KakiDex v1 Core

WHITEPAPER

September 2021

Abstract

This technical whitepaper explains some of the design decisions behind the Kaki core contracts. Kaki Exchange v1 is a noncustodial automated market maker intending to be implemented for multiple blockchains (Polygon Network, Binance Smart Chain, Ethereum Mainnet and more Layer 2 Scaling Solution Blockchains) - Kaki Dex decided to start on Polygon Network because of the small development costs and interoperability with the Ethereum Mainnet. Our contracts architecture for the exchange uses deep references of the Uniswap V2 Protocol. This whitepaper describes the mechanics of the core contracts including the pair contract that stores liquidity providers` funds—and the factory contract used to instantiate pair contracts.

1. INTRODUCTION

Kaki has the entire smart contract architecture ready to be deployed on any Ethereum Virtual Machine based blockchain. The first blockchain that hosts the Kaki AMM is on the Polygon blockchain, implementing an automated liquidity protocol based on a "constant product formula". We used UniswapV2 Pair, Router and Factory contracts as a reference because it is audited and battle-tested. Same like Uniswap, Kaki "pair stores pooled reserves of two assets, and provides liquidity for those two assets, maintaining the invariant that the product of the reserves cannot decrease". Traders pay a 30-basis-point fee on trades, which goes to liquidity providers. The contracts do not have any type of proxy which makes them immutable.

Kaki Dex allows the creation of arbitrary ERC20/ERC20 pairs, as well as ERC20/ETH pairs. Inherited from Uniswap too, a hardened price oracle that accumulates the relative price of the two assets at the beginning of each block. This allows other contracts on Polygon to estimate the time-weighted average price for the two assets over arbitrary intervals. Finally, it enables "flash swaps" where users can receive assets freely and use them elsewhere on the chain, only paying for (or returning) those assets at the end of the transaction. While the contract is not generally upgradeable, there is a private key that has the ability to update a variable on the factory contract to turn on an on-chain 5-basis-point fee on trades. This fee will initially be turned off, but could be turned on in the future, after which liquidity providers would earn 25 basis points on every trade, rather than 30 basis points.

The difference between Kaki Dex and Centralized exchanges like Binance, KuCoin or Coinbase, is that Kaki does not hold your coins in our custody, you have 100% control and ownership of your own assets.

This paper describes the mechanics of the core contract, as well as the factory contract used to instantiate those contracts. Actually, using Kaki will require calling the pair contract through a "router" contract that computes the trade or deposit amount and transfers funds to the pair contract.

2. FEATURES

2.1. PRICE ORACLE

The marginal price offered by Kaki (not including fees) at time *t* can be computed by dividing the reserves of asset a by the reserves of asset b.

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

Since arbitrageurs will trade with Kaki if this price is incorrect (by a sufficient amount to make up for the fee), the price offered by Kaki tends to track the relative market price of the assets. This means it can be used as an approximate price oracle. This oracle is measuring and recording the price before the first trade of each block (or equivalently, after the last trade of the previous block).

This price is more difficult to manipulate than prices during a block. If an attacker submits a transaction that attempts to manipulate the price at the end of a block, some other arbitrageur may be able to submit another transaction to trade back immediately afterward in the same block. A miner (or an attacker who uses enough gas to fill an entire block) could manipulate the price at the end of a block, but unless they mine the next block as well, they may not have a particular advantage in arbitraging the trade back.

Specifically, Kaki *accumulates* this price, by keeping track of the cumulative sum of prices at the beginning of each block in which someone interacts with the contract. Each price is weighted by the amount of time that has passed since the last block in which it was updated, according to the block timestamp. This means that the accumulator value at any given time (after being updated) should be the sum of the spot price at each second in the history of the contract.”

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

To estimate the *time-weighted average price* from time t_1 to t_2, an external caller can checkpoint the accumulator`s value at t_1 and then again at t_2, subtract the first value from the second, and divide by the number of seconds elapsed. (Note that the contract itself does not store historical values for this accumulator—the caller has to call the contract at the beginning of the period to read and store this value.)

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

Users of the oracle can choose when to start and end this period. Choosing a longer period makes it more expensive for an attacker to manipulate the TWAP, although it results in a less up-to-date price. One complication: should we measure the price of asset A in terms of asset B, or the price of asset B in terms of asset A? While the spot price of A in terms of B is always the reciprocal of the spot price of B in terms of A, the mean price of asset A in terms of asset B over a particular period of time is not equal to the reciprocal of the mean price of asset B in terms of asset A.

For example, if the USD/ETH price is 100 in block 1 and 300 in block 2, the average USD/ETH price will be 200 USD/ETH, but the average ETH/USD price will be 1/150 ETH/USD. Since the contract cannot know which of the two assets users would want to use as the unit of account, Kaki tracks both prices. Another complication is that it is possible for someone to send assets to the pair contract—and thus change its balances and marginal price - *without* interacting with it, and thus without triggering an oracle update. If the contract simply checked its own balances and updated the oracle based on the current price, an attacker could manipulate the oracle by sending an asset to the contract immediately before calling it for the first time in a block. If the last trade was in a block whose timestamp was X seconds ago, the contract would incorrectly multiply the new price by X before accumulating it, even though nobody has had an opportunity to trade at that price. To prevent this, the core contract caches its reserves after each interaction, and updates the oracle using the price derived from the cached reserves rather than the current reserves.

2.1.1 PRECISION

Because Solidity does not have first-class support for non-integer numeric data types, the Kaki uses a simple binary fixed point format to encode and manipulate prices. Specifically, prices at a given moment are stored as UQ112.112 numbers, meaning that 112 fractional bits of precision are specified on either side of the decimal point, with no sign.

These numbers have a range of $[0, 2^{112} - 1]^4$ and a precision of $1/2^{112}$.

The UQ112.112 format was chosen for a pragmatic reason — because these numbers can be stored in a uint224, this leaves 32 bits of a 256 bit storage slot free. It also happens that the reserves, each stored in a uint112, also leave 32 bits free in a (packed) 256 bit storage slot. These free spaces are used for the accumulation process described above. Specifically, the reserves are stored alongside the timestamp of the most recent block with at least one trade, modded with 2^{32} so that it fits into 32 bits. Additionally, although the price at any given moment (stored as a UQ112.112 number) is guaranteed to fit in 224 bits, the accumulation of this price over an interval is not. The extra 32 bits on the end of the storage slots for the accumulated price of A/B and B/A are used to store overflow bits resulting from repeated summations of prices. This design means that the price oracle only adds an additional three SSTORE operations (a current cost of about 15,000 gas) to the first trade in each block. The primary downside is that 32 bits isn`t quite enough to store timestamp values that will reasonably never overflow. In fact, the date when the Unix timestamp overflows a uint32 is 02/07/2106. To ensure that this system continues to function properly after this date, and every multiple of $2^{32} - 1$ seconds thereafter, oracles are simply required to check prices at least once per interval (approximately 136 years). This is because the core method of accumulation (and modding of timestamp), is actually overflow-safe, meaning that trades across overflow intervals can be appropriately accounted for given that oracles are using the proper (simple) overflow arithmetic to compute deltas.

2.2. ERC-20 PAIRS

Kaki using MATIC as a bridge currency case: Every pair included MATIC as one of its assets. The same strategy will be followed on each blockchain. Native assets will be used as pair roots. This makes routing simpler—every trade between ABC and XYZ goes through the MATIC/ABC pair and the MATIC/XYZ pair—and reduces fragmentation of liquidity. However, this rule imposes significant costs on liquidity providers. All liquidity providers have exposure to MATC, and suffer impermanent loss based on changes in the prices of other assets relative to MATIC. When two assets ABC and XYZ are correlated—for example, if they are both USD stablecoins—liquidity providers on a Kaki pair ABC/XYZ would generally be subject to less impermanent loss than the ABC/MATIC or XYZ/MATIC pairs. Using MATIC as a mandatory bridge currency also imposes costs on traders. Traders have to pay twice as much in fees as they would on a direct ABC/XYZ pair, and they suffer slippage twice. Kaki allows liquidity providers to create pair contracts for any two ERC-20s. A proliferation of pairs between arbitrary ERC-20s could make it somewhat more difficult to find the best path to trade a particular pair, but routing can be handled at a higher layer (either off-chain or through an on-chain router or aggregator).

2.3. FLASH SWAPS

Kaki allows a user to receive and use an asset before paying for it, as long as they make the payment within the same atomic transaction. The swap function makes a call to an optional user-specified callback contract in between transferring out the tokens requested by the user and enforcing the invariant. Once the callback is complete, the contract checks the new balances and confirms that the invariant is satisfied (after adjusting for fees on the amounts paid in). If the contract does not have sufficient funds, it reverts the entire transaction. A user can also repay the Kaki pool using the same token, rather than completing the swap. This is effectively the same as letting anyone flash-borrow any of the assets stored in a Kaki pool (for the same 0.30% fee as Kaki charges for trading).

2.4. PROTOCOL FEE

Kaki includes a 0.05% protocol fee that can be turned on and off. If turned on, this fee would be sent to a **feeTo** address specified in the factory contract.

Initially, **feeTo** is not set, and no fee is collected. A pre-specified address—**feeToSetter**—can call the **setFeeTo** function on the Kaki factory contract, setting **feeTo** to a different value.

feeToSetter can also call the **setFeeToSetter** to change the **feeToSetter** address itself.

If the **feeTo** address is set, the protocol will begin charging a 5-basis-point fee, which is taken as a 1 6 cut of the 30-basis-point fees earned by liquidity providers. That is, traders will continue to pay a 0.30% fee on all trades; 83.3% of that fee (0.25% of the amount traded) will go to liquidity providers, and 16.6% of that fee (0.05% of the amount traded) will go to the **feeTo** address. Collecting this 0.05% fee at the time of the trade would impose an additional gas cost on every trade. To avoid this, accumulated fees are collected only when liquidity is deposited or withdrawn. The contract computes the accumulated fees, and mints new liquidity tokens to the fee beneficiary, immediately before any tokens are minted or burned.

The **feeTo** address will be in our case the Staking Pit contract. For every swap on the exchange on MATIC, 0.05% of the swap fees are distributed as KAKI proportional to your share of the Pit. When your KAKI is staked into the Pit, you receive xKAKI in return for a fully composable token that can interact with other protocols. Your xKAKI is continuously compounding, when you unstake you will receive all the originally deposited KAKI and any additional from fees.

The total collected fees can be computed by measuring the growth in √ k (that is, √ (x · y)) since the last time fees were collected. This formula gives you the accumulated fees between t_1 and t_2 as a percentage of the liquidity in the pool at $t_{2:}$

$$f_{1,2} = 1 - \frac{\sqrt{k_1}}{\sqrt{k_2}}$$

If the fee was activated before t_1, the **feeTo** address should capture 1 6 of fees that were accumulated between t_1 and t_2. Therefore, we want to mint new liquidity tokens to the **feeTo** address that represent ϕ · $f_{1,2}$ of the pool, where ϕ is 1/6. That is, we want to choose s_m to satisfy the following relationship, where s_1 is the total quantity of outstanding shares at time t_1:

$$\frac{s_m}{s_m + s_1} = \phi \cdot f_{1,2}$$

After some manipulation, including substituting 1 − √$k_{1/}$ √k_2 for $f_{1,2}$ and solving for s_m, we can rewrite this as:

$$s_m = \frac{\sqrt{k_2} - \sqrt{k_1}}{(\frac{1}{\phi} - 1) \cdot \sqrt{k_2} + \sqrt{k_1}} \cdot s_1$$

Setting ϕ to 1/6 gives us the following formula:

$$s_m = \frac{\sqrt{k_2} - \sqrt{k_1}}{5 \cdot \sqrt{k_2} + \sqrt{k_1}} \cdot s_1$$

Suppose the initial depositor puts 100 DAI and 1 ETH into a pair, receiving 10 shares. Some time later (without any other depositor having participated in that pair), they attempt to withdraw it, at a time when the pair has 96 DAI and 1.5 ETH. Plugging those values into the above formula gives us the following:

$$s_m = \frac{\sqrt{1.5 \cdot 96} - \sqrt{1 \cdot 100}}{5 \cdot \sqrt{1.5 \cdot 96} + \sqrt{1 \cdot 100}} \cdot 10 \approx 0.0286$$

2.5. META TRANSACTIONS FOR POOL SHARE

The Liquidity Pool tokens minted by Kaki pairs natively support meta transactions. This means users can authorize a transfer of their pool shares with a signature, rather than an on-chain transaction from their address. Anyone can submit this signature on the user's behalf by calling the *permit* function, paying gas fees and possibly performing other actions in the same transaction.

3. DESIGN

3.1. SOLIDITY

Kaki is implemented in Solidity and developed on the basis of audited open-source protocols with a twist. Solidity provides more infrastructure and documentation in terms of implementation strategies than other languages for smart contracts.

3.2. CONTRACT ARCHITECTURE

The main functionality of the Kaki protocol is the exchange, liquidity pools, staking and liquidity farming. Those components were achieved by forking the smart contract of one of the biggest protocols on the Ethereum Mainnet, Uniswap and Sushiswap. Using the same descriptions for certain parts is because they come from the Uniswap Whitepaper. This usage of an already tested system provides us with more security and a longer testing time than the team could ever achieve. Furthermore, after participating and observing the two top projects, along with many other smaller ones, the tokenomics along with the pools will have a unique design, but keeping the security that the UniswapV2 contracts provide.

Any bugs in this contract could be disastrous, since millions of dollars of liquidity might be stolen or frozen. Kaki development team is constantly looking for mitigating the risks and also organizes bounty campaigns for any kind of bugs.

When evaluating the security of this core contract, the most important question is whether it protects *liquidity providers* from having their assets stolen or locked. Any feature that is meant to support or protect *traders*—other than the basic functionality of allowing one asset in the pool to be swapped for another—can be handled in a "router" contract. In fact, even part of the swap functionality can be pulled out into the router contract.

In Kaki, the seller sends the asset to the core contract *before* calling the swap function. Then, the contract measures how much of the asset it has received, by comparing the last recorded balance to its current balance. This means the core contract is agnostic to the way in which the trader transfers the asset. Instead of **transferFrom**, it could be a meta transaction, or any other future mechanism for authorizing the transfer of ERC-20s.

3.2.1. SYNC() AND SKIM()

To protect against bespoke token implementations that can update the pair contract`s balance, and to more gracefully handle tokens whose total supply can be greater than 2^{112}, Kaki has two bail-out functions: **sync()**and **skim()**.

sync() functions as a recovery mechanism in the case that a token asynchronously deflates the balance of a pair. In this case, trades will receive sub-optimal rates, and if no liquidity provider is willing to rectify the situation, the pair is stuck. **sync()** exists to set the reserves of the contract to the current balances, providing a somewhat graceful recovery from this situation.

skim() functions as a recovery mechanism in case enough tokens are sent to a pair to overflow the two uint112 storage slots for reserves, which could otherwise cause trades to fail. **skim()** allows a user to withdraw the difference between the current balance of the pair and 2^{112} - 1 to the caller, if that difference is greater than 0.

3.3. HANDLING NON-STANDARD AND UNUSUAL TOKENS

The ERC-20 standard requires that transfer() and transferFrom() return a boolean in dictating the success or failure of the call. The implementations of one or both of these functions on some tokens—including popular ones like Tether (USDT) and Binance Coin (BNB)—instead have no return value.

Kaki handles non-standard implementations differently. Specifically, if a transfer() call has no return value, Kaki interprets it as a success rather than as a failure. This change should not affect any ERC-20 tokens that conform to the standard (because in those tokens, transfer() always has a return value).

The current protocol includes a "lock" that directly prevents reentrancy to all public statechanging functions. This also protects against reentrancy from the user-specified callback in a flash swap, as described in section 2.3.

3.4. INITIALIZATION OF LIQUIDITY TOKEN SUPPLY

Case: the value of the minimum quantity of liquidity pool shares (1e-18 pool shares) is worth so much that it becomes infeasible for small liquidity providers to provide any liquidity.

To mitigate this, Kaki burns the first 1e-15 (0.000000000000001) pool shares that are minted (1000 times the minimum quantity of pool shares), sending them to the zero address instead of to the minter. This should be a negligible cost for almost any token pair. But it dramatically increases the cost of the above attack. In order to raise the value of a liquidity pool share to $100, the attacker would need to donate $100,000 to the pool, which would be permanently locked up as liquidity.

3.5. DETERMINISTIC PAIR ADDRESSES

Kaki uses Ethereum`s CREATE2 opcode to generate a pair contract with a deterministic address. This means that it is possible to calculate a pair`s address (if it exists) off-chain, without having to look at the chain state.

3.7. MAXIMUM TOKEN BALANCE

In order to efficiently implement the oracle mechanism, Kaki only supports reserve balances of up to $2^{112} - 1$. This number is high enough to support 18-decimal-place tokens with a totalSupply over 1 quadrillion.

If either reserve balance does go above $2^{112} - 1$, any call to the swap function will begin to fail (due to a check in the _update() function). To recover from this situation, any user can call the skim() function to remove excess assets from the liquidity pool.

References

[1] Hayden Adams. 2018. url: https://hackmd.io/@477aQ9OrQTCbVR3fq1Qzxg/HJ9jLsfTz?

type=view.
[2] Guillermo Angeris et al. An analysis of Uniswap markets. 2019. arXiv: 1911.03380 [q-fin.TR].
[3] samczsun. Taking undercollateralized loans for fun and for profit. Sept. 2019. url: https : / / samczsun . com / taking - undercollateralized - loans - for - fun - and - for-profit/.
[4] Fabian Vogelsteller and Vitalik Buterin. Nov. 2015. url: https://eips.ethereum.org/EIPS/eip-20.
[5] Jordi Baylina Jacques Dafflon and Thomas Shababi. EIP 777: ERC777 Token Standard. Nov. 2017. url: https://eips.ethereum.org/EIPS/eip-777.
[6] Radar. WTF is WETH? url: https://weth.io/.
[7] Uniswap.info. Wrapped Ether (WETH). url: https://uniswap.info/token/0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2.
[8] Vitalik Buterin. EIP 1014: Skinny CREATE2. Apr. 2018. url: https://eips.ethereum.org/EIPS/eip-1014.
[9] UniswapV2Core Whitepaper. url: https://uniswap.org/whitepaper.pdf

DISCLAIMER

This paper is for general information purposes only. It does not constitute investment advice or a recommendation or solicitation to buy or sell any investment and should not be used in the evaluation of the merits of making any investment decision. It should not be relied upon for accounting, legal or tax advice or investment recommendations. This paper reflects current opinions of the authors. The opinions reflected herein are subject to change without being updated.