LimeChain

Building an Improved Runes Marketplace Using OP_CAT

A research paper

by LimeChain

April 2025

limechain.tech

Table of Contents

Abstract

Contrary to popular perception, Bitcoin is not static—it is a dynamic and ever-evolving ecosystem. The rise of Ordinals and Runes demonstrates Bitcoin's potential to expand beyond its perceived use case as digital gold, offering new opportunities for innovation. Adding to this mix, Bitcoin Layer 2 (L2) solutions have further transformed capabilities by unlocking efficient scaling, much like Ethereum's L2.

At LimeChain, we are committed to exploring the advancements within Bitcoin's ecosystem. Recently, we completed a research project funded by the Starkware OP_CAT Research Fund. This research focused on improving fungible token (Runes) trading on Bitcoin using the proposed OP_CAT (Opcode Concatenate) - an opcode that enables more advanced Bitcoin scripting.

Our research seeks to demonstrate OP_CAT's capabilities and potential to make the Runes marketplace more efficient, user-friendly, and scalable.

What Are Ordinals, Runes, and Bitcoin L2s?

To understand the significance of this research, it's crucial to grasp the distinctive roles of Ordinals, Runes, and Bitcoin Layer 2 solutions.

Ordinals and Runes

Launched in 2023, Ordinals transformed Bitcoin by allowing unique digital assets to be inscribed directly onto satoshis, the smallest unit of Bitcoin. These inscriptions create a pathway for NFTs and other use cases on Bitcoin.

Meanwhile, the Runes Protocol, introduced in April 2024, is a new fungible token standard for Bitcoin. It enables efficient token creation and seamless transfers within Bitcoin’s native ecosystem, enhancing its functionality and utility. Runes are designed to leverage Bitcoin's UTXO (Unspent Transaction Output) model for improved on-chain efficiency and security, requiring only one transaction per transfer compared to the three needed by it predecessor the BRC-20 standard, making them faster and more cost-effective.

Bitcoin Layer 2 Solutions

Bitcoin Layer 2s, like the Lightning Network, aim to address the blockchain's scalability and transaction throughput. Similar to Ethereum L2s, these solutions provide faster, cheaper transactions by offloading execution and data validation to secondary layers. Yet, Bitcoin L2s are distinct in that they must overcome limitations within Bitcoin's scripting language, which lacks Turing completeness.

Compared to Ethereum's L2 landscape, Bitcoin's L2 growth remains nascent but holds immense potential for fostering new use cases.

Runes Marketplaces Today

Let's start by examining some of the most prominent marketplaces where you can trade Runes today:

1. https://magiceden.io/
2. https://www.okx.com/
3. https://unisat.io/

The Drawbacks

Currently, all transactions rely on Partially Signed Bitcoin Transactions (PSBTs). In this system, the seller signs a transaction using `SIGHASH_SINGLE | ANYONECANPAY`, which includes their Runes UTXO (Unspent Transaction Output) as an input and creates an output back to them with a specified amount. Buyers interested in purchasing tokens can then add their own inputs and outputs to validate the transaction before broadcasting it to the network.

However, there's a significant limitation: sell orders cannot be partially filled. To increase the chances of their orders being fulfilled, sellers are incentivized to split their tokens across multiple UTXOs. As a result, buyers often face two challenges: they must either fill several small orders or forgo purchases because the available offers exceed the number of tokens they wish to buy. This process creates inefficiency, leading to an increase in broadcasted transactions and higher fees for users.

Current Runes Marketplace UX: An Example

Here's a step-by-step breakdown of how the current Runes marketplace works:

- The seller (**S**) owns 100 R tokens and wants to sell 50 of them.
- The buyer (**B**) has Bitcoin and is looking to purchase 40 R tokens.

1. **S** needs to split their Runes UTXO from 1×100 into 2×50.
2. **S** creates a PSBT (Partially Signed Bitcoin Transaction) that signs for RUNE_INPUT committing to an output BTC_OUTPUT paying themselves the sale price. This is then submitted to the Runes marketplace.
3. **B** browses the Runes marketplace, reviewing all available listings for R tokens, sorted by price per token.
4. **B** finds **S**'s listing but cannot complete the purchase because it's for 50 tokens, while **B** only wants 40. **B** is left with two options: either search for another listing or wait for **S** to split their UTXO further (e.g., into 5×10 tokens) and then buy 4 of them from the updated listing.

This example highlights some of the inefficiencies in the current system, particularly when it comes to matching buyers and sellers.

Covenants in Bitcoin

In Bitcoin (BTC), while you can restrict who can spend a specific UTXO, you cannot currently enforce how that UTXO should be spent. The concept of imposing conditions on how UTXOs are spent is known as a covenant, and several proposals have been made to introduce this functionality to Bitcoin.

In 2021, Andrew Poelstra discovered that enabling `OP_CAT` alone could facilitate covenant transactions. By utilizing recursive covenants—self-replicating conditions that persist through subsequent UTXOs—we can enable innovative use cases. For instance, smart offers could be created, allowing partial fulfilment of transactions while enforcing the correct payment amount, based on purchased tokens, is sent to the seller.

Similar solutions already exist on alternative Bitcoin networks that support covenant transactions. A notable example is Agora, which facilitates swapping SLP/ALP tokens on eCash. These solutions demonstrate the feasibility of the concept, inspiring us to build a Runes marketplace directly on Bitcoin (BTC).

Selecting the Right Tools for Bitcoin Covenants

We evaluated three approaches for building covenants on Bitcoin, each with unique strengths and trade-offs. Here's what we found:

Writing Opcodes - Purrfect Vault

The Purrfect Vault is a straightforward vault contract created by Rijndael. It implements a multi-step withdrawal process using raw scripts, written without any additional helpers or tooling. While this approach offers full control, we determined it wasn't practical for our purposes due to our limited time constraints (2 months).

Bitcoin Wildlife Sanctuary Tooling

Next, we explored the Covenants Gadgets repository by the Bitcoin Wildlife Sanctuary. This toolkit simplifies the process of building Bitcoin covenant transactions by abstracting much of the underlying logic. With excellent documentation and robust tools, it’s a strong choice for those developing highly optimized, production-ready applications.

sCrypt eDSL

sCrypt is an embedded Domain-Specific Language (eDSL) built on TypeScript, designed for writing Bitcoin smart contracts. While primarily used for Bitcoin BSV, the sCrypt team has impressively adapted examples from the previous repositories for Bitcoin BTC, as showcased in their https://github.com/sCrypt-Inc/cat-contracts repo.

After careful evaluation, we chose sCrypt for our proof-of-concept (PoC) marketplace. Its ability to simplify complex contract logic—particularly for sell-order functionality—makes it the ideal solution for our needs.

Writing Script Contracts for Sell Orders

The sell order script requires four key parameters:

1. `sellerPubKey` - the public key of the seller.
2. `sellerOutput` - a byte string representing the P2WPKH of the seller.
3. `minTokenThreshold` - the minimum number of tokens the covenant must hold.
4. `runeId` - a byte string identifying the RuneID.

The covenant stores its state through a caboose, an additional `OP_RETURN` output. This caboose stores vital information, including the remaining token count and the exchange rate. When constructing the transaction that creates the covenant, it is critical that the Runes specified in the edict (i.e., the transfer of Runes) are equal to the amount recorded in the caboose.

The sell order script supports three distinct spending conditions: `buy`, `fullBuy`, and `updateOrCancel`. The `buy` and `fullBuy` conditions function similarly, with the key difference being that `buy` enforces the remaining tokens to be sent back to the recursive covenant while simultaneously updating its state. These two functions could potentially be merged in future iterations for simplicity.

The `updateOrCancel` condition, on the other hand, can only be executed by the creator of the order - the seller. It allows the creator of the order to spend the funds without any restrictions.

Multiplication in Bitcoin Scripts

Since Bitcoin scripts lack native support for multiplication opcodes, two simulated multiplication opcodes are employed: `mul128` and `u15Mul`.

- `mul128`: This opcode multiplies a number by 128 and is essential for implementing the `RSHIFT_7` arithmetic operator. It can be simulated by duplicating and adding the number seven times. While the initial implementation in sCrypt wasn't optimized, we leveraged the assembly `asm` option in the sCrypt compiler to hand-write a more efficient version. The result? An optimized `mul128` using just 14 opcodes.

`mul128` in just 14 opcodes

- `u15Mul`: This is designed for multiplying arbitrary 15-bit unsigned integers (up to a maximum value of 32,767). For this, we adopted an implementation from one of `sCrypt`'s repositories.

Encoding Runes Amount

The runes amount is encoded using LEB128, a format not supported in Bitcoin. To ensure the correct amount is sent when fulfilling an order, we developed a custom encoder and an `rshift7` arithmetic operator, which shifts a number 7 bits to the right.

For the `rshift7` implementation, rather than directly coding the shifting logic, as seen in Robin Linus's example, we opted to use hints. This approach significantly reduces the number of instructions required, cutting them down from 290 to ~20 for a single shift.

For example, shifting the number 2000 to the right by 7 bits results in 15:

$2000 >> 7 = 15$

The reverse operation involves multiplying the result by 128 (requiring a `mul128` function) and adding a remainder to it, which must be between 0 and 127.

To shift a number like 2000 in script, the calculation must be performed off-chain. The remainder and multiplier are then provided as hints—in this case, 80 and 15 respectively. The script validates the calculation by ensuring:

$multiplier * 128 + reminder == number$.

If the equation is correct, the multiplier is our shifted value.

```
@method()
static rshift7(num: bigint, reminderHint: bigint, multiplierHint: bigint): bigint {
    assert(reminderHint >= 0n && reminderHint < 128n)
    assert(SellOrder.mul128(multiplierHint) + reminderHint == num)
    return multiplierHint
}
```

`rshift7` in sCrypt (NOTE: might contain bugs, use with caution)

LEB128 encoding can be implemented by repeatedly shifting a number right by 7 until it reaches 0, while flipping every byte except the final one.

```
@method()
static leb128(num: bigint, reminderHints: FixedArray<bigint, 4>, multiplierHints: FixedArray<bigint, 4>) : ByteString {
    let ret: ByteString = toByteString("00")
    if (multiplierHints[0] == 0n) {
        ret = int2ByteString(num)
    } else {
        SellOrder.rshift7(num, reminderHints[0], multiplierHints[0])
        ret = int2ByteString(-reminderHints[0])
        num = multiplierHints[0]
        for (let i = 0; i < 3; i++) {
            if (num != 0n) {
                SellOrder.rshift7(num, reminderHints[i+1], multiplierHints[i+1])
                num = multiplierHints[i+1]
                if (num == 0n) {
                    ret += int2ByteString(reminderHints[i+1])
                } else {
                    ret += int2ByteString(-reminderHints[i+1])
                }
            }
        }
    }
    return ret
}
```

LEB128 encoding in sCrypt (NOTE: might contain bugs, use with caution)

```
{
    num: Buffer.from('d007', 'hex'), // 2000
    reminderHint: [Buffer.from('50', 'hex'), Buffer.from('0f', 'hex'), Buffer.from('00', 'hex'), Buffer.from('00', 'hex')], // 80,15,0,0
    multiplierHint: [Buffer.from('0f', 'hex'), Buffer.from('00', 'hex'), Buffer.from('00', 'hex'), Buffer.from('00', 'hex')], // 15,0,0,0
    res: Buffer.from('d00f', 'hex'),
},
```

Test vector that encodes 2000 in LEB128

Our implementation uses 4-element arrays for hints, enabling the encoding of numbers `>260,000,000`. To optimize space, the hints can be reduced to 3, as the maximum `u15` number (32,767) requires only 3 hints. Alternatively, the hints can be expanded to 5, if we implement `u31` multiplication.

Findings

Non-Standard Transactions

Marketplace transactions include two `OP_RETURN` outputs: one for the covenant state and caboose, and another for the Runes transfer. This structure classifies

them as non-standard, meaning most nodes will not propagate them. However, this limitation could be addressed in the future through the implementation of TXID reflection.

Transaction Size

The transaction size is ~5kB, but since most of the data resides in the witness, its virtual size is reduced to ~1.5kvB. Currently, the entire script is contained within a single Tapleaf. By reorganizing the three spending conditions into separate Tapleafs or combining `buy` and `fullBuy`, the virtual size could potentially be reduced further to ~800vB.

Multiplication

The covenant uses multiplication for calculating the bitcoin amount that needs to be sent to the seller using the equation:

$btcAmount = purchasedTokens * exchangeRate$.

Bitcoin doesn't have a multiplication opcode so it needs to be simulated using addition. The current implementation uses `u15` multiplication which means the maximum tokens that can be purchased from the covenant and the maximum exchange rate of satoshis for 1 token is 32,767 so the marketplace is not suitable for Runes with very large supply or big divisibility.

Token Divisibility	Maximum Purchasable Tokens
0	32767
1	3276,7
2	327,67
3	32,767

Note: when a user wants to sell a token with a divisibility of 2 for 300 sats/token, the correct exchange rate recorded in the contract is 3 sats/token. This is because the amount calculations in Bitcoin scripts operate using the token's smallest unit.

In the current implementation, the exchange rate is stored as an integer, as Bitcoin does not support floating-point numbers. Consequently, selling a token like Rune for a rational number of sats/token is impossible with the current implementation. A potential improvement would be to represent the exchange rate as a numerator and denominator, enabling the approximation of division through hints, similar to the functionality provided by the `rshift7` opcode.

Change Output

The covenant currently does not support sending the remaining satoshis to a change output. As a result, buyers must first transfer their bitcoin to a new output with an amount calculated as:

$btcPaymentAmount + (~1500 sats * currentSatsPerVByteFee)$,

Failing to do so may result in unnecessarily high transaction fees. However, the covenant can be easily updated in the future to accommodate additional outputs, improving its functionality.

Building the Marketplace

Infrastructure

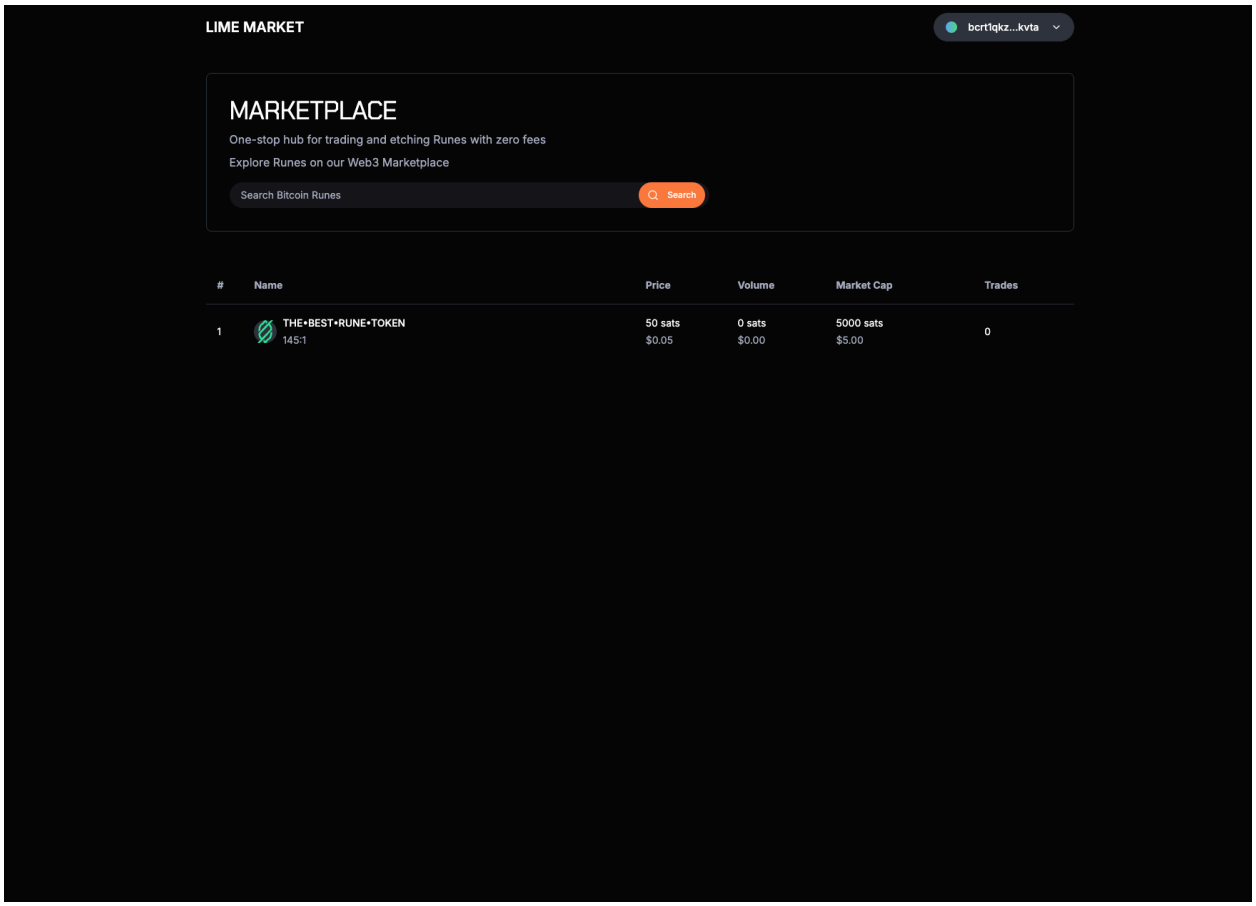
Our marketplace infrastructure consists of a `bitcoind` running on a regtest network, paired with an `ord` server. To streamline development, we've built a custom script that automates the process of restarting and resetting the regtest network. This script funds both the seller and buyer accounts, etches (mints) a Runes token, and transfers it to the seller—all designed to enable quick development iterations.

Backend

The backend is a lightweight Express server integrated with a SQLite database, connected to the `ord` server. Its primary functions include broadcasting transactions coming from the frontend, updating the listings state in the database, and retrieving Runes token data using the `ord` indexer.

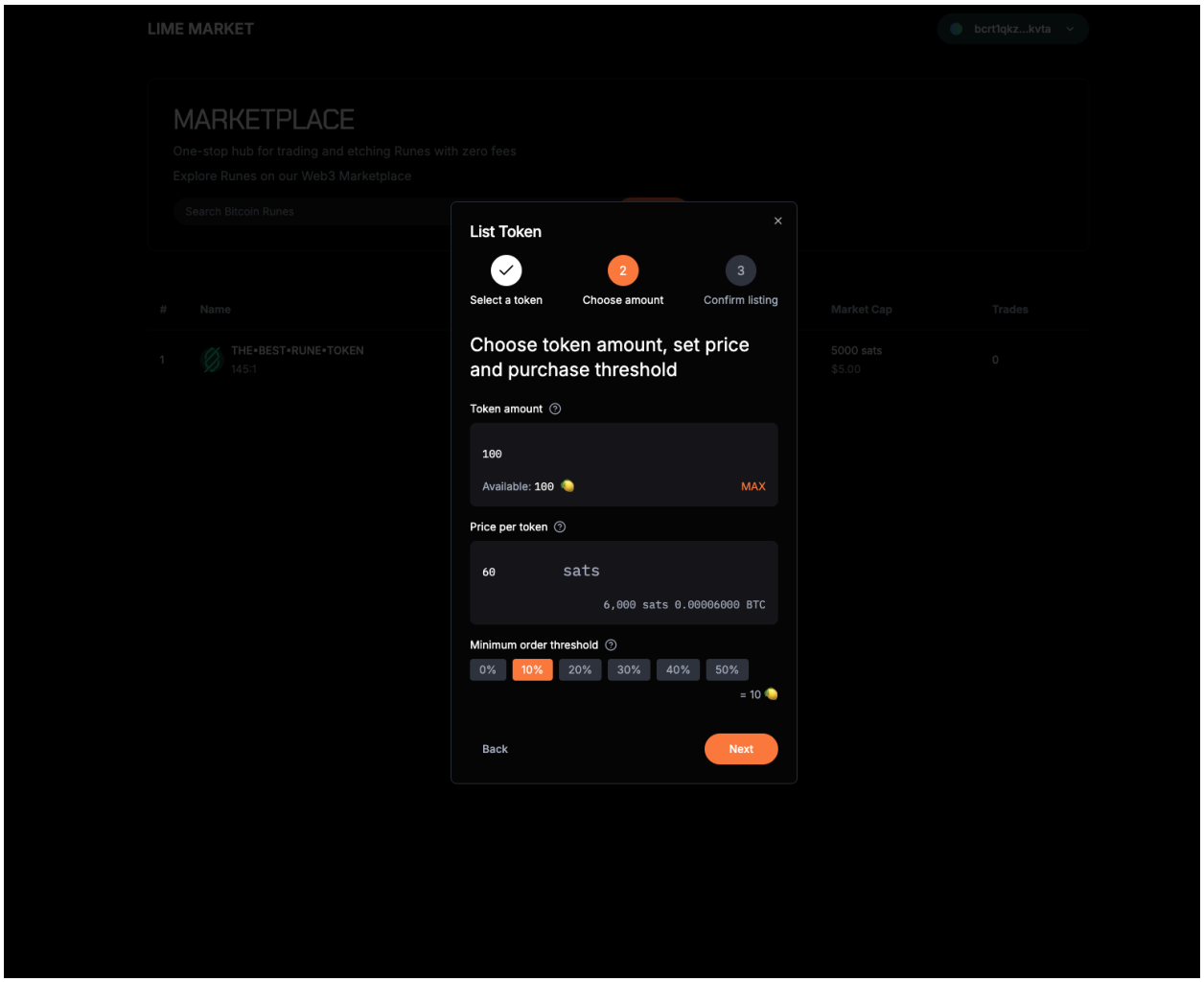
Frontend

The frontend is built with Next.js, offering an intuitive interface for users. It allows users to log in securely using a mnemonic phrase. Once logged in, they can access their dashboard to view open listings and manage their available tokens.



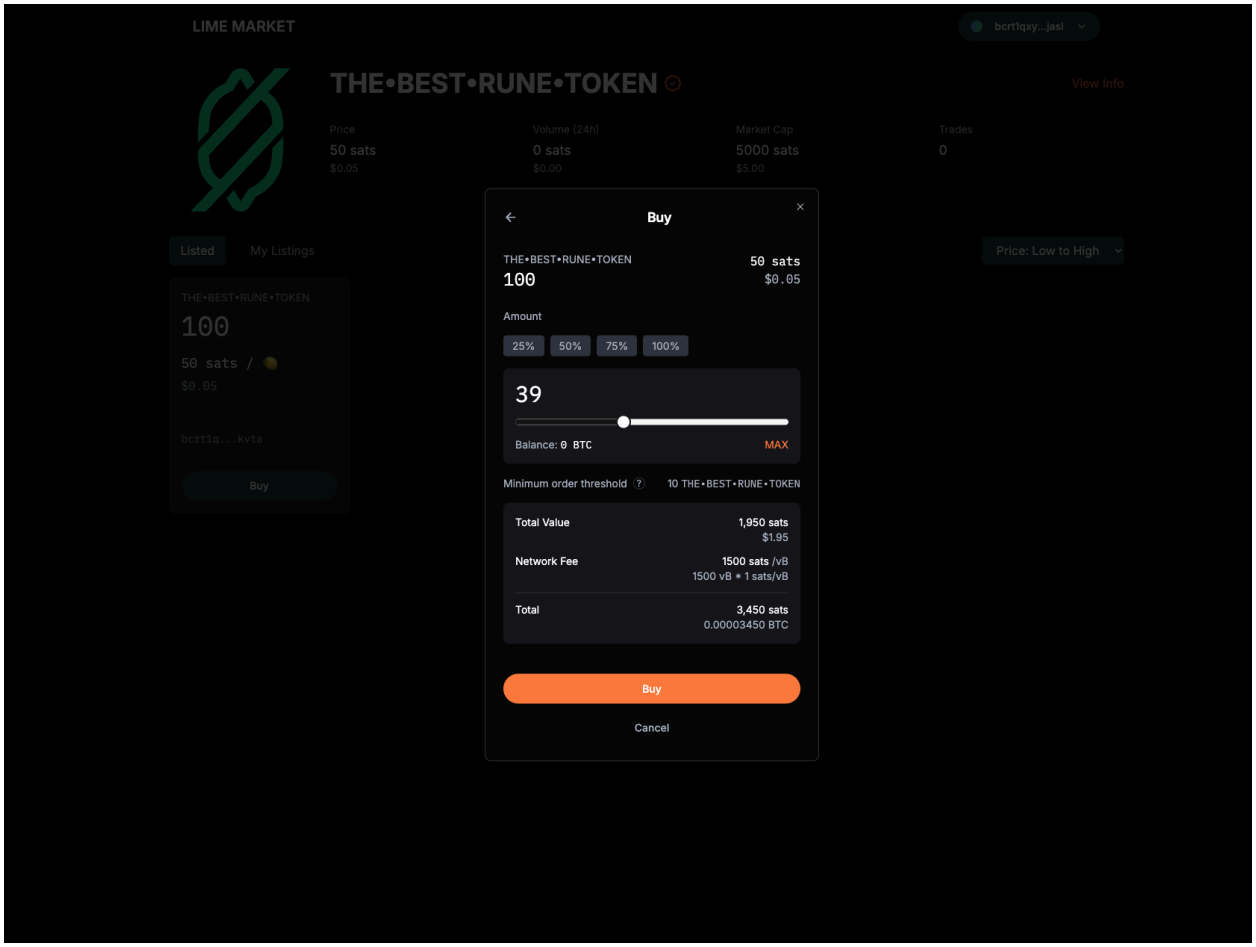
Homepage of the marketplace

Sellers can list their Runes for sale by creating offers that include a specified exchange rate and a minimum token threshold.



Creating a sell offer

Buyers can browse available token listings and purchase Runes in partial amounts directly from them.



Buying only a part of the tokens in an offer

Building Transactions on the Front-End

Integrating our contract into the marketplace was a super easy process. The sCrypt team provides detailed documentation on how to integrate with a front-end, allowing us to leverage much of the dependencies and logic already used in the contract tests.

All transaction construction and signing are handled entirely on the front end. Once the transactions are prepared, they are sent to the backend for broadcasting to the blockchain.

Conclusion

Our research proved that enabling `OP_CAT` could significantly improve and make more efficient the trading of Runes, offering a smoother user experience and eliminating the need for UTXO splitting of tokens. While the proposed swap mechanism might be more expensive than the basic swaps currently in use, it provides substantial benefits, particularly by enabling partial order fulfilment—an improvement for both buyers and sellers.

That said, Bitcoin's lack of certain operations, such as multiplication, requires workarounds that increase transaction sizes. However, with targeted optimizations, we estimate these sizes can be reduced by approximately 2x (50%).

Whether `OP_CAT` will be implemented remains to be seen, but our research suggests it has the potential to act as a significant catalyst for innovation in the Bitcoin ecosystem, unlocking new possibilities and applications.

If you'd like to discuss the subject matter further and explore future developments in the Bitcoin ecosystem, feel free to reach out to us at hi@limechain.tech.

Make sure to also check out our Runes marketplace PoC repo at https://github.com/LimeChain/runes-marketplace.