Stochastic Finance Protocol Whitepaper

January 17, 2025

Abstract

The Stochastic Finance Protocol (STFIN) transforms decentralized finance by providing a seamless platform for minting and trading digital options. Leveraging the ERC-1155 token standard and the Uniswap exchange protocol, it allows users to create and trade on-chain derivatives with concentrated liquidity across a predefined grid of strikes and maturities. Collected fees are distributed to native STFIN token holders. By employing this advanced technology, Stochastic.Finance empowers users to execute sophisticated trading strategies, access enhanced leverage, and manage risks effectively, all within a decentralized financial ecosystem.

Introduction

In recent years, decentralized finance (DeFi) has emerged as a rapidly evolving sector within the blockchain ecosystem, offering a wide array of financial instruments without intermediaries. Among these innovations, decentralized derivatives, specifically options, have gained increasing traction as they enable market participants to hedge, speculate, or enhance yield in a permissionless manner.

Projects such as Hegic, Panoptics and Derive(Lyra) have pioneered decentralized options protocols, each contributing unique approaches to on-chain options trading. Hegic, for instance, introduced a peer-to-pool model with simple options mechanics, offering users non-custodial, permissionless trading, but used fixed Implied Volatility model can lead to mispricing. Panoptic adopts a highly advanced approach to perpetual options, offering significant potential but also entails a high degree of complexity. Derive(Lyra) has sought to optimize capital efficiency through its automated market maker (AMM) model, which adjusts dynamically to market conditions, however has regulatory and geographic restrictions.

Our project seeks to address these limitations by introducing a novel approach:

- **Permissionless Options**: Options are ERC-1155 standard tokens that can be minted by liquidity providers who supply USDC as collateral. These options can be freely bought or sold by traders through an AMM. American options can be exercised at the holder's discretion, while European options are automatically executed upon expiration.
- **Deterministic Liquidity**: A predefined grid of strikes and maturities for each underlying asset ensures a predictable distribution of liquidity, effectively minimizing liquidity fragmentation.
- **Underlying Data Feed**: The option execution logic leverages Chainlink data feeds to determine the price of underlying assets. This design allows for future expansion beyond crypto pairs like ETH/USDC to include other assets and even traditional financial securities.
- **Explicit User Roles**: Each option comprises two positions or so-called 'leg': buyer and seller, represented by the bitwise structure of the TokenID. During the minting event, the address providing collateral gains ownership of both positions. Either position can be exchanged for USDC through the Stochastic Finance Swap AMM. Holders of these option positions receive a share of the collateral at the execution event, determined by the realized payoff of their respective option leg.
- **Swap AMM for ERC-1155**: The protocol features a *fee-less* AMM based on the Uniswap V2 architecture, allowing option positions to be exchanged for USDC. However, the protocol remains agnostic to the AMM used, enabling users to choose their preferred route for trading options tokens.
- **Composability**: All options for a given underlying asset are tokenized under a single ERC-1155 contract. These tokens are interoperable with other DeFi protocols, enhancing composability and integration across the ecosystem.

This design improves liquidity efficiency while addressing the liquidity fragmentation that hinders other decentralized options protocols. By combining the flexibility of ERC-1155 tokens with a tailored AMM, our protocol aims to redefine decentralized options trading, offering a scalable, efficient, and user-friendly solution for trading or hedging risks.

As the DeFi space continues to evolve, we are committed to innovation, enhancing Stochastic Finance to serve both novice and experienced traders in decentralized options markets.

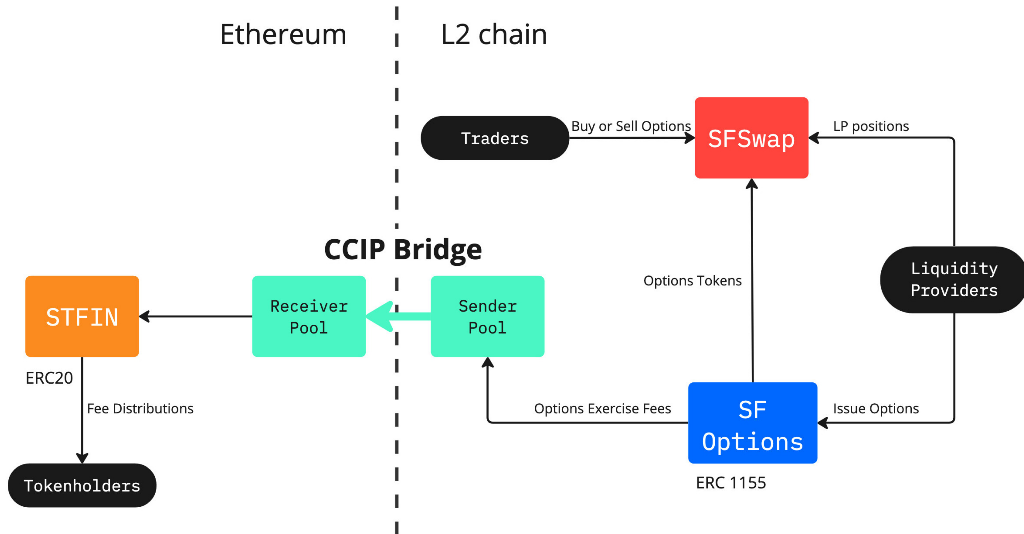

Figure 1: *Stochastic Finance Protocol: Smart contract schema.*

Stochastic Finance Protocol

The Stochastic Finance Protocol consists of several smart contracts. Their interactions with each other and external participants are illustrated in Figure 1. The main components are:

- **SF Options**: The core ERC-1155 contract responsible for issuing options, tracking participant balances, facilitating option exercises, and distributing collateral to counterparties upon exercise.
- **SF Swap**: A modified version of the Uniswap V2 AMM engine designed to enable the exchange of option tokens for USDC with *zero trading fees*, offering an enhanced user experience for trading ERC-1155 fungible tokens.
- **STFIN token**: The native ERC-20 token of the Stochastic Finance Protocol. All fees collected from exercising options are transferred to this contract and distributed equally among STFIN token holders. The token is pre-minted with a fixed supply of 1,000,000.
- **CCIP Bridge**: Since the native STFIN token resides on the Ethereum mainnet, while SF Options and SF Swap operate on an L2 chains to leverage significantly lower gas fees, a mechanism is required to transfer fees from the L2 chains to STFIN. To achieve this, Chainlink's Cross-Chain Interoperability Protocol (CCIP) will be utilized, incorporating two smart contracts: SendPool and ReceivePool.

In conclusion, the Stochastic Finance Protocol is a tightly integrated system of smart contracts designed to facilitate options issuance, trading, and fee distribution to tokenholders with minimal friction. By utilizing established technologies like ERC-1155, Uniswap V2 and Chainlink's CCIP, the protocol ensures efficient trading operations as well as interoperability across Ethereum mainnet and L2 chains while maintaining transparency and reliability. Further, we will proceed to discuss the details of each component and their specific roles within the protocol.

Stochastic Finance Options

Options Parameters Grid and Token ID

One of innovative decisions introduced by Stochastic Finance Protocol is predefined grid of strikes and maturities for all possible underlying pairs. The grid is defined by a tuple of parameters: strike and maturity. The resulting option grid is shown in table 1.

Example Token IDs, American Call

Strike/Maturity	M_i	...
...	...	...
K_i	$TokenID(K_i, M_i, A, C, D)$	...
...	...	...

Table 1: *TokenID Grid*

The set of strikes K_i is generated by starting at price equal 10 and then incrementing the price by 10% each step for 100 steps and deceasing starting price for 10% for 50 steps, thus, producing a vector of 200 possible strikes. This will cover a price range for most of actively traded assets like BTC, ETH, etc. The generated strikes are rounded to first 4 significant digits and transformed in exponential notation. Later a strike is converted to a binary representation of 6 hexadecimal digits long or 24 bits, such that first 4

digits are encoded by first 16 bits, exponent sign is 5th digit and exponent power is given by 6th digit. Strikes are predefined and stored as `uint24[]` in the smart-contract at the deployment.

The set of maturities M_i is defined on weekly basis, each Friday of the week. Similarly to strikes, maturities are also encoded by 24 bits: a year by first 8 bits, followed month and day by 8 bits as well. Maturities are predefined for 3 years ahead and stored as `uint24[]` in the smart-contract at the deployment.

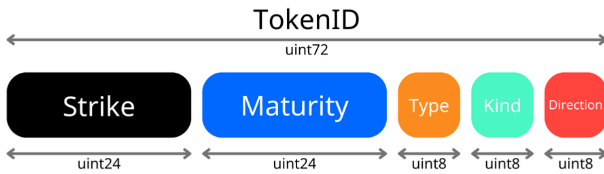

Figure 2: *TokenID bitwise structure*

Another 2 bytes are used to denote a type of option: Call or Put and a kind of option: European or American. Final byte determines direction of the leg: buyer's or seller's leg of the options.

Binding all options parameters yields a `uint72` TokenID data structure as shown on figure 2.

On-chain Ledger Structure

Next major innovative step is to utilize **ERC-1155** as a data structure[1] to store all TokenIDs generated from grid parameters permutations, as well as their respective balances and counterparties. Storing such a structure on-chain is a non-trivial task. To achieve this, we use several nested built-in Solidity mappings along with a specialized mapping data structure called `IterableMapping`[2].

At the top level, there is a `participants` mapping that associates an `address` with a `Participant` struct. This struct contains a mapping from `TokenId` to a `Trades` struct. The `Trades` struct, in turn, includes an `IterableMapping` object that holds positions and their respective counterparties.

Figure 3: *Ledger structure after issuance of 1 option by address A.*

To better understand how the balance is managed, let's walk through an example usage of the protocol: a user with Address A wants to mint 1 option on the ETH/USDC pair with a specific strike.

First, the function `issueOption` is called with option parameters such as maturity, strike, type, kind, and direction. It requests approval for `transferFrom` from `msg.sender` of a stablecoin like USDC as collateral. The value of the collateral corresponds to the number of options being issued; for example, 1 USDC of collateral equals 1 option on the ETH/USDC pair.

Afterward, the validity of the input is checked to ensure the specified strike and maturity exist on the Options Parameters Grid, and the respective `TokenId` is determined. Once all checks are passed, two tokens are minted:

• The buyer's leg, i.e. long call option, associated with the `TokenId`.
• The seller's leg, i.e. short call option, associated with the inverted direction `TokenId`.

At this point, the issuer owns both legs of the option (buyer's and seller's sides) as illustrated on figure 3. In the case of expiration or early execution, the issuer would receive the entire initial collateral. Address A can then sell or transfer one of the option legs, just like any ERC-1155 type token. Minting options is a mechanism for providing liquidity to the Stochastic Finance Protocol.

Option Exercising and Payoff

Similar to traditional finance, European options are executed at maturity, which occurs on a weekly basis according to the grid. In contrast, holders of American option tokens can exercise their options at any time. When an option is exercised, the protocol leverages the Chainlink data feed to fetch the latest price of the underlying asset.

Since Stochastic Finance uses collateralized options, the payoff function differs from that of conventional vanilla options[3]. It features a non-trivial structure where the payoff represents a fraction of the supplied collateral. The payoffs for the buyer's and seller's legs are calculated as follows:

$$\text{Payoff}_{call\ long} = C\left[\frac{\max(S_T - K, 0)}{\max(S_T - K, 0) + K}\right]$$

$$\text{Payoff}_{call\ short} = C\left[1 - \frac{\max(S_T - K, 0)}{\max(S_T - K, 0) + K}\right]$$

where K is a strike, S_T is a price of underlying pair at option exercise, C - is a notional of collateral.

Let's consider a simple example to understand fractional payoffs. Address A holds 1 buyer's leg of a call option on ETH/USDC, while address B holds 1 seller's leg of the same call option. The underlying collateral, which will be distributed between the buyer and the seller, is equal to 1 USDC. The distribution works as follows:

1) If at the option's exercise time, the price S_T of ETH/USDC is lower than the option's strike price K, then the buyer will not receive any fraction of the collateral, and the seller will receive the full collateral amount, minus SFP fees.

2) If the price S_T of ETH/USDC is higher than the

strike price K : The buyer will receive a fraction of the collateral. For instance, if S_T is twice as high as K , the buyer's fraction will be 50%, minus fees. If S_T is four times as high as K , the fraction increases to 75%, and so on. This fractional payoff function is distinct from the traditional vanilla call option payoff, which is well-studied in academic literature.

Option Pricing

Correspondingly, the price of such fractional option is also non-trivial. Assuming geometric Brownian motion process for underlying asset ETH/USDC one can derive the analytic expression for a price for such option. The price of the option V is given by the expected value of the discounted payoff under the risk-neutral measure $\mathbb{Q}$:

$$V = e^{-rT}\mathbb{E}^{\mathbb{Q}}\left[\frac{\max(S_T - K, 0)}{\max(S_T - K, 0) + K}\right]$$

Substituting the explicit form of the payoff:

$$V = e^{-rT}\mathbb{E}^{\mathbb{Q}}\left[\frac{S_T - K}{S_T} \cdot \mathbb{I}(S_T > K)\right]$$

where $\mathbb{I}(S_T > K)$ is an indicator function that equals 1 when $S_T > K$ and 0 otherwise. Let's express the expected value in terms of the distribution of S_T, the stock price at time T under the risk-neutral measure:

$$V = e^{-rT}\int_K^\infty \frac{S_T - K}{S_T} f_{S_T}(S_T)dS_T$$

Under the Black-Scholes model, the stock price S_T is log-normally distributed:

$$S_T = S_0 \exp\left(\left(r - \frac{\sigma^2}{2}\right)T + \sigma W_T\right)$$

where W_T is a standard Brownian motion under the risk-neutral measure and PDF of S_T is:

$$f(S_T) = \frac{1}{S_T\sigma\sqrt{2\pi T}} \exp\left(-\frac{\left(\ln\left(\frac{S_T}{S_0}\right) - \left(r - \frac{\sigma^2}{2}\right)T\right)^2}{2\sigma^2 T}\right)$$

We can now rewrite the option price as:

$$V = e^{-rT}\int_K^\infty \left(1 - \frac{K}{S_T}\right) f_{S_T}(S_T)dS_T$$

This can be split into two integrals:

$$V = e^{-rT}\left[\int_K^\infty f_{S_T}(S_T)dS_T - K\int_K^\infty \frac{1}{S_T} f_{S_T}(S_T)dS_T\right]$$

The first integral

$$\int_K^\infty f_{S_T}(S_T)dS_T$$

is simply the probability $\mathbb{P}^{\mathbb{Q}}(S_T > K)$, which is given by $\Phi(d_2)$ where: $d_2 = \frac{\ln\left(\frac{S_0}{K}\right) + \left(r - \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}}$

The second integral can be expressed as:

$$\frac{1}{\sigma\sqrt{2\pi T}}\int_K^\infty \frac{1}{S_T^2} \exp\left(-\frac{\left(\ln\left(\frac{S_T}{S_0}\right) - \left(r - \frac{\sigma^2}{2}\right)T\right)^2}{2\sigma^2 T}\right) dS_T$$

Fortunatly such integral has a closed-form solution. The final formula for long and short call option prices look as following:

$$C_{long\ call} = \Phi(d_2)\exp^{-rT} - \frac{K}{S_0}(1 - \Phi(\eta\sqrt{2}))\exp^{-(2r-\sigma^2)T}$$

$$C_{short\ call} = \Phi(-d_2)\exp^{-rT} + \frac{K}{S_0}(1 - \Phi(\eta\sqrt{2}))\exp^{-(2r-\sigma^2)T}$$

where:

$$\zeta = \frac{3\sqrt{2}T\sigma^2 - 2^{3/2}Tr + 2^{3/2}\ln\left(\frac{K}{S_0}\right)}{4\sqrt{T}\sigma}$$

Using the same assumptions one can derive the price of both directions of a put option:

$$P_{long\ put} = \Phi(-d_2)\exp^{-rT} - \frac{S_0}{K}(1 - \Phi(\eta\sqrt{2}))$$

$$P_{short\ put} = \Phi(d_2)\exp^{-rT} + \frac{S_0}{K}(1 - \Phi(\eta\sqrt{2}))$$

where:

$$\eta = \frac{T\left(\sigma^2 + 2r\right) - 2\ln\left(\frac{K}{S_0}\right)}{2^{\frac{3}{2}}\sqrt{T}\sigma}$$

As usual, the analytic expression of prices may appear intimidating, but they can be validated using a simple Monte Carlo simulation, which produces the same results as shown in the listing 1.

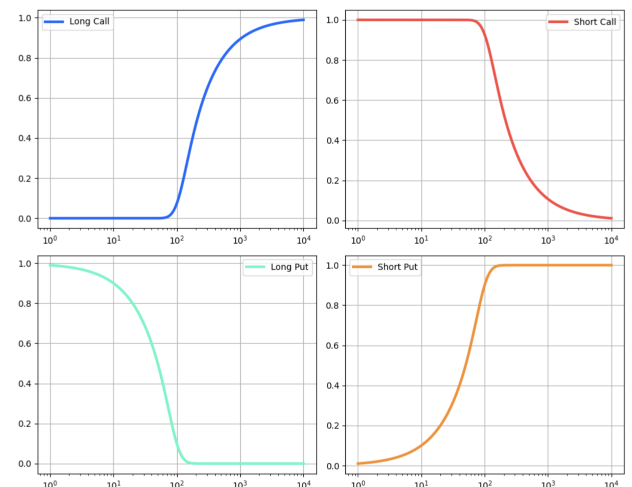

Figure 4: *Option prices depending on price os the underlying for K=100, T=1*

Price dependence on underlying S_T for all options is shown on figure 4. Here similar to vanila options the price quickly fades as options goes out-of-money. However, asymptotic behavior of the price in in-the-money region is different: fractional options do not exhibit linear behavior of vanila options when at $S_T >> K$ option price is almost equal to the price of underlying. Instead, as $S_T >> K$ for long calls the price approaches 1, meaning that the owner of those options would get 100% collateral.

```
1  import numpy as np
2  from scipy.stats import norm
3  from scipy.special import erfc
4
5  def option_price(S0, K, r, T, sigma):
6      d2 = (np.log(S0 / K) + (r - 0.5 * sigma**2) * T) / (sigma * np.sqrt(T))
7      zetta = (3 * np.sqrt(2) * T * sigma**2 - 2**(3 / 2) * T * r + 2**(3 / 2)
          * np.log(K / S0))/(4 * np.sqrt(T) * sigma)
8      option_price = np.exp(-r * T) * (norm.cdf(d2)) - K / S0 * 0.5 * erfc(
          zetta)*np.exp(sigma**2 * T)*np.exp(-2*r * T)
9      return option_price
10
11 def monte_carlo_price(S0, K, r, T, sigma, num_simulations=100000000):
12     Z = np.random.standard_normal(num_simulations)
13     ST = S0 * np.exp((r - 0.5*sigma**2)*T + sigma*np.sqrt(T)*Z)
14     payoff = np.maximum(ST - K, 0) / (np.maximum(ST - K, 0) + K)
15     return np.exp(-r*T) * np.mean(payoff)
16
17 # Test parameters
18 S = 100  # Current stock price
19 K = 100  # Strike price
20 r = 0.05  # Risk-free rate
21 T = 1     # Time to maturity (in years)
22 sigma = 0.25  # Volatility
23
24 # Calculate prices
25 analytical = option_price(S, K, r, T, sigma)
26 mc = monte_carlo_price(S, K, r, T, sigma)
27
28 print(f"Analytical price: {analytical:.6f}")
29 print(f"Monte Carlo price: {mc:.6f}")
30 print(f"Difference: {abs(analytical - mc):.6f}")
31
32 >>> Analytical price: 0.0893656
33 >>> Monte Carlo price: 0.089356
34 >>> Difference: 0.000000
```

Listing 1: *Analytic solution vs Monte-Carlo simulations results.*

Analytic formulas for pricing are extremely useful when constructing more complex option strategies, such as straddles, spreads, butterflies, and others. These expressions will later be integrated into the Strategy Builder UI, enabling users to forecast payoffs for any custom option strategy.

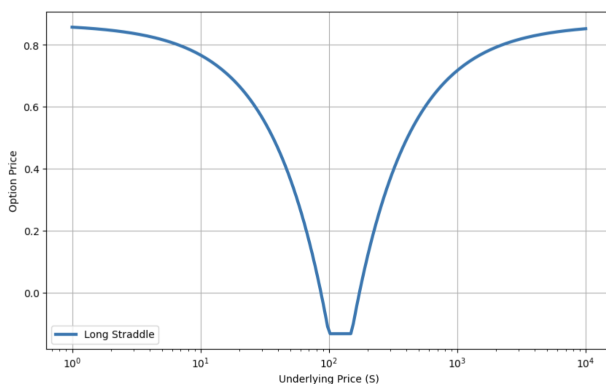

Figure 5: *Example Long Straddle strategy, consistiing of long call at K=100 and long put at K=150*

Stochastic Finance Swap

Users who are only interested in purchasing options or trading can buy or sell them on any DEX that supports the ERC-1155 token standard.

However, to provide a better user experience for traders, we will be launching a specialized DEX based on the Uniswap V2 [4] engine with *zero fees* on swap operations. Feeless swaps will further enhance trading liquidity and increase asset turnover.

This section offers detailed explanations of the design solutions underlying the primary Stochastic Finance Swap contracts (hereinafter referred to as SF-Swap). Here, we provide a brief overview of the contract functions, including support for arbitrary pairs between ERC20 (stablecoin) and ERC1155 (trading coin), an AMM mechanism that allows traders to monitor pair liquidity and collect price data, as well as "flash swaps," which enable traders to buy or sell options created on the Stochastic Finance dApp.

SFSwap enables liquidity providers to create paired contracts between the stablecoin (USDC) and any fungible token defined within an ERC-1155 contract. The numerous possible pairs between USDC and ERC-1155 tokens can make identifying the optimal trading path for a specific option or pair challenging. However, the Stochastic Finance dApp provides a user-friendly interface to facilitate such trades efficiently.

The SFSwap contract builds upon and enhances the core logic of Uniswap V2, incorporating its best features. Additionally, we have introduced several significant improvements and features focused on enhancing security.

Below is a list of key differences and a brief description of the solutions we implemented:

- **Updated Solidity Version**: While SFSwap inherits many foundational elements from Uniswap, we use Solidity version 0.8.24 to minimize the risk of vulnerabilities present in earlier versions of the language. For more detailed information about changes in Solidity versions, refer to the Release Announcements section on the Solidity project's website [5].
- **Secure Mathematics**: Throughout development, we prioritized the use of secure mathematical operations to mitigate potential errors. We relied heavily on Math and SafeCast technologies from OpenZeppelin version 5 to ensure accuracy and prevent issues such as overflows and underflows. Although Solidity now includes built-in protections against overflow and underflow, we conducted additional checks using well-established tools to further enhance security.

- **Reentrancy Protection**: Special attention was given to guarding against reentrancy attacks to ensure the safety of users' tokens during transactions. To achieve this, we avoided using custom-written modifiers and instead implemented OpenZeppelin's ReentrancyGuard from version 5, a robust and tested solution for preventing such vulnerabilities [6].

These represent just a few of the potential vulnerabilities we identified and addressed during development. We have taken every possible measure to ensure the security and reliability of our platform, prioritizing the safety of our users at every step.

We will discuss them in details in subsequent sections.

Price Oracle

In the absence of transactions, the marginal price on SFSwap can be calculated as the ratio of the reserves of asset a to the reserves of asset b:

$$p_t = \frac{r_t^a}{r_t^b}$$

SFSwap extends the basic functionality of this price oracle by measuring and registering the price before the first transaction of each block (or equivalently, after the last transaction of the previous block). This design significantly increases the difficulty of price manipulation compared to manipulating prices within a block.

For instance, if an attacker attempts to manipulate the price at the end of a block, another arbitrageur can immediately send a counteracting transaction within the same block. While a miner or an attacker with sufficient resources could manipulate the price at the block's end, they would gain little advantage in arbitraging if they are not mining the subsequent block.

To enhance security, SFSwap accumulates price data by tracking the cumulative volume of prices at the beginning of each block where the contract is interacted with. Each price is weighted based on the time elapsed since the last block update, as determined by the block's timestamp. Consequently, the cumulative cost at any given time (after an update) equals the sum of spot prices for every second in the contract's history:

$$a_t = \sum_{i=1}^{t} p_i$$

To compute the time-weighted average price $TWAP$ between two points, t_1 and t_2, an external user can retrieve the cumulative value at t_1, retrieve it again at t_2, subtract the initial value from the latter, and divide the result by the elapsed time in seconds:

$$TWAP_{t_1,t_2} = \frac{a_{t_2} - a_{t_1}}{t_2 - t_1}$$

It is important to note that the contract does not store historical values for this cumulative price. Therefore, the user must query the contract at the start of the desired period to record the initial value and again at the end of the period to complete the calculation:

$$p_{t_1,t_2} = \frac{\sum_{i=t_1}^{t_2} p_i}{t_2 - t_1} = \frac{\sum_{i=1}^{t_2} p_i - \sum_{i=1}^{t_1} p_i}{t_2 - t_1} = \frac{a_{t_2} - a_{t_1}}{t_2 - t_1}$$

Precision

Due to Solidity's lack of first-class support for non-integer numeric data types, SFSwap adopts a simple binary fixed-point format to encode and manipulate prices. Specifically, prices at any given moment are represented as UQ112.112 numbers. This format allocates 112 fractional bits of precision on either side of the decimal point and does not include a sign. These numbers have a range of $[0, 2^{112} - 1]$ [1] and a precision of $\frac{1}{2^{112}}$.

The UQ112.112 format was chosen for pragmatic reasons. Since these numbers can be stored in a single uint224, this leaves 32 bits free in a 256-bit storage slot. Additionally, reserves, each stored as a uint112, also leave 32 bits free in a (packed) 256-bit storage slot. These free bits are utilized for the accumulation process described earlier. Specifically, reserves are stored alongside the timestamp of the most recent block with at least one trade. This timestamp is modded by 2^{32} to fit within the remaining 32 bits.

Although the price at any given moment (stored as a UQ112.112 number) fits comfortably within 224 bits, the cumulative price over an interval may exceed this limit. The additional 32 bits in the storage slots for the accumulated prices of A/B and B/A are reserved to capture overflow bits resulting from repeated summations of prices.

One potential drawback of this approach is that 32 bits are insufficient to store timestamp values that are guaranteed not to overflow. The Unix timestamp will overflow a uint32 on February 7, 2106. To ensure continued functionality beyond this date—and every subsequent interval of 2^{32} - 1 seconds (approximately 136 years)—oracles must check prices at least once per interval.

[1] The theoretical upper bound of $2^{112} - \frac{1}{2^{112}}$ does not apply in this setting, as UQ112.112 numbers in Uniswap are always generated from the ratio of two uint112s. The largest such ratio is $\frac{2^{112}-1}{1} = 2^{112} - 1$

The system is designed to be overflow-safe. The core method of accumulation and timestamp modding ensures that trades spanning overflow intervals are correctly accounted for. This requires oracles to use straightforward overflow arithmetic to compute deltas accurately, allowing the system to maintain precision and functionality over extended periods.

Flash Swaps

SFSwap allows users to receive and utilize an asset without prepayment, as long as the payment is completed within the same transaction. The swap function triggers the execution of an optional callback contract specified by the user, which occurs between the transfer of tokens requested by the user and the enforcement of the invariant.

After the callback is executed, the contract verifies the updated balances to ensure that the invariant is maintained, factoring in the fees applied to the deposited amounts. If the contract does not contain sufficient funds to satisfy the invariant, the transaction is reverted.

Fee to liquidity providers

Our protocol assumes a fee for liquidity providers in the amount of 1 % of each transaction, which can be earned on each pair selected by the provider. Also, provided that when an option is issued, each provider receives two legs (buyer and seller side), them can be provided as liquidity assets to either one or both pools. Reward is distributed among providers depending on their contribution to a particular pool in accordance with their share. The rewards from each transaction are immediately transferred back to the pool, which allows to maintain a more stable price of assets and reduce the price impact. Accumulated fees can be obtained at any time by burning off liquidity tokens by any of the providers.

Contract Architecture

A core focus of SFSwap's development is minimizing complexity while enhancing the security of the primary pair contract, which holds the assets of liquidity providers.

Errors in this contract could have severe consequences, potentially leading to the theft or locking of liquidity. Therefore, the most critical consideration when evaluating the security of this fundamental contract is whether it adequately safeguards liquidity providers against the loss or immobilization of their assets.

Additional functions aimed at supporting or protecting traders, beyond the core functionality of exchanging one asset for another within a pool, can be implemented in a router contract. In fact, even some exchange-related functions can be delegated to the router contract to reduce the complexity of the pair contract.

As previously mentioned, SFSwap maintains the last recorded balance of each asset to prevent the manipulative exploitation of its oracle mechanism.

Initialization of Liquidity Token Supply

When a new liquidity provider contributes tokens to an existing SFSwap pair, shares are issued. The number of shares issued is determined by the geometric average of the amounts deposited into the liquidity pool.

The formula for calculating the number of shares is as follows:

$$s_{\text{minted}} = \sqrt{x_{\text{deposited}} \cdot y_{\text{deposited}}}$$

where:

$x_{\text{deposited}}$ is the amount of token x deposited into the liquidity pool; $y_{\text{deposited}}$ is the amount of token y deposited into the liquidity pool. This formula ensures that the value of a liquidity pool share remains stable, regardless of the initial ratio of the deposited assets.

For example, if the current value of 1 ABC is 100 XYZ, and an initial deposit of 2 ABC and 200 XYZ is made (maintaining a 1:100 ratio), the depositor will receive:

$$\sqrt{2 \cdot 200} = 20 \text{ shares}$$

The value of these shares represents 2 ABC and 200 XYZ, plus any accumulated commissions.

This formula guarantees that the value of a liquidity pool share will never fall below the geometric average of the pool's reserves [2]. However, the value of a share may increase over time due to "donations" or excess funds contributed to the liquidity pool. Theoretically, this could result in a situation where the minimum number of pool shares (1e-18) becomes disproportionately valuable, making it impractical for small liquidity providers to contribute liquidity.

To mitigate this, SFSwap burns the first 1e-15 (0.0000000000000001) of the total pool resources created (1,000 times the minimum value of total pool resources). These burned resources are sent to an

[2] This also reduces the likelihood of rounding errors, since the number of bits in the quantity of shares will be approximately the mean of the number of bits in the quantity of asset X in the reserves, and the number of bits in the quantity of asset Y in the reserves:

$$log_2\sqrt{x \cdot y} = \frac{log_2 x + log_2 y}{2}$$

address with a zero balance, ensuring they are permanently removed, rather than returned to the mint. This adjustment has a negligible cost for almost any token pair but significantly increases the difficulty of exploiting this scenario.

For instance, to raise the value of a liquidity pool share to $100, an attacker would need to deposit $100,000 into the pool, which would be permanently locked as liquid funds.

Deterministic Pair Addresses

With advancements in the Solidity language and the potential vulnerabilities associated with the inline, custom use of the CREATE and CREATE2 opcodes in Uniswap V2, we decided to avoid these methods. These vulnerabilities could result in user funds being at risk if not implemented correctly.

Despite this, the need for cloning contracts persisted. To optimize gas usage, we adopted the cloneDeterministic proxy technology from OpenZeppelin version 5 [7]. This technology creates and returns the address of a clone that replicates the behavior of the original implementation.

The cloneDeterministic function utilizes the CREATE2 opcode along with a salt value to deploy deterministic clones. Importantly, the same implementation and salt value cannot be reused, as deploying a clone at an address already in use is not allowed.

This approach provides a more standardized and robust solution, offering greater protection against potential attacks.

As mentioned earlier, the development of SFSwap placed significant emphasis on addressing security concerns and leveraging advanced security technologies in Solidity.

Pair timing

Two hours before maturity, each of the pools is closed for trading and/or adding liquidity to increase the security of assets located inside the pool. The regulation of such a process is provided directly by the blockchain by calculating the block to maturity. Asset withdrawal is still available to any pool participant.

Pair disband before exercise

After the pool has been "frozen" for the deposit and exchange of assets. After maturity occurs, the option is being exercised. At the time of the exercising, all assets are returned to the providers in accordance with their shares. Thus, in order to avoid any tricky situations, the token representing a specific pair must be held by the provider of this pair and should not be transferred to third parties. In a normal situation, a token representing liquidity will be extracted from your account and transferred to burn, after which you will receive a payment of both parts of your asset. And after that, an exercise will be performed.

sync() and skim()

To mitigate the risk of custom token implementations updating the balance of paired contracts, and to more gracefully handle tokens with a total supply exceeding 2^{112}, SFSwap offers two fallback mechanisms: `sync()` and `skim()` similar to UniSwap v2 [4].

`sync()` serves as a recovery method when a token reduces the balance of its paired contract asynchronously. In such a scenario, trades may receive suboptimal exchange rates, and if no liquidity providers are available to rectify the situation, the paired contract may become stalled. `sync()` sets the contract's reserves to their current values, facilitating a relatively smooth resolution.

`skim()` acts as a contingency plan in cases where excess tokens are transferred to a paired contract, potentially exceeding the capacity of the two uint128 slots for reserves and potentially causing trade failures. `skim()` allows users to withdraw any difference between the current balance and $2^{112} - 1$ if it exceeds zero.

Divergence (impermanent) loss

When providing liquidity, the provider needs to understand the risks and advantages that it carries as a provider in terms of holding assets in the pool. The concept of divergence loss covers a situation in which, when the price of a trading asset changes, the token holder incurs losses in relation to the situation if he simply held such assets. There are many high-quality examples in the community representing the classic situation where there are two tokens in the pool, and when the price changes, we get a loss for the provider. In general, such a loss can be described by the following formula [8]:

$$divergenceLoss = \frac{1 + r - 2 \cdot \sqrt{r}}{1 + r}$$

or this approach:

$$divergenceLoss = \frac{2 \cdot \sqrt{r}}{(1 + r) - 1}$$

where r - price ratio: $r = \frac{p'}{p}$
r - price ratio,
p' - current price
p - initial price

and we can imagine the losses for the provider as follows:

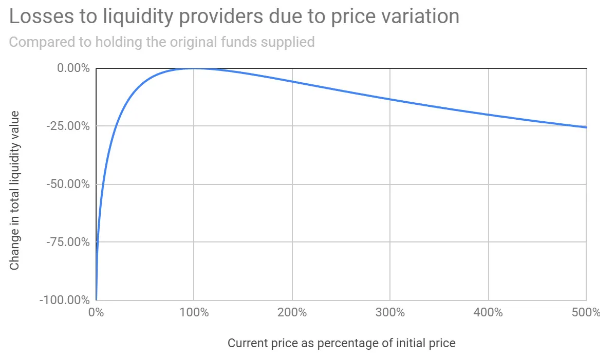

Figure 6: *Example of losses for liquidity providers due to price variation[9]*

or in numbers:
Price change to loss relative to HODL

- 1.25x cnange turns in a 0.6% loss
- 1.50x cnange turns in a 2.0% loss
- 1.75x cnange turns in a 3.8% loss
- 2x cnange turns in a 5.7% loss
- 3x cnange turns in a 13.4% loss
- 4x cnange turns in a 20.0% loss
- 5x cnange turns in a 25.5% loss

However, in this explanation, we still did not touch in any way on the reward that will be accumulated by the provider with each transaction. Taking it into account, we will get such a result that gradually, as we go through the transactions with the assets of each of the liquidity holders, his remuneration will accumulate inside the pool and as a result at some point of time, the remuneration will cover the loss of value.

So we can consider two examples:

1. Negative P&L example. Here we will take big price impact due to transactions.

1. Initial option price: 0.5 USDC
2. The final option price after all completed transactions $\approx$ 0.953 USDC
3. Number of transactions: 200
4. Initial liquidity supply: 1 500 USDC
5. Number of providers: 2
6. Share of each provider: 50 %

Further, after performing calculations, each of providers will receive:

Divergence loss ≈ -0.050
Accumulated fees $\approx$ 19.661 UDSC
Losses relative to simple asset retention:
$1400.016 - 1452.690 \approx -52.674$, taking into account the fee overlap in 1%

2. Positive P&L example Here, if we take all the same initial conditions, however, change the initial option price:

1. Initial option price: 1 USDC
2. The final option price after all completed transactions $\approx$ 0.996 USDC
3. Number of transactions: 200
4. Initial liquidity supply: 2 000 USDC
5. Number of providers: 2
6. Share of each provider: 50 %

Divirgence loss $= -0.000002246287445717421$
Accumulated fees $\approx$ 19.909 USDC
Gains relative to simple asset retention:
$2015.674 - 1995.770 \approx 19.904$, taking into account the fee overlap of 1%

Thus, we can understand that with an increase in the number of transactions, more and more fees will flow into the provider's account and cover an increasing change in value, as well as an increase in the number of assets provided will keep the value more in place, which will also reduce the impermanent loss.

Last but not least, we can say that the very first thing is that all these risks may come true only for a situation where the provider uses only one leg of its option in the liquidity pool. Thanks to the capabilities of our service, the entire price impact can be levelled if both legs are provided with liquidity at once, in which case option prices will change counter-directionally and the provider will receive pure earnings due to accumulated fee.

Pair name construction

When creating our service, in order to improve the user experience, we adopt a unique method of naming pairs (pools) based on generally accepted methods and international practices. We always take into account the composition of the name:

1. Underlying assets
2. Option type: American or European
3. Option kind: Call or Put
4. Direction: Buy or Sell
5. Maturity
6. Strike

The most understandable example will be: `ETH250117ACB88200000`
Here:

1. Underlying assets:USDC and Option
2. Option type: American
3. Option kind: Call
4. Direction: Buy
5. Maturity: 17.01.2025
6. Strike: 882.0

For even more easy understanding users may consider take a look to our schema:

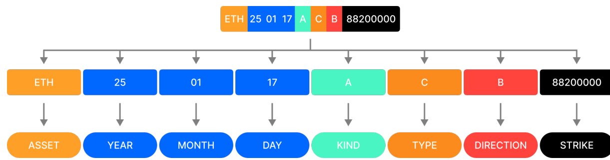

Figure 7: *Example of how to pools are named*

STFIN Token

Tokenomics

STFIN is the native token of the Stochastic Finance Protocol, designed to distribute all fees collected from exercising options. While trading options via SF Swap incurs no fees apart from gas costs, exercising options carries a 0.5% fee on the collateral. This fee is evenly split between the buyer and the seller upon exercise.

To estimate the potential annual cash flow distributable to STFIN tokenholders, we consider typical option dynamics. Observing trends from other option platforms, most liquid options tend to have weekly maturities. For simplicity, we assume that the Total Value Locked (TVL) is fully concentrated in weekly options. As these options roll over each week, 0.5% of the TVL is converted to fees and distributed to STFIN tokenholders. Over a year, this amounts to $0.5\% \times 52 = 26\%$ of TVL being collected as fees, all of which are distributed to tokenholders.

To project the future TVL of the protocol, we employ a statistical approach. Using data from **defillama.com** as of December 25, 2024, which encompasses TVL from 3,081 DeFi projects, we analyze the distribution of TVL and calculate quantiles to simulate different scenarios for the protocol's TVL. By combining these projections with the annual fee rate, we derive the protocol's potential annual cash flow and assess its future valuation. Table 2 presents the possible cash-flow outcomes based on these projections.

These results indicate that even if Stochastic.Finance achieves a most common, i.e. median TVL across the DeFi industry, it would still generate $1.5 million in annual cash flow for its tokenholders.

We plan to distribute STFIN tokens using a standard allocation model commonly seen in the DeFi space:

- **Team and Founders**: 30%
- **Investors**: 20%
- **Advisors**: 5%
- **Marketing**: 10%
- **Public Sale**: 10%
- **Community Fund**: 20%
- **Airdrop**: 0.5%

	TVL 25%	TVL 50%	TVL 75%	TVL 90%
fee 1%	$146,546	$3,124,815	$41,601,760	$439,265,200
fee 0.5%	$73,273	$1,562,407	$20,800,880	$219,632,600
fee 0.3%	$43,964	$937,444	$12,480,530	$131,779,600
fee 0.2%	$29,309	$624,963	$8,320,352	$87,853,050

Table 2: *Annual cash flow for variaous TVL scenarios and fees. Baseline case with 0.5% fees is highlighted.*

This includes allocations for the team, investors, community incentives and ecosystem growth to ensure long-term sustainability. By following this well-established approach, we aim to balance rewarding early contributors and investors while fostering community participation and providing resources for future development and innovation.

CCIP Bridge

Since the cash flow-generating contract, SF Options, is deployed on L2 chains to benefit from significantly lower gas fees, and the STFIN token resides on Ethereum mainnet, a mechanism is needed to transfer collected fees from L2 to Ethereum. For this purpose, we have chosen USDC, issued by Circle, as the collateral. USDC is currently the only stablecoin with a clear protocol for cross-chain transactions through its Cross-Chain Transfer Protocol (CCTP). Specifically, we will utilize Chainlink's version of CCTP so-called Cross Chain Interoperability Protocol (CCIP) for seamless interoperability.

The core idea of this protocol is that USDC is burned on the source chain with associated parameters, including the destination address, target chain, and an 'attestation' — a proof that USDC was burned for the intended cross-chain operation. On the target chain, Ethereum in our case, USDC is minted by presenting this 'attestation'.

This process requires the deployment of two smart contracts: SendPool on the source chain to handle USDC burning and ReceivePool on the target chain to mint USDC and forward it to the STFIN contract. This setup ensures efficient and secure cross-chain fee transfers, supporting the protocol's decentralized operations.

References

[1] OpenZeppelin. URL: https://docs.openzeppelin.com/contracts/5.x/erc1155.

[2] Solidity Team. URL: https://docs.soliditylang.org/en/latest/types.html#iterable-mappings.

[3] J. Hull. *Options, Futures and Other Derivatives*. Eastern economy edition. Pearson/Prentice Hall, 2009. ISBN: 9780136015864. URL: https://books.google.de/books?id=sEmQZoHoJCcC.

[4] Uniswap Labs, Adams, H. and Zinsmeister N., and Robinson D. *Uniswap v2 Core*. 2020.

[5] Solidity Team. URL: https://soliditylang.org/blog/2024/01/26/solidity-0.8.24-release-announcement/.

[6] OpenZeppelin. URL: https://docs.openzeppelin.com/contracts/5.x/api/utils.

[7] OpenZeppelin. URL: https://docs.openzeppelin.com/contracts/5.x/api/proxy.

[8] Harshit Verma. URL: https://blog.blockmagnates.com/understanding-divergence-loss-in-uniswap-a-step-by-step-guide-3c8474f0bafa.

[9] Pintail. URL: https://pintail.medium.com/uniswap-a-good-deal-for-liquidity-providers-104c0b6816f2.