFY

//Verify that settlement output 1 holds the same value(8)
EQUALVERIFY

//Fetch old token pool supplies and drop
<3> ROLL <3> ROLL 2DROP SWAP

//If the slippage is overflow
ELSE

//Inspect settlement output 1 asset
<3+N> INSPECTOUTPUTASSET

//Verify the prefix(1)
<0x01> EQUALVERIFY

//Get Token asset ID and verify that settlement output 1 holds the L-BTC assetID(32)
<13> PICK EQUALVERIFY

//Drop overflown L-BTC received amount and the two new failed pool supply
DROP 2DROP

//Get tradable user Token amount from altstack
FROMALTSTACK

//Inspect settlement output  1 amount
<3+N> INSPECTOUTPUTVALUE

//Verify the prefix(1)
<0x01> EQUALVERIFY

//Verify that settlement output 1 holds the same value(8)
EQUALVERIFY

ENDIF

//Send top 5 stack elements to altstack for further higher index slot interpretations
TOALTSTACK TOALTSTACK TOALTSTACK TOALTSTACK TOALTSTACK

- - - - - - - - - - - - - - - - - - - END CASE 2 : Swap Token for L-BTC - - - - - - - - - - - - - - - - - - -
```

4.4.2.3.2.4.3 Add Liquidity

An L-BTC supply, a Token supply, a base fee of 1200, an ordering fee varying from 1-2147483647, and a dust 650 sats are supplied in the commitment transaction. The pool transaction constrains settlement output 1 to give the user liquidity shares (LP tokens) proportional to liquidity provided.

Inputs	Asset	Amount
Commitment Input 1	L-BTC	Supply + Base Fee (1200) + Ordering Fee (1-2147483647) + Dust (650)
Commitment Input 2	Token	Supply
Outputs	**Asset**	**Amount**
Settlement Output 1	LP	Liquidity Share
Settlement Output 2	L-BTC	0 [OP_RETURN]

Based on parameters;

I. pl is a state of pool L-BTC supply held by the main covenant
II. pt is a state of pool Token supply held by the Token holder covenant
III. $plp$is a state of pool LP supply held by the LP holder covenant
IV. $mlp$is the max pool LP supply, set to 5000000000000000
V. $olp$is a state of occupied LP supply
VI. ul is the user-added L-BTC supply
VII. ut is the user-added Token supply
VIII. ulp is the LP tokens user receives
IX. rl is the L-BTC addition ratio
X. rt is the Token addition ratio

1. Existing pool L-BTC supply magnified 1000x $pl' = pl * 1000$
2. Magnified pool L-BTC supply divided to added user L-BTC supply $rl = \frac{pl'}{ul}$
3. The current pool Token supply magnified 1000x $pt' = pt * 1000$
4. Magnified pool Token supply divided to added user Token supply $rl = \frac{pt'}{ut}$
5. Total occupied LP tokens $olp = mlp - plp$
6. LP tokens user received $ulp = \frac{olp}{max(rl, rt)} * 1000$
7. New state of occupied LP supply $olp' = olp + ulp$
8. New state of pool LP supply $plp' = plp' - ulp$

```
- - - - - - - - - - - - - - - - - - - - - - - - - BEGIN CASE 3 : Add Liquidity - - - - - - - - - - - - - - - - - - - - - -

//Push 5000000000000000 max LP pool supply
<5000000000000000>

//Make a copy of the current LP pool supply
<3> PICK

//Subtract max LP pool supply with current LP pool supply to get the current LP used
SUB64

//Make a copy of the current L-BTC pool supply
<1> PICK

//Push int 1000 and convert to LE64
<1000> SCIPTNUMTOLE64

//Magnify current L-BTC pool supply 1000 times
MUL64

//Verify no overflow
VERIFY

//Fetch available user L-BTC supply
<7> ROLL

//Push int 650 and convert to LE64
<650> SCIPTNUMTOLE64

//Subtract 650 from available user L-BTC supply, make one copy and send the copy to altstack
SUB64 DUP TOALTSTACK

//Divide magnified L-BTC supply with available user L-BTC supply to calculate the ratio
DIV64

//Verify no overflow
VERIFY

//Make a copy of the current Token pool supply
<3> PICK

//Push int 1000 and convert to LE64
<1000> SCIPTNUMTOLE64

//Magnify current Token pool supply 10000 times and verify no overflow
VERIFY

//Inspect commitment input 2 asset
<4+N> INSPECTINPUTASSET

//Verify the prefix(1)
<0x01> EQUALVERIFY

//Make a copy of Token asset ID
<14> PICK
```

```
//Verify that commitment input 2 holds the Token assetID(32)
EQUALVERIFY

//Inspect commitment input 2 amount
<4+N> INSPECTOUTPUTVALUE

//Verify the prefix(1)
<0x01> EQUALVERIFY

//Make a copy of the commitment input 2 amount and send the copy to altstack
DUP TOALTSTACK

//Divide magnified Token pool supply with available user Token supply to calculate the second ratio
DIV64

//Verify no overflow
VERIFY

//Duplicate the two ratios
2DUP

//Compare the two ratios
GREATERTHANOREQUAL64

IF

//Drop the smaller ratio (last stack element)
DROP
ELSE

//Drop the smaller ratio (second to last stack element)
NIP

ENDIF

//Divide current LP with ratio to calculate LP amount user received
DIV64

//Verify no overflow
VERIFY

//Push int 1000 and convert to LE64
<1000> SCIPTNUMTOLE64

//Apply the same 1000x magnification this side to get LP tokens user received
MUL64

//Verify no overflow
VERIFY

//Inspect LP token asset ID
<2> INSPECTOUTPUTASSET

//Verify the prefix(1)
<0x01> EQUALVERIFY

//Inspect settlement output 1 asset ID
<3+N> INSPECTOUTPUTASSET
```

```
//Verify the prefix(1)
<0x01> EQUALVERIFY

//Constrain settlement output 1 to LP token asset ID
EQUALVERIFY

//Make a copy of LP tokens user received
DUP

//Inspect settlement output  1 amount
<4+N> INSPECTOUTPUTVALUE

//Verify the prefix(1)
<0x01> EQUALVERIFY

//Make sure user receives deserved LP tokens
EQUALVERIFY

//Fetch previous LP pool supply
<3> ROLL

SWAP

//Update new LP pool supply and verify no overflow
SUB64 VERIFY

//Fetch previous Token pool supply
<2> ROLL

//Get Token user supply from altstack
FROMALTSTACK

//Update new Token pool supply
ADD64

//Verify no overflow
VERIFY

//Fetch previous L-BTC pool supply
<2> ROLL

//Get user L-BTC supply from altstack
FROMALTSTACK

//Update new L-BTC pool supply
ADD64

//Verify no overflow
VERIFY

//Send top 5 stack elements to altstack for further higher index slot interpretations
TOALTSTACK TOALTSTACK TOALTSTACK TOALTSTACK TOALTSTACK

//Drop slippage limit
DROP

- - - - - - - - - - - - - - - - - - - - - - - - - END CASE 3 : Add Liquidity - - - - - - - - - - - - - - - - - - - - - - -
```

4.4.2.3.2.4.4 Remove Liquidity

A base fee of 1200, an ordering fee varying from 1-2147483647, a dust 650 sats, and LP tokens are supplied in the commitment transaction. The pool transaction redeems LP tokens and constrains settlement output 1 and 2 to give the user their proportional liquidity.

Inputs	**Asset**	**Amount**
Commitment Input 1	L-BTC	Base Fee (1200) + Ordering Fee (1-2147483647) + Dust (650)
Commitment Input 2	LP	Liquidity Share
Outputs	**Asset**	**Amount**
Settlement Output 1	L-BTC	Supply
Settlement Output 2	Token	Supply

Based on parameters;

I. pl is a state of pool L-BTC supply held by the main covenant

II. pt is a state of pool Token supply held by the Token holder covenant

III. $plp$is a state of pool LP supply held by the LP holder covenant

IV. $mlp$is the max pool LP supply, set to 5000000000000000

V. $olp$is a state of occupied LP supply

VI. ulp is the amount of LP tokens the user redeems

VII. rlp is the amount of LP redemption ratio

VIII. ul is the L-BTC amount user receives

IX. ut is the Token amount user receives

1. Total occupied LP tokens olp = $mlp - plp$
2. Total occupied LP tokens is magnified 1000x olp' = $olp * 1000$
3. LP redemption ratio rlp = $\frac{olp'}{ulp}$
4. L-BTC amount user receives ul = $\frac{pl}{rlp} * 1000$
5. Token amount the user receives ut = $\frac{pt}{rlp} * 1000$
6. New occupied LP token supply $olp'' = olp - ulp$
7. New pool LP token supply $plp' = plp + ulp$
8. New pool L-BTC supply $pl' = pl - ul$
9. New pool Token supply $pt' = pt - ut$

```
- - - - - - - - - - - - - - - - - - - - - - - BEGIN CASE 4 : Remove Liquidity - - - - - - - - - - - - - - - - - - - -
//Fetch commitment input 1 tradable L-BTC supply
<5> ROLL
//Push a dust value 650 onto stack
<650> SCIPTNUMTOLE64
//Verify that commitment input 1 holds dust L-BTC
EQUALVERIFY
//Inspect commitment input 2 asset and verify the prefix(1)
<4+N> INSPECTINPUTASSET <0x01> EQUALVERIFY
//Get LP token asset ID and verify the prefix(1)
<2> INSPECTOUTPUTASSET <0x01> EQUALVERIFY
//Verify that commitment input 2 holds the LP assetID(32)
EQUALVERIFY
//Push 5000000000000000 max LP pool supply
<5000000000000000>
//Make a copy of the current LP pool supply
<3> PICK
//Subtract max LP pool supply with current LP pool supply to get the current LP used
SUB64
//Verify no overflow
VERIFY
//Push int 1000 and convert to LE64
<1000> SCIPTNUMTOLE64
//Magnify current lp pool supply 10000 times and verify no overflow
MUL64 VERIFY
//Inspect commitment input 2 amount and verify the prefix(1)
<4+N> INSPECTINPUTVALUE <0x01> EQUALVERIFY
//Make one copy and send to altstack
DUP TOALTSTACK
//Calculate the LP ratio and verify no overflow
DIV64 VERIFY
//Make a copy of the current L-BTC pool supply
<1> PICK
//Make a copy of LP ratio
<1> PICK
//Divide current L-BTC pool supply with LP ratio to calculate L-BTC user received and verify no overflow
DIV64 VERIFY
//Push int 1000 and convert to LE64
<1000> SCIPTNUMTOLE64
```

```
//Apply the same 1000x magnification on this side and verify no overflow
MUL64 VERIFY

//Make one copy and send to altstack
DUP TOALTSTACK

//Inspect settlement output 1 asset
<3+N> INSPECTOUTPUTASSET

//Verify the prefix(1)
<0x01> EQUALVERIFY

//Get L-BTC asset ID
<14> PICK

//Verify that settlement output 1 holds the L-BTC assetID(32)
EQUALVERIFY

//Inspect settlement output 1 amount
<3+N> INSPECTOUTPUTVALUE

//Verify the prefix(1)
<0x01> EQUALVERIFY

//Make sure user gets deserved L-BTC amount
EQUALVERIFY

//Make a copy of the current L-BTC pool supply
<2> PICK

SWAP

//Divide current Token pool supply with LP ratio to calculate Token amount user received
DIV64

//Verify no overflow
VERIFY

//Push int 1000 and convert to LE64
<1000> SCIPTNUMTOLE64

//Apply the same 1000x magnification on this side and verify no overflow
MUL64 VERIFY

//Make one copy and send to altstack
DUP TOALTSTACK

//Inspect settlement output 2 asset and verify the prefix(1)
<4+N> INSPECTOUTPUTASSET

//Get Token asset ID
<12> PICK

//Verify that settlement output 1 holds the Token assetID(32)
EQUALVERIFY
```

```
//Inspect settlement output 2 amount
<4+N> INSPECTOUTPUTVALUE

//Verify the prefix(1)
<0x01> EQUALVERIFY

//Make sure user gets deserved Token amount
EQUALVERIFY

//Get Token user received
FROMALTSTACK

//Get L-BTC user received
FROMALTSTACK

//Get LP tokens user sent
FROMALTSTACK

//Fetch previous LP pool supply
<5> ROLL

//Update new LP pool supply and verify no overflow
ADD64 VERIFY

//Fetch previous Token pool supply
<4> ROLL

//Fetch Token user received
<3> ROLL

//Update new Token pool supply  and verify no overflow
SUB64 VERIFY

//Fetch previous L-BTC pool supply
<3> ROLL

//Fetch L-BTC user received
<3> ROLL

//Update new L-BTC pool supply and verify no overflow
SUB64 VERIFY

//Inspect settlement output 1 witness program and verify v0 witnes version
//<N+3> INSPECTOUTPUTSCRIPTPUBKEY <0> EQUALVERIFY

//Inspect settlement output 2 witness program and verify v0 witnes version
<N+4> INSPECTOUTPUTSCRIPTPUBKEY <0> EQUALVERIFY

//Make settlement output 1 and settlement output 2 witness programs are equal
EQUALVERIFY

//Send top 5 stack elements to altstack for further higher index slot interpretations
TOALTSTACK TOALTSTACK TOALTSTACK TOALTSTACK TOALTSTACK

//Drop slippage limit
DROP

- - - - - - - - - - - - - - - - - - - - - - - END CASE 4 : Remove Liquidity - - - - - - - - - - - - - - - - - - -
```

4.4.2.3.3 Witness Script Footer

The witness script footer is a fixed-size witness script making the last part of the maing covenant witness script.

```
- - - - - - - - - - - - - - - - - - - - - - - - - - BEGIN SCRIPT FOOTER - - - - - - - - - - - - - - - - - - - - - - -

//Get latest-index leaf tx fees
FROMALTSTACK

//Get total leaf tx fees
FROMALTSTACK

//Get latest-state pool LP supply
FROMALTSTACK

//Get latest-state pool Token supply
FROMALTSTACK

//Get latest-state pool L-BTC supply
FROMALTSTACK

//Inspect L-BTC amount the main covenant holds and verify the prefix(1)
<3> INSPECTINPUTVALUE <0x01> EQUALVERIFY

//Constrain main covenant L-BTC pool amount to latest state L-BTC supply
EQUALVERIFY

//Inspect Token amount the main covenant holds and verify the prefix(1)
<1> INSPECTINPUTVALUE <0x01> EQUALVERIFY

//Constraint Token covenant L-BTC pool amount to latest state Token supply
EQUALVERIFY

//Inspect LP amount the LP holder covenant holds and verify the prefix(1)
<1> INSPECTINPUTVALUE <0x01> EQUALVERIFY

//Constrain LP covenant LP pool amount to latest state LP supply
EQUALVERIFY

//Convert total fees paid to LE64
SCIPTNUMTOLE64

//Calculate fees paid as tx fees and verify no overflow
INSPECTNUMINPUTS <4> SUB VERIFY

//Multiply the number of inputs with  total fees paid and verify no overflow
MUL64 VERIFY

//Divide the result with the max  number of inputs to calculate the total tx fees paid
<64> SCIPTNUMTOLE64 DIV64

//Verify no overflow
VERIFY

//Get the fee output index
INSPECTNUMOUTPUTS <8> <1> SUB
```

```
/Make a copy of fee output index
DUP
//Inspect fee output value and verify the prefix(1)
INSPECTOUTPUTVALUE<0x01> EQUALVERIFY
//Constrain tx fees
EQUALVERIFY
//Get the service fee output index
INSPECTNUMOUTPUTS <8> <2> SUB
//Inspect service fee output asset and verify the prefix(1)
INSPECTOUTPUTASSET <0x01> EQUALVERIFY
//Make a copy of the L-BTC asset ID
<8> PICK
//Constrain service fee output asset ID to L-BTC
EQUALVERIFY
//Inspect service fee output witness program
INSPECTOUTPUTSCRIPTPUBKEY
//Constrain to witness version 0
<0> EQUALVERIFY
//Constrain witness program 1ef08948ed902a517d90ef6955c66c183a444afd
<0x1ef08948ed902a517d90ef6955c66c183a444afd> EQUALVERIFY
//Empty stack
2DROP 2DROP 2DROP <1>

- - - - - - - - - - - - - - - - - - - - - - - - - - END SCRIPT FOOTER - - - - - - - - - - - - - - - - - - - - - - -
```

5. Precision Downgrade

Applying arithmetic operations over two 64-bit signed asset values to calculate a constant value overflows the operation. To avoid this, inspected L-BTC and Token amounts are downgraded to fit in 32-bit form. A 32-bit signed int could store max 21.47483647 L-BTC value and 21.47483647 USDT. By downgrading L-BTC value 32 times and Token value 4194304 times, a Bitmatrix pool can accommodate up to 687.19476704 L-BTC and 90,071,992 USDT. The tradeoff counts a single L-BTC unit for 32 sats and a minimum USDT unit for 4 cents. More than one AMM pool for the same pair can be created if one pool hits the max liquidity limit (687 L-BTC).