LOL: Layer Of Liquidity
A Sovereign Automated Market Maker Protocol on Bitcoin

lamachina (monsieur.foo)
`tx.lol`

OPI-001: Standards Track

June 2025
Version 1.0

Abstract

We present LOL (Layer Of Liquidity), a decentralized asset exchange protocol architected for the Universal BRC-20 extension on Bitcoin. LOL introduces a virtual Automated Market Maker (AMM) whose state is derived deterministically from OP_RETURN messages, obviating the need for custodial bridges or complex Layer 2 constructions. The protocol defines two atomic operations: `init` for providing time-locked liquidity commitments and `exe` for swap execution, governed by a "Total Lock-up Model" to foster deep, stable markets. By leveraging Bitcoin's UTXO model as an immutable log of intent and maintaining execution state off-chain through deterministic computation, LOL establishes a trustless, capital-efficient liquidity layer. This work formalizes the mathematical foundations of constant product market making on Bitcoin, analyzes the economic incentives of time-locked liquidity provision, and provides a complete specification for indexer implementation. LOL serves as the foundational, composable liquidity primitive for a sovereign financial system on Bitcoin's base layer.

Keywords: Bitcoin, BRC-20, Universal Protocol, Automated Market Maker, Liquidity, DeFi, OP_RETURN, Time-Locked Commitments, Virtual State Machine

Contents

1 Introduction **4**
1.1 The Problem: Liquidity Fragmentation and Mercenary Capital 4
1.2 Contributions 4
1.3 Design Philosophy 4

2 Fundamental Concepts **6**
2.1 The Virtual AMM Model 6
2.2 UTXO-Based Identity 6
2.3 Canonical Pool Identification 6

3 Operation Specifications **7**
3.1 Transaction Structure 7
3.2 Operation 1: Liquidity Provision (`init`) 7
3.2.1 Payload Specification 7
3.2.2 Semantic Interpretation 8
3.2.3 State Transition 8
3.3 Operation 2: Swap Execution (`exe`) 8
3.3.1 Payload Specification 8
3.3.2 Constant Product Invariant 9
3.3.3 Swap Formula Derivation 9
3.3.4 Slippage and Partial Fills 9
3.3.5 State Transition 9

4 The Total Lock-up Model **10**
4.1 Economic Rationale 10
4.2 The Lock Mechanism 10
4.3 Fee Distribution Model 10
4.4 Economic Analysis 10

5 Indexer Consensus Rules **11**
5.1 Chronological Processing 11
5.2 Validation Rules 11
5.3 Partial Fill Algorithm 12
5.4 Lock Expiration Processing 13

6 Mathematical Foundations **14**
6.1 Liquidity Share Accounting 14
6.2 Impermanent Loss Under Lock 14
6.3 Price Impact 14

7 Security Analysis **15**
7.1 Threat Model 15
7.2 Security Properties 15
7.3 Protection Against Front-Running 15
7.4 Double-Spending Prevention 15
7.5 Sybil Resistance 15

8 Implementation Reference **16**
8.1 State Data Structures 16
8.2 Operation Handlers 17
8.3 Complete State Transition 19

9 Economic Incentives and Game Theory **20**
9.1 Liquidity Provider Strategy 20
9.2 Optimal Lock Period 20
9.3 Protocol-Level Incentive Alignment 20

10 Comparative Analysis **21**

11 Future Work **22**

12 Conclusion **23**

A Transaction Examples **24**
A.1 Example 1: Initial Liquidity Provision 24
A.2 Example 2: Swap Execution 24

B Formal Notation Reference **25**

1 Introduction

Expressive finance on Bitcoin has historically been constrained by a false choice between custodial risk, Layer 2 complexity, and smart contract programming overhead. The emergence of token protocols like BRC-20 and its Universal extension has demonstrated that sophisticated economic primitives can be constructed through simple, deterministic off-chain computation over Bitcoin's native UTXO structure.

LOL (Layer Of Liquidity) represents the next logical evolution: a fully decentralized Automated Market Maker (AMM) that operates entirely through OP_RETURN messages, with no custodians, no bridges, and no additional consensus layers. The protocol achieves this through a fundamental architectural insight: Bitcoin provides the perfect *immutable log of intent*, while execution logic can live off-chain as a deterministic state machine that any honest participant can verify.

1.1 The Problem: Liquidity Fragmentation and Mercenary Capital

Existing approaches to decentralized exchange on Bitcoin face three critical challenges:

1. **Custodial Risk:** Wrapped tokens and centralized bridges introduce single points of failure, contradicting Bitcoin's sovereignty guarantee.
2. **Layer 2 Complexity:** Sidechains and state channels require additional security assumptions, complex peg mechanisms, and often sacrifice the transparency of Layer 1.
3. **Mercenary Liquidity:** Traditional AMMs suffer from transient, profit-seeking liquidity providers who exit during volatility, precisely when liquidity is most needed.

LOL addresses these challenges through principled design choices rooted in Bitcoin's native capabilities.

1.2 Contributions

This work makes the following contributions:

1. **Virtual AMM Architecture:** A formal specification for a deterministic, off-chain AMM state machine driven by on-chain OP_RETURN messages, eliminating the need for custodial or Layer 2 infrastructure.
2. **Total Lock-up Model:** An economic mechanism requiring time-locked liquidity commitments, creating aligned incentives and deep, stable markets.
3. **Canonical Pool Ordering:** A protocol-level solution to liquidity fragmentation through deterministic pool identification.
4. **Mathematical Formalization:** Complete derivation of constant product invariants, slippage calculations, and liquidity share accounting adapted for time-locked positions.
5. **Indexer Specification:** A comprehensive reference implementation defining consensus rules for state transitions, partial fills, and lock expiration.

1.3 Design Philosophy

LOL is guided by three core principles:

1. **Bifurcated Security:** Bitcoin provides the immutable, ordered log of operations. Security emerges from the mathematical certainty that any honest node computing over this log arrives at identical state.

2. **Virtual State as Efficiency:** State maintenance is a computational concern, not a consensus concern. Off-chain state machines are infinitely more agile and cost-effective than on-chain alternatives.

3. **Commitment over Opportunism:** The mandatory `lock` parameter transforms liquidity provision from a speculative activity into a deliberate capital commitment, aligning provider incentives with protocol health.

2 Fundamental Concepts

2.1 The Virtual AMM Model

LOL operates as a *virtual state machine* whose transitions are triggered by Bitcoin transactions containing OP_RETURN messages. This architecture achieves several critical properties:

Definition 1 (Virtual State Machine)**.** *A virtual state machine $M = (S, \Sigma, \delta, s_0)$ where:*

- *S is the set of possible protocol states*
- *Σ is the alphabet of valid operations (encoded in OP_RETURN)*
- *$\delta : S \times \Sigma \to S$ is the deterministic state transition function*
- *s_0 is the initial state (empty pools, zero balances)*

The key insight is that δ is *completely deterministic* and *publicly specified*. Any participant can independently compute the current state s_t by applying δ sequentially to the ordered sequence of Bitcoin transactions from genesis to block height t.

2.2 UTXO-Based Identity

Identity in LOL is tied directly to Bitcoin's UTXO model, eliminating the need for separate authentication systems.

Definition 2 (Operator Identity)**.** *For a transaction T containing a `swap` operation, the operator identity $\mathcal{I}(T)$ is defined as the address controlling $T.inputs[0]$, the first input UTXO.*

This design provides several security properties:

- **Sybil Resistance:** Creating distinct identities requires controlling separate UTXOs with real economic cost.
- **Signature Verification:** Bitcoin's signature verification provides authentication without additional cryptographic infrastructure.
- **Temporal Ordering:** UTXO ordering provides a deterministic resolution mechanism for simultaneous operations.

2.3 Canonical Pool Identification

To prevent liquidity fragmentation, LOL enforces a canonical ordering of token pairs.

Definition 3 (Canonical Pool ID)**.** *For tokens A and B, the canonical pool identifier is:*

$$PoolID(A, B) = \begin{cases} \texttt{A-B} & \text{if } A \prec_{lex} B \\ \texttt{B-A} & \text{otherwise} \end{cases} \tag{1}$$

where $\prec_{lex}$ denotes lexicographic (alphabetical) ordering.

This ensures that all liquidity for a given pair aggregates into a single pool, regardless of the direction in which users specify the swap.

3 Operation Specifications

3.1 Transaction Structure

All LOL operations follow the Universal BRC-20 transaction structure with specific requirements:

Component	Description
Input[0]	The first input UTXO. Its controlling address is the unique, verifiable identifier of the operator.
Input[n+]	Additional inputs as needed for network fees.
Output[0]	An OP_RETURN output containing the JSON payload (`{"p": "brc-20", "op": "swap", ...}`) that dictates the state transition.
Output[n]	Optional change output and other standard transaction components.

Table 1: LOL Transaction Structure

Critical Requirements:

- Input ordering MUST be preserved; Input[0] defines operator identity.
- Output[0] MUST be an OP_RETURN with value 0 sats.
- The JSON payload is UTF-8 encoded, then hex-encoded into the OP_RETURN script.

3.2 Operation 1: Liquidity Provision (`init`)

The `init` operation creates a time-locked liquidity position in a pool.

3.2.1 Payload Specification

```
{
  "p": "brc-20",
  "op": "swap",
  "init": "lol,wtf",
  "amt": "101000",
  "lock": "101"
}
```

Field	Type	Description
`p`	String	Protocol identifier; must be `"brc-20"`.
`op`	String	Operation type; must be `"swap"`.
`init`	String	The token pair `<ticker_provided>,<ticker_sought>`, defining the direction of provision.
`amt`	String	The amount of `ticker_provided` being locked.
`lock`	String	The mandatory lock-up duration in Bitcoin blocks (≥ 1). The position remains locked for this period.

Table 2: Liquidity Provision Payload Fields

3.2.2 Semantic Interpretation

Let $P = \text{PoolID}(A, B)$ be the canonical pool for tokens A and B. An `init` operation with parameters $(A \to B, x, \ell)$ represents:

- A commitment to provide x units of token A to pool P
- A lock duration of ℓ blocks from the confirmation block h_0
- The position cannot be withdrawn until block $h_0 + \ell$
- The operator receives a proportional share of future trading fees during the lock period

3.2.3 State Transition

Let s denote the current state and $\mathcal{O}$ the operator. The `init` operation transitions state as follows:

$$\text{Balance}_A(\mathcal{O}) \leftarrow \text{Balance}_A(\mathcal{O}) - x \tag{2}$$
$$\text{LockedBalance}_P(\mathcal{O}, A) \leftarrow \text{LockedBalance}_P(\mathcal{O}, A) + x \tag{3}$$
$$\text{UnlockHeight}_P(\mathcal{O}) \leftarrow h_0 + \ell \tag{4}$$
$$\text{Reserve}_P(A) \leftarrow \text{Reserve}_P(A) + x \tag{5}$$

where $\text{LockedBalance}_P(\mathcal{O}, A)$ represents the operator's locked position in pool P for token A.

3.3 Operation 2: Swap Execution (`exe`)

The `exe` operation executes an asset exchange against an active liquidity pool.

3.3.1 Payload Specification

```
{
  "p": "brc-20",
  "op": "swap",
  "exe": "wtf,lol",
  "amt": "21",
  "slip": "0.5"
}
```

Field	Type	Description
`p`	String	Protocol identifier; must be `"brc-20"`.
`op`	String	Operation type; must be `"swap"`.
`exe`	String	The swap direction `<ticker_given>,<ticker_received>`.
`amt`	String	The amount of `ticker_given` offered for the swap.
`slip`	String	Maximum slippage tolerance, as a decimal percentage string ($0 < \text{slip} < 100$). Example: `"0.5"`.

Table 3: Swap Execution Payload Fields

3.3.2 Constant Product Invariant

LOL implements a constant product market maker (CPMM) with invariant:

$$k = R_A \cdot R_B \tag{6}$$

where R_A and R_B are the reserve amounts of tokens A and B in the pool, and k is the constant product.

3.3.3 Swap Formula Derivation

Consider a swap offering Δ_A units of token A to receive Δ_B units of token B. The invariant must hold:

$$(R_A + \Delta_A) \cdot (R_B - \Delta_B) = k = R_A \cdot R_B \tag{7}$$

Solving for Δ_B:

$$R_A \cdot R_B = (R_A + \Delta_A)(R_B - \Delta_B) \tag{8}$$
$$R_A \cdot R_B = R_A R_B - R_A \Delta_B + \Delta_A R_B - \Delta_A \Delta_B \tag{9}$$
$$0 = -R_A \Delta_B + \Delta_A R_B - \Delta_A \Delta_B \tag{10}$$
$$R_A \Delta_B = \Delta_A R_B - \Delta_A \Delta_B \tag{11}$$
$$R_A \Delta_B = \Delta_A (R_B - \Delta_B) \tag{12}$$
$$\Delta_B = \frac{\Delta_A \cdot R_B}{R_A + \Delta_A} \tag{13}$$

This is the fundamental swap formula for constant product AMMs.

3.3.4 Slippage and Partial Fills

The slippage σ represents the maximum acceptable deviation from the expected exchange rate. Given an expected rate $r_0 = R_B/R_A$ and actual received amount Δ_B, the slippage is:

$$\sigma = \left| \frac{r_0 \cdot \Delta_A - \Delta_B}{r_0 \cdot \Delta_A} \right| \times 100\% \tag{14}$$

Partial Fill Mechanism: If the requested swap Δ_A would exceed the slippage tolerance $\sigma_{\max}$, the protocol computes the maximum fillable amount Δ_A^* such that:

$$\sigma(\Delta_A^*) = \sigma_{\max} \tag{15}$$

This ensures that trades are always executed to the maximum extent possible within the user's risk parameters, preventing total rejection due to slippage.

3.3.5 State Transition

For a swap of Δ_A^* units of A receiving Δ_B units of B:

$$\text{Balance}_A(\mathcal{O}) \leftarrow \text{Balance}_A(\mathcal{O}) - \Delta_A^* \tag{16}$$
$$\text{Balance}_B(\mathcal{O}) \leftarrow \text{Balance}_B(\mathcal{O}) + \Delta_B \tag{17}$$
$$\text{Reserve}_P(A) \leftarrow \text{Reserve}_P(A) + \Delta_A^* \tag{18}$$
$$\text{Reserve}_P(B) \leftarrow \text{Reserve}_P(B) - \Delta_B \tag{19}$$
$$\text{AccumulatedFees}_P \leftarrow \text{AccumulatedFees}_P + f(\Delta_A^*) \tag{20}$$

4 The Total Lock-up Model

The Total Lock-up Model represents a fundamental departure from traditional AMM liquidity provision, addressing the systemic instability caused by mercenary capital.

4.1 Economic Rationale

Traditional AMMs suffer from a critical flaw: liquidity providers can withdraw at any time, creating a tragedy of the commons where rational actors exit during volatility, precisely when liquidity is most needed. This produces:

- **Pro-cyclical Instability:** Liquidity depth correlates positively with market calm and negatively with volatility.
- **Impermanent Loss Avoidance:** Providers exit before significant price movements, defeating the purpose of an AMM.

4.2 The Lock Mechanism

The `lock` parameter mandates a minimum commitment period measured in Bitcoin blocks:

Definition 4 (Lock Period)**.** *A liquidity position with lock parameter ℓ confirmed at block height h_0 cannot be withdrawn before block height $h_{unlock} = h_0 + \ell$.*

This creates several beneficial properties:

1. **Aligned Incentives:** Providers are economically invested in the pool's long-term success.
2. **Predictable Depth:** Market participants can observe the distribution of unlock heights, creating transparency around future liquidity availability.

4.3 Fee Distribution Model

During the lock period, liquidity providers earn a proportional share of trading fees:

$$\text{FeeShare}_{\mathcal{O}}(t) = \frac{\text{LockedBalance}_P(\mathcal{O}, A) + \text{LockedBalance}_P(\mathcal{O}, B)}{\text{Reserve}_P(A) + \text{Reserve}_P(B)} \cdot \text{AccumulatedFees}_P(t) \quad (21)$$

Fees accumulate continuously and are credited upon position unlock.

4.4 Economic Analysis

Theorem 1 (Nash Equilibrium Under Lock)**.** *Given a pool with total reserves R and n liquidity providers with lock periods $\{\ell_1, \ldots, \ell_n\}$, the Nash equilibrium strategy for a rational provider maximizing long-term expected returns is to choose $\ell_i > \bar{\ell}$, where $\bar{\ell}$ is the median lock period.*

Proof Sketch. Providers with longer lock periods:

1. Capture a larger share of trading fees over time due to persistent participation
2. Benefit from reduced competition as shorter-term providers exit
3. Face lower opportunity cost if pool growth is expected (early commitment advantage)

Therefore, in a growing protocol, longer commitments yield higher risk-adjusted returns, creating a self-reinforcing incentive structure. □

5 Indexer Consensus Rules

Indexers implementing LOL MUST enforce the following rules to maintain global state consistency.

5.1 Chronological Processing

1. **Block Ordering:** Operations are processed in strict block height order.
2. **Transaction Ordering:** Within a block, operations are processed according to transaction index order.
3. **Lock Expiration Priority:** At the beginning of each block's processing, all positions with UnlockHeight $\leq h$ are processed and funds returned to main balances before any new operations.

5.2 Validation Rules

For each operation, indexers MUST verify:

1. **Operator Identity:** Extract $\mathcal{O} = \mathcal{I}(T)$ from Input[0].
2. **Pool Existence:** For `exe` operations, verify that $\text{Reserve}_P(A) > 0$ and $\text{Reserve}_P(B) > 0$.
3. **Sufficient Balance:**
   - For `init`: $\text{Balance}_A(\mathcal{O}) \geq x$
   - For `exe`: $\text{Balance}_A(\mathcal{O}) \geq \Delta_A$
4. **Canonical Pool ID:** The pool identifier is computed as $\text{PoolID}(A, B)$ regardless of operation direction.
5. **Slippage Calculation:** For `exe`, compute Δ_A^* such that $\sigma(\Delta_A^*) \leq \sigma_{\max}$.
6. **Atomic State Transition:** All balance updates, reserve adjustments, and fee distributions occur atomically or not at all.

5.3 Partial Fill Algorithm

```python
def compute_partial_fill(reserve_A, reserve_B, delta_A_requested,
                         slip_max):
    """
    Compute maximum fillable amount within slippage tolerance.

    Returns: (delta_A_actual, delta_B_received)
    """
    # Binary search for maximum fillable amount
    left, right = 0, delta_A_requested
    epsilon = 1e-8  # Precision threshold

    while right - left > epsilon:
        mid = (left + right) / 2
        delta_B = (mid * reserve_B) / (reserve_A + mid)

        # Compute effective price and slippage
        expected_rate = reserve_B / reserve_A
        actual_rate = delta_B / mid
        slippage = abs((expected_rate - actual_rate) / expected_rate) *
            100

        if slippage <= slip_max:
            left = mid
        else:
            right = mid

    delta_A_actual = left
    delta_B_received = (delta_A_actual * reserve_B) / (reserve_A +
       delta_A_actual)

    return (delta_A_actual, delta_B_received)
```

5.4 Lock Expiration Processing

At the beginning of block h processing:

```
def process_lock_expirations(state, block_height):
    """
␣␣␣␣Process␣all␣expired␣locks␣at␣the␣start␣of␣block␣processing.
␣␣␣␣"""
    for pool in state.pools:
        for operator in pool.locked_positions:
            unlock_height = pool.unlock_heights[operator]

            if unlock_height <= block_height:
                # Compute operator's fee share
                fee_share = compute_fee_share(pool, operator)

                # Return locked tokens + fees to main balance
                for token in [pool.token_A, pool.token_B]:
                    locked_amt = pool.locked_balances[operator][token]
                    state.balances[operator][token] += locked_amt +
                       fee_share[token]

                    # Update pool reserves
                    pool.reserves[token] -= locked_amt

                # Clear locked position
                del pool.locked_positions[operator]

    return state
```

6 Mathematical Foundations

6.1 Liquidity Share Accounting

When a provider adds liquidity (x_A, x_B) to a pool with existing reserves (R_A, R_B), they receive shares proportional to their contribution:

$$s = \frac{x_A}{R_A} = \frac{x_B}{R_B} \tag{22}$$

However, in LOL's time-locked model, shares are not immediately fungible tokens but rather accounting entries tracking proportional ownership.

6.2 Impermanent Loss Under Lock

Impermanent loss is the opportunity cost of providing liquidity versus holding assets:

$$\text{IL} = \frac{2\sqrt{p}}{1+p} - 1 \tag{23}$$

where $p = \frac{P_t}{P_0}$ is the price ratio between current and initial prices.

The lock mechanism *does not eliminate* impermanent loss but rather prevents providers from avoiding it through strategic withdrawal, ensuring market stability.

6.3 Price Impact

For a swap of Δ_A units, the price impact is:

$$\text{Impact} = \frac{R_B}{R_A} - \frac{R_B - \Delta_B}{R_A + \Delta_A} = \frac{\Delta_A \cdot R_B}{(R_A + \Delta_A) \cdot R_A} \tag{24}$$

This quantifies the divergence from the spot price due to trade size.

7 Security Analysis

7.1 Threat Model

We consider the following adversarial actors:

1. **Malicious Operators:** Attempting to extract value through invalid operations or double-spending.
2. **Front-Running Miners:** Reordering transactions to capture MEV (Miner Extractable Value).
3. **Indexer Divergence:** Honest but buggy indexers computing inconsistent state.
4. **Economic Attackers:** Manipulating pool ratios for profit or denial-of-service.

7.2 Security Properties

Theorem 2 (State Consistency)**.** *Given two honest indexers I_1 and I_2 processing the same Bitcoin blockchain up to height h, the computed states $s_1(h) = s_2(h)$ are identical if both implement the canonical consensus rules.*

Proof. The state transition function δ is deterministic and operates over a totally ordered log (Bitcoin's blockchain). Given identical inputs (blockchain data) and identical transition logic (consensus rules), the outputs must be identical by determinism. □

7.3 Protection Against Front-Running

While LOL cannot eliminate MEV at the protocol level (miners can always reorder transactions), the partial fill mechanism provides user protection:

- Users specify maximum slippage tolerance $\sigma_{\max}$
- Fills occur at best possible rate within tolerance
- Even if front-run, users never receive worse execution than specified

7.4 Double-Spending Prevention

Bitcoin's UTXO model provides native protection against double-spending. An operator cannot execute multiple operations simultaneously because:

1. Each operation is tied to a specific UTXO (Input[0])
2. Once a UTXO is spent, it cannot be spent again
3. Conflicting transactions are resolved by Bitcoin's consensus mechanism

7.5 Sybil Resistance

Creating multiple identities requires controlling multiple UTXOs, each with associated economic cost (dust limit + fees). This provides natural Sybil resistance without additional mechanisms.

8 Implementation Reference

8.1 State Data Structures

```python
class Pool:
    """Represents a liquidity pool for a token pair."""
    def __init__(self, token_A, token_B):
        self.token_A = token_A
        self.token_B = token_B
        self.pool_id = canonical_pool_id(token_A, token_B)

        # Current reserves
        self.reserves = {token_A: Decimal('0'), token_B: Decimal('0')}

        # Locked positions: operator -> {token -> amount}
        self.locked_balances = defaultdict(lambda: defaultdict(Decimal)
            )

        # Unlock heights: operator -> block_height
        self.unlock_heights = {}

        # Accumulated fees for distribution
        self.accumulated_fees = {token_A: Decimal('0'), token_B:
            Decimal('0')}

class GlobalState:
    """Global protocol state."""
    def __init__(self):
        # Main balances: operator -> {token -> amount}
        self.balances = defaultdict(lambda: defaultdict(Decimal))

        # Active pools: pool_id -> Pool
        self.pools = {}

        # Current block height
        self.block_height = 0
```

8.2 Operation Handlers

```
def handle_init(state, operator, token_provided, token_sought,
                amount, lock_blocks):
    """
␣␣␣␣Process␣an␣init␣operation.
␣␣␣␣"""
    # Validate sufficient balance
    if state.balances[operator][token_provided] < amount:
        return state  # Insufficient balance

    # Get or create pool
    pool_id = canonical_pool_id(token_provided, token_sought)
    if pool_id not in state.pools:
        state.pools[pool_id] = Pool(token_provided, token_sought)
    pool = state.pools[pool_id]

    # Debit main balance
    state.balances[operator][token_provided] -= amount

    # Credit locked balance
    pool.locked_balances[operator][token_provided] += amount

    # Set unlock height
    unlock_height = state.block_height + lock_blocks
    pool.unlock_heights[operator] = max(
        pool.unlock_heights.get(operator, 0),
        unlock_height
    )

    # Update reserves
    pool.reserves[token_provided] += amount

    return state

def handle_exe(state, operator, token_given, token_received,
               amount_given, slippage_max):
    """
␣␣␣␣Process␣an␣exe␣(swap)␣operation.
␣␣␣␣"""
    # Get pool
    pool_id = canonical_pool_id(token_given, token_received)
    if pool_id not in state.pools:
        return state  # Pool doesn't exist
    pool = state.pools[pool_id]

    # Validate pool is active
    if pool.reserves[token_given] == 0 or pool.reserves[token_received]
        == 0:
        return state  # Inactive pool

    # Validate sufficient balance
    if state.balances[operator][token_given] < amount_given:
        return state  # Insufficient balance

    # Compute partial fill
    amount_actual, amount_received = compute_partial_fill(
        pool.reserves[token_given],
```

```
        pool.reserves[token_received],
        amount_given,
        slippage_max
    )

    # Compute trading fee (e.g., 0.3%)
    fee = amount_actual * Decimal('0.003')
    amount_to_pool = amount_actual - fee

    # Update balances
    state.balances[operator][token_given] -= amount_actual
    state.balances[operator][token_received] += amount_received

    # Update reserves
    pool.reserves[token_given] += amount_to_pool
    pool.reserves[token_received] -= amount_received

    # Accumulate fees for distribution
    pool.accumulated_fees[token_given] += fee

    return state
```

8.3 Complete State Transition

```python
def process_block(state, block):
    """
␣␣␣␣Process␣all␣operations␣in␣a␣Bitcoin␣block.
␣␣␣␣"""
    # First: process all lock expirations
    state = process_lock_expirations(state, block.height)
    state.block_height = block.height

    # Then: process operations in transaction order
    for tx in block.transactions:
        if not is_swap_operation(tx):
            continue

        # Extract operator from first input
        operator = get_address(tx.inputs[0])

        # Parse OP_RETURN payload
        op_return = tx.outputs[0]
        payload = parse_json(op_return.script)

        if payload['op'] != 'swap':
            continue

        # Route to appropriate handler
        if 'init' in payload:
            token_provided, token_sought = payload['init'].split(',')
            state = handle_init(
                state, operator,
                token_provided, token_sought,
                Decimal(payload['amt']),
                int(payload['lock'])
            )
        elif 'exe' in payload:
            token_given, token_received = payload['exe'].split(',')
            state = handle_exe(
                state, operator,
                token_given, token_received,
                Decimal(payload['amt']),
                Decimal(payload['slip'])
            )

    return state
```

9 Economic Incentives and Game Theory

9.1 Liquidity Provider Strategy

Consider a liquidity provider's decision problem. Let:

- V_t = expected trading volume at time t
- f = fee rate (e.g., 0.3%)
- r_t = risk-free rate at time t
- ℓ = lock period in blocks
- x = amount of liquidity provided

The expected return from providing liquidity is:

$$\mathbb{E}[R] = \sum_{t=0}^{\ell} \frac{x}{R_t} \cdot f \cdot V_t \cdot e^{-r_t \cdot t} - \mathbb{E}[\text{IL}] \tag{25}$$

where R_t is total pool liquidity at time t.

9.2 Optimal Lock Period

Lemma 3 (Optimal Lock Strategy)**.** *For a provider expecting pool growth rate $g > 0$, the optimal lock period ℓ^* increases with:*

1. *Higher expected trading volume $\mathbb{E}[V]$*
2. *Lower opportunity cost r*
3. *Higher expected pool growth rate g*

9.3 Protocol-Level Incentive Alignment

The Total Lock-up Model creates a cooperative equilibrium:

Theorem 4 (Cooperative Equilibrium)**.** *In a repeated game setting where providers observe historical lock behaviors, the Nash equilibrium converges to longer average lock periods as the protocol matures, creating a self-reinforcing stability mechanism.*

10 Comparative Analysis

Property	**LOL**	**Uniswap V2**	**Wrapped BTC DEXs**
Custody	Non-custodial, no bridges	Non-custodial (on Ethereum)	Custodial or federated
Security Model	Bitcoin L1 + deterministic computation	Ethereum L1 smart contracts	Trust in wrapper operators
Liquidity Stability	Time-locked commitments	Instant withdrawal	Instant withdrawal
State Storage	Off-chain, virtual	On-chain, expensive	Varies by implementation
Transaction Cost	Bitcoin L1 fees only	Ethereum gas fees	Bitcoin fees + bridge fees
MEV Protection	Slippage limits	Minimal (front-running common)	Varies
Composability	OPI ecosystem	ERC-20 ecosystem	Limited

Table 4: Comparative Analysis of AMM Implementations

11 Future Work

Several extensions merit further research:

1. **Concentrated Liquidity:** Adapting Uniswap V3's range orders to the time-locked model, allowing providers to specify price ranges.
2. **Dynamic Fee Structures:** Implementing variable fees based on volatility or pool utilization, with consensus rules for fee adjustment.
3. **Multi-Hop Routing:** Formal specification for optimal routing across multiple pools (e.g., $A \to B \to C$ swaps).
4. **Oracle Integration:** Using pool prices as inputs for lending protocols and derivatives (future OPIs).
5. **Governance Mechanisms:** Decentralized parameter adjustment (fee rates, minimum lock periods) through token-weighted voting.
6. **Formal Verification:** Machine-checked proofs of invariants (reserve conservation, no double-spending) using Coq or Isabelle.
7. **Layer 2 Integration:** Specifications for representing LOL operations in Lightning channels or sidechains.

12 Conclusion

LOL (Layer Of Liquidity) demonstrates that sophisticated financial primitives can be constructed on Bitcoin without sacrificing the sovereignty, security, or simplicity that define the network. By leveraging Bitcoin's UTXO model as an immutable log of intent and maintaining execution state through deterministic off-chain computation, LOL achieves the functionality of a complex AMM while preserving Bitcoin's native properties.

The Total Lock-up Model represents a fundamental innovation in DeFi design, transforming liquidity provision from a speculative activity into a capital commitment mechanism. This creates aligned incentives, stable markets, and predictable liquidity depth—addressing the systemic instability that plagues traditional AMMs.

Through the OPI governance framework, LOL establishes not just a standalone DEX but a foundational liquidity layer upon which an entire ecosystem of composable financial primitives can be built. As the first major OPI for the Universal Protocol, it sets a precedent for rigorous specification, deterministic execution, and community-driven evolution.

The protocol is live, the code is open, and the invitation is clear: build the sovereign financial stack on Bitcoin, one operation at a time.

A Transaction Examples

A.1 Example 1: Initial Liquidity Provision

```
Transaction: 7a3f2...
Inputs:
  [0] 1A2b3c... (Alice's address, 10,000 LOL balance)
Outputs:
  [0] OP_RETURN: {"p":"brc-20","op":"swap","init":"lol,wtf","amt
      ":"5000","lock":"1000"}
  [1] Change: 1A2b3c... (remaining BTC)

State Transition:
  Alice's LOL balance: 10,000 -> 5,000
  Alice's locked LOL in lol-wtf pool: 0 -> 5,000
  Pool lol-wtf LOL reserve: 0 -> 5,000
  Alice's unlock height: block_height + 1000
```

A.2 Example 2: Swap Execution

```
Transaction: 9c1e5...
Inputs:
  [0] 1B4d6f... (Bob's address, 100 WTF balance)
Outputs:
  [0] OP_RETURN: {"p":"brc-20","op":"swap","exe":"wtf,lol","amt":"50","
      slip":"0.5"}
  [1] Change: 1B4d6f... (remaining BTC)

Pool State Before:
  LOL reserve: 5,000
  WTF reserve: 2,500
  k = 12,500,000

Calculation:
  delta_B = (50 * 5000) / (2500 + 50) = 98.04 LOL
  Slippage = acceptable within 0.5%

State Transition:
  Bob's WTF balance: 100 -> 50
  Bob's LOL balance: 0 -> 98.04
  Pool LOL reserve: 5,000 -> 4,901.96
  Pool WTF reserve: 2,500 -> 2,550
  k = 12,500,000 (preserved)
```

B Formal Notation Reference

Symbol	Definition
$\mathcal{O}$	Operator identity (address controlling Input[0])
P	Pool identifier (canonical)
R_A, R_B	Reserve amounts for tokens A and B
k	Constant product invariant ($k = R_A \cdot R_B$)
Δ_A, Δ_B	Swap amounts (input and output)
σ	Slippage (percentage)
ℓ	Lock period (blocks)
h	Block height
δ	State transition function
S	Global protocol state

Table 5: Formal Notation

References

[1] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System,” Cryptography Mailing List, 2008.

[2] H. Adams, et al., “Uniswap v2 Core,” Uniswap Protocol Documentation, 2020.

[3] H. Adams, et al., “Uniswap v3 Core,” Uniswap Protocol Documentation, 2021.

[4] V. Buterin, “On Path Independence,” Ethereum Research, 2016.

[5] G. Angeris and T. Chitra, “Improved Price Oracles: Constant Function Market Makers,” arXiv:1911.03380, 2019.

[6] C. Rodarmor, “Ordinals: A numbering scheme for tracking individual satoshis,” Ordinals Documentation, 2023.

[7] Domo, “BRC-20: An experimental fungible token standard for Bitcoin,” 2023.

[8] The Universal Protocol Development Team, “The Universal Protocol & Simplicity Indexer,” 2025.

[9] E. Hertzog, G. Benartzi, and G. Benartzi, “Bancor Protocol: Continuous Liquidity for Cryptographic Tokens through their Smart Contracts,” 2017.