OniSwap

Oracle-parametrized liquidity on Solana:
how much a lying keeper can take, and where the pool earns

The OniSwap contributors

Draft 0.3, November 2026

Abstract

Proprietary AMMs now fill more than half of SOL/USDC volume on Solana. They reprice from an off-chain price, and nobody outside the operating firm can fund them. OniSwap is a public pool that reprices the same way, built on the oracle-parametrized curve $(Ox)^\lambda + y^\lambda = C$ of Amini and Feinstein and running on Solana mainnet since 18 November 2026. A keeper posts the price, and each swap checks it against Pyth's on-chain feed inside a band β. Most of this paper is about one question: how much can a keeper who lies inside that band take? The first-order answer, a fee above $\lambda\beta$, is wrong for large trades. We prove that the part of the fee the pool keeps must cover the whole band, by exhibiting a potential that no swap of any size can lower; under that condition a lying keeper takes a one-time amount and then nothing. An exact computation over every trade sequence agrees with our first-order bound to within 0.25% and, below the condition, finds the pumps that first-order analysis misses. The band is set by the oracle. Pyth keeps push feeds live on Solana for only five of the 22 assets we checked, but a keeper can post signed updates itself for 20,475 lamports per pair, which lowers the fee floor from 20 bp to between 6 and 12 bp. Replaying four weeks of one-second prices for six crypto assets and seventeen sessions of one-minute prices for ten US stocks, oracle-pool LPs earn 1.5 to 7.5 times what constant-product LPs earn at the same price for traders. Against the competition that actually exists, measured from 1,144 live Jupiter round trips and each token's real public-pool flow, the picture is narrower and, we think, more useful: a small pool wins most of the flow in thin markets such as JTO and earns tens of percent a year, roughly breaks even on tokenized stocks where a constant-product pool of the same size loses up to 8%, and wins almost nothing on SOL, JUP or TRUMP.

1 Introduction

An automated market maker quotes from its own inventory. When the price moves somewhere deeper, the pool keeps quoting the old one until an arbitrageur trades it back, and the arbitrageur's profit is what the liquidity providers (LPs) lose. Milionis et al. called this loss-versus-rebalancing and showed that it accrues at $\frac{1}{8}\sigma^2 V$ for a constant-product pool of value V [2]. Retail flow makes it worse, because every retail trade pushes the pool off the market and someone collects the gap on the way back.

Solana's market makers solved this for themselves by closing the pool. HumidiFi, SolFi, Tessera V and BisonFi each quote from one firm's inventory and reprice with a small on-chain update, and by 2026 they fill more than half of SOL/USDC volume [4]. They work well. They also leave the ordinary depositor with the stale public pools.

Amini and Feinstein recently gave the public alternative a theory [1]. Their oracle-parametrized AMM quotes a price that depends on both its reserves and an oracle, is deeper near the market than a constant-product pool when the oracle is good, and loses less to arbitrage, while a noisy or stale oracle undoes both gains. The paper is written with tokenized securities in mind and stops short of an implementation.

We built one. The obvious way to feed such a pool on Solana is a keeper that posts prices, checked against Pyth so that it cannot post anything it likes. That design raises a question the theory does not answer, because in the theory the oracle is noisy but not adversarial: what can a keeper do if it lies, but only inside the band the check allows? Our first answer was a first-order bound with the fee condition $f > \lambda\beta$, the same inequality as Amini and Feinstein's protection against a sandwiched update. It is not enough. Large round trips, bought at one edge of the band and sold at the other, pump the pool at fees all the way up to the band itself, and the fix is to require that the fee the pool keeps covers the band. We prove that this suffices for trades of every size (Section 4).

Once the fee has to cover the band, the economics of the design are decided by the oracle, so we measured it on mainnet (Section 5), then asked two separate questions. Is the mechanism better for LPs than a public constant-product pool at the same price for traders (Section 6)? And would a pool built this way actually win trades against what Solana offers today (Section 7)? The answers are yes and sometimes. In short:

- a potential-function proof that a lying keeper cannot pump the pool once the retained fee covers the band, an exact computation over all trade sequences that matches our first-order bound, and the finding that the first-order fee condition fails for large trades;
- a fallback rule that looks safe and is a drain, found in our own first build;
- measurements of Pyth on Solana: which push feeds are live, how confident they are, and what refreshing them costs;
- LP results on sixteen assets at matched prices, and a market test against Jupiter's live routes using each token's real public flow.

2 Background

We use Amini and Feinstein's notation [1]. A pool holds x of a risky asset and y of a numéraire, $r = y/x$, and π is the oracle's log-price. An oracle-parametrized AMM is a quoted log-price $p(r, \pi)$, strictly increasing and onto in r, with $0 \le \partial_\pi p \le 1$; with the oracle held fixed it is an ordinary AMM. In the geometric-average family on a constant-product base, $p = \lambda\pi + (1 - \lambda)\log r$, so the marginal price is $m = O^\lambda (y/x)^{1-\lambda}$ with $O = e^\pi$ and the trading curves are level sets of

$$F(x, y; O) = (Ox)^\lambda + y^\lambda \tag{1}$$

(their Example 2.9). At $\lambda \to 0$ this is constant product and at $\lambda \to 1$ it quotes the oracle; an oracle move shifts the quote by λ times the move, with no trade needed.

Three of their results matter here. With an accurate oracle the pool is $1/(1 - \lambda)$ times as deep as an equally funded constant-product pool near the market (Example 2.11). Under a perfect oracle its loss to rebalancing per unit of value is $(1 - \lambda)$ times constant product's, and it stays ahead only while the oracle error's volatility is below $\sqrt{(1 - \lambda)/\lambda}\,\sigma$ (Example 3.6). And with a fee γ, $\Gamma = -\log(1 - \gamma)$, front-running one oracle update gains nothing beyond the ordinary arbitrage after it when $\lambda|\Delta\pi| \le \Gamma$ (Lemma 3.14). That last condition is the one that does not carry over to a keeper who can move the price back and forth.

On Solana, Jito's Block Assembly Marketplace offers a Maker Priority Plugin that runs an enrolled price update first in each 20 ms batch on BAM slots (40.5% of stake when we looked). OniSwap's update has the shape it requires. Nothing below depends on it.

3 Design

Pools and hooks. OniSwap is a hooks AMM. A pool fixes at creation its two mints, its base fee, optionally a *hook* (a separate program called before and after each swap that may raise the

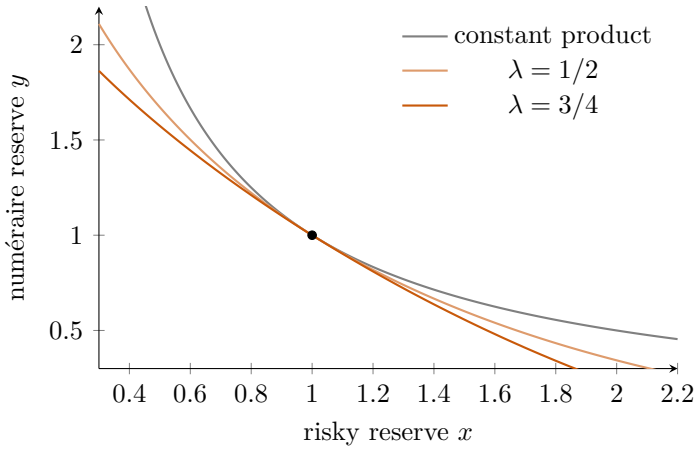


Figure 1: Trading curves through one balanced state at $O = 1$. The oracle curves are flatter near the market price, $1/(1-\lambda)$ times as deep there, and bend harder away from it.

fee), and, for an oracle pool, its price feed and λ. All of these go into the pool's address, so nothing can re-point a pool and a hooked pool never collides with a plain one. The pool hands a hook one account, the hook's own state, writable and unsigned, through a plain `invoke`: no vault, no vault authority, no token program, nothing of the swapper's. A hostile hook we wrote for the purpose failed all 17 of its attacks (writing pool state, moving vault tokens, re-entering the pool, looping forever); handed the vault and its signer instead, the same hook took half the vault, which is why the shape is fixed. A hook call costs about 2,000 compute units. Deposits, withdrawals, claims and closure never call it, so the worst a hook can do is stop swaps in its own pool, and the application shows each hook's program id, its live upgrade authority next to the one recorded at the pool's creation, and its payee.

The curve in integers. With a, b the two reserves and O the feed price in raw b per raw a, an oracle pool trades on $(Oa)^\lambda + b^\lambda = C$ with λ equal to $1/2$ or $3/4$. C is never stored; each swap evaluates it at the current price, which is what lets an update reprice the pool. Both terms are carried in raw b scaled by 10^{12}. For $\lambda = 1/2$ we take integer square roots and invert exactly. For $\lambda = 3/4$ we compute $z^{3/4} = \sqrt{z} \cdot z^{1/4}$ after lifting z by a power of 2^4 toward 2^{126}, which keeps about 62 significant bits, and invert with a 42-bit cube root stepped upward until the rounded power clears its target. Every rounding goes the pool's way, and the swap checks that C after the trade, rounded down, is at least C before it, rounded up. Starting the roots from the root of the top 64 bits and finishing with two or three 128-bit Newton steps took a $\lambda = 3/4$ swap from 80,093 to about 48,000 compute units.

Fees. The fee is charged on the input. The protocol's share and, on hooked pools, the hook's share are set aside and leave the reserves; the rest stays. On mainnet today the shares are 10% and 5%. Section 4.6 explains why it is the retained part that has to cover the band, and that is what pool creation now checks:

$$f\,(1-\sigma) \geq \beta, \tag{2}$$

with f the base fee, σ the protocol share plus the hook share if the pool has a hook, and β the feed's band, all in basis points. A hook may raise a swap's fee but never take it below the base fee.

Feed, keeper and guard. A `PriceFeed` account names a keeper, the two mints, one or two Pyth accounts and three limits: the band, the keeper's staleness in slots, and Pyth's age in seconds. Only the keeper updates it, with an increasing sequence number, in a 279-byte,

3,198-unit transaction. A swap must pass exactly the feed and Pyth accounts the pool names, and goes through only if the keeper's price is fresh, every Pyth price is fully verified, positive and within its age limit, the two Pyth confidence intervals (each in basis points of its own price) sum to at most the band, and the keeper's price is within the band of Pyth's. Each Pyth account must be the push-feed address derived from its own feed id, which Pyth's push oracle only ever moves forward in time. When any check fails the swap is refused; Section 4.2 shows why refusing is the only safe choice.

Pyth pays for updates to some push feeds itself. Anyone may post a signed update to the same account, verified on chain against Wormhole guardian signatures, and only a newer price is accepted. A keeper can therefore refresh the guard together with its own price. It still cannot move Pyth, only choose when to bring it up to date, and the band then needs to cover Pyth's confidence rather than the gap between a fresh keeper and a minute-old Pyth. Section 5 prices both options.

Liquidity. Deposits and withdrawals are proportional, never read the feed and never call a hook, so an LP can always leave. Every account the program creates can be closed for its rent.

4 Security

Let P be the true price, f the fee, $g = 1 - f$ and $\phi = -\ln g$. A keeper who lies may post any price O in the window $[Ge^{-\beta}, Ge^{\beta}]$ around Pyth's price $G = Pe^{c}$, where c is how far Pyth is from the truth. The attacker owns the keeper, chooses O before every trade, and trades any size in any order. Since only the attacker trades, its profit is exactly what the pool loses at the true price, $W = Pa + b$.

4.1 At a fixed price nothing can be taken

Proposition 1. *While the feed price stays put, any sequence of swaps that returns reserve a to where it started leaves reserve b no lower.*

Proof. Each swap leaves $(Oa)^{\lambda} + b^{\lambda}$ at least where it was: the curve step preserves it with rounding in the pool's favour, checked on chain, and the retained fee only adds to the input reserve. With a back where it started, $b_{\text{end}}^{\lambda} \geq b_{\text{start}}^{\lambda}$. □

4.2 Falling back to constant product is a drain

Our first build priced a swap as constant product whenever the guard failed. It looked conservative. It is a drain, because the two curves meet at the current reserves with the same slope and different curvature: take $P = 1$ and $a_0 = b_0 = 1$, buy along the oracle curve to $a_1^{\lambda} + b_1^{\lambda} = 2$, so that $a_1 b_1 \leq 1$ by the inequality of means, wait for the guard to fail, and sell back along constant product to a_0, which leaves $b_2 = a_1 b_1$. The gain $1 - a_1 b_1$ grows faster with size than the fee. No dishonest keeper is needed, since Pyth's sponsored SOL/USD feed ages past its limit at a predictable second. On a local validator the unfixed program paid a bot $43,905.83 for a $200,000 buy on a $1M pool. A failing guard now refuses the swap.

4.3 A first-order bound

Write $x = \ln(b/(aP))$ for how far the reserves' own price is from P and $y = \ln(O/P)$ for the keeper's offset, so that $\ln(m/P) = \lambda y + (1-\lambda)x$, and the guard admits $y \in [c - \beta, c + \beta]$. Near balance $V/4$ dollars of trade move x by one unit, and the attacker's total is

$$G_{\text{att}} = \frac{V}{4}\Big[-\lambda \int y\, dx - \phi \int |dx| - \frac{1-\lambda}{2}\big(x_{\text{end}}^2 - x_0^2\big)\Big]. \tag{3}$$

Splitting off the net move (Appendix A) gives the following.

Proposition 2. *To first order in the price gaps, if $\phi > \lambda\beta$ every round trip beyond the net move loses money, and the attacker takes at most*

$$G^* = \frac{VD^2}{8(1-\lambda)}, \qquad D = \max(0,\, D_{buy},\, D_{sell}), \tag{4}$$

with $D_{buy} = -\phi + \lambda(\beta - c) - (1-\lambda)x_0$ and $D_{sell} = -\phi + \lambda(\beta + c) + (1-\lambda)x_0$, of which at most one is positive.

For $\lambda \to 0$, $VD^2/8$ is what an arbitrageur takes from a constant-product pool after a jump of D, so a lying keeper takes what an arbitrageur would after a jump, scaled by $1/(1-\lambda)$. This is the bound we started with, and its condition $\phi > \lambda\beta$ is where we went wrong.

4.4 The fee the pool keeps must cover the band

The first-order picture sees only small trades. A trade that is large next to the reserve it draws on behaves differently, and the right object to look at is the pool's curve evaluated at Pyth's price rather than at the keeper's.

Theorem 1. *Let a swap at curve price O keep a fraction ρ of its input in the reserves, with g of the input entering the curve ($g \le \rho \le 1$), and suppose $\ln(\rho/g) \ge \beta$. Then*

$$H(a,b) = (Ga)^\lambda + b^\lambda \tag{5}$$

does not decrease under any swap, of any size and in either direction, at any $O \in [Ge^{-\beta}, Ge^{\beta}]$.

The proof (Appendix A) reduces both directions to one inequality in a single variable. With no share, $\rho = 1$ and the condition reads $\phi \ge \beta$; with a share σ of the fee paid out, $\rho = 1 - \sigma f$ and rule (2) implies it, because then $f \ge \beta$ and $\rho/g = 1 + f(1-\sigma)/(1-f) \ge 1/(1-\beta) \ge e^{\beta}$. The engine's integer rounding (fee rounded up, shares rounded down) only raises ρ/g.

Corollary 1. *Under the condition of Theorem 1, no sequence of trades takes more from the pool than*

$$W_0 - \min\{Pa + b : H(a,b) \ge H_0\} \;=\; W_0 - H_0^{1/\lambda}\left(1 + (P/G)^{\frac{\lambda}{\lambda-1}}\right)^{\frac{\lambda-1}{\lambda}}, \tag{6}$$

a fixed amount determined by the starting state and Pyth's offset. In particular, no pump exists.

The bound in the corollary is what a frictionless arbitrageur could take from a pool priced at Pyth, which is generous: it ignores the fees the attacker pays on the way. To see how tight things really are we computed the attacker's exact optimum.

4.5 The exact worst case

Two facts make the set of states an attacker can reach computable (both proved in Appendix A). From a given state, the reserve b' reached at any target a' is increasing in the starting b, for buys and sells alike, so only the lowest reachable b at each a matters. And the attacker's best price is always an edge of the window, the low edge for buying and the high edge for selling. The lowest reachable b as a function of a is then the fixed point of relaxing every state against every other with two moves, and a full pass that changes nothing certifies that no longer sequence of trades does better. We run it on a grid of reserves, dense (log step 2×10^{-5}) within 3% of the start and log-spaced out to a thousandth and a thousand times the starting reserve. Confining the attacker to grid states can only lower its result; refining the grid changed nothing at the precision shown.

Band	λ	Pool state	Exact optimum	First-order G^*	Corollary 1
10 bp	1/2	balanced	0.00	0.00	0.00
10 bp	1/2	after a rise	0.06	0.06	0.25
10 bp	1/2	after a rise, Pyth 50 bp late	2.25	2.25	3.06
10 bp	3/4	balanced	0.00	0.00	0.00
10 bp	3/4	after a rise	0.28	0.28	0.50
10 bp	3/4	after a rise, Pyth 50 bp late	10.13	10.12	11.28
20 bp	1/2	balanced	0.00	0.00	0.00
20 bp	1/2	after a rise	0.25	0.25	1.00
20 bp	1/2	after a rise, Pyth 50 bp late	3.06	3.06	5.06
20 bp	3/4	balanced	0.00	0.00	0.00
20 bp	3/4	after a rise	1.12	1.12	2.00
20 bp	3/4	after a rise, Pyth 50 bp late	13.78	13.77	16.53
50 bp	1/2	balanced	0.00	0.00	0.00
50 bp	1/2	after a rise	1.55	1.55	6.25
50 bp	1/2	after a rise, Pyth 50 bp late	6.23	6.22	14.06
50 bp	3/4	balanced	0.00	0.00	0.00
50 bp	3/4	after a rise	7.00	6.98	12.50
50 bp	3/4	after a rise, Pyth 50 bp late	28.10	28.03	38.28

Table 1: Dollars a keeper-controlling attacker can take from a $1M pool whose fee equals the band. "After a rise" is where arbitrage leaves the pool after the price goes up; "Pyth late" puts Pyth 50 bp behind the truth. Every exact value was certified by a pass with no change.

Table 1 has the three numbers side by side. The exact optimum and the first-order bound agree to within 0.25% in all eighteen cases, and the corollary's bound sits above both, as it must. We also checked the theorem directly on 1,129,259 random swaps (random λ, band from 1 to 1,000 bp, sizes from a millionth to a million times the reserve, any price in the window) at $\ln(\rho/g) = \beta$ exactly: the largest relative drop in H was 5.7×10^{-16}, which is floating-point noise.

Below the condition the same computation finds what the first-order analysis cannot. Table 2 gives the lowest fee at which the exact optimum stays finite, found by bisection, and Figure 2 shows what the attacker takes on either side of it.

Band	$\lambda = 1/2$, first order	exact edge	$\lambda = 3/4$, first order	exact edge
5 bp	2.50 bp	4.87 bp (97.4%)	3.75 bp	4.95 bp (99.0%)
10 bp	5.00 bp	9.74 bp (97.4%)	7.50 bp	9.90 bp (99.0%)
20 bp	10.00 bp	19.47 bp (97.4%)	15.00 bp	19.79 bp (99.0%)
50 bp	25.00 bp	48.60 bp (97.2%)	37.50 bp	49.57 bp (99.1%)

Table 2: The lowest fee at which no pump exists, with trades up to 100 times the reserves. Allowing up to 1,000 times moves the edge to 99.0% ($\lambda = 1/2$) and 99.5% ($\lambda = 3/4$) of a 20 bp band, toward the band itself, as Theorem 1 predicts.

The practical lesson took us a while to accept: λ buys depth, not a lower fee floor. The fee is set by the band, and the band by how fresh and how confident the oracle is.

4.6 Shares, and the rule on mainnet

A protocol or hook share is paid by the attacker like the rest of the fee, so it makes a round trip more expensive for the attacker (for $\lambda = 1/2$ and a 20 bp band, the fee at which one large round trip stops paying falls from 15.9 bp with no share to 13.5 bp with a 30% share). It does not protect the LPs, though, because it leaves the reserves, and Theorem 1 needs the retained fraction. The program deployed on 18 November checks the gross fee, $f \geq \beta$, which with the

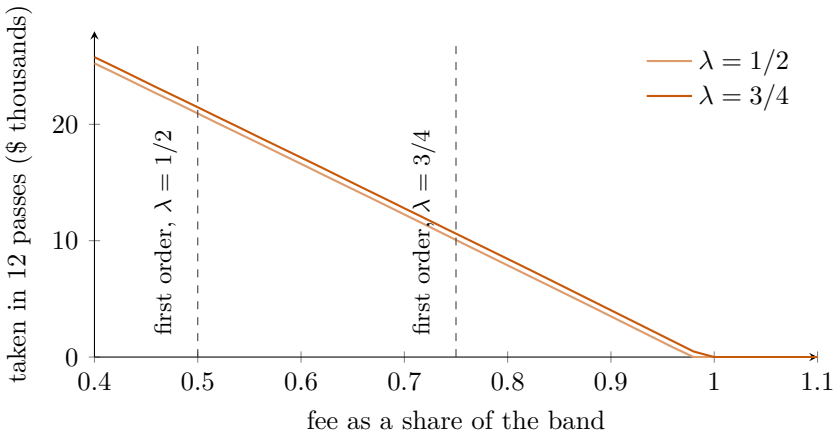


Figure 2: What a keeper-controlling attacker takes from a balanced $1M pool in twelve relaxation passes (band 20 bp, trades up to a thousand times the reserves). Below the edge the take grows with every pass, above it the take is zero, and both first-order thresholds lie well inside the pumping region.

live 10% share leaves $\ln(\rho/g)$ at about 0.9β. Rule (2) is in the current build and its test suite, and goes to mainnet with the next upgrade; until then the application refuses to open an oracle pool that breaks it.

Deposits and withdrawals scale both reserves, leave x unchanged and never read the feed. Unlike constant product, the geometric-average curve reaches an empty reserve at finite cost ([1], Remark 3.13), and the engine refuses any trade that would empty one.

5 The oracle on mainnet

We read the push-feed account of each of the 22 assets in this study from mainnet on 18 November 2026. Five were live (SOL, USDC, JUP, JTO and PYTH), updated every 52 seconds or so and never more than 60 seconds apart. The other 17 were stale: WIF, BONK, W, TRUMP and FARTCOIN by about a day, RAY by 55 days and PENGU by 91, and all ten US equities by 68 to 96 days. A pool guarded by a stale feed does not trade until someone posts an update.

Guarding on Pyth's own minute-long rhythm costs a wide band. Over a calm hour of mainnet samples a 20 bp band with a 90-second age limit let a pool trade 99.9% of the time and a 10 bp band 95.7%. Replaying a volatile week, with Pyth modelled from those samples, a 20 bp band traded 97.3% of the time, the pauses cost LPs $414 per $1M, and a keeper lying every second inside the band cost them about 0.3% a week on the $\lambda = 3/4$ curve. A 20 bp floor is a long way above what Solana's prop AMMs charge.

Refreshing Pyth costs less than we expected. We traced the transactions that update the sponsored feeds: writing and verifying the Wormhole-signed message, updating the feeds and closing the temporary account all happen in one transaction, at 17,475 lamports for one feed, 20,475 for two and 35,475 for seven (140,000 to 420,000 compute units). A keeper that posts its own price together with a fresh signed Pyth update keeps the guard seconds old, so the gap between keeper and guard is zero at each post and the band only has to clear the confidence check. Table 3 shows that check, computed the way the engine computes it, over 1,146 samples taken every two seconds across 0.6 hours.

So a refreshed guard puts the floor at the oracle's confidence, at most 6 bp for SOL/USDC and 9 to 12 bp for PYTH, JTO and JUP against USDC. We use bands of 10 bp for SOL and 15 bp for the other crypto assets. For the stocks, whose feeds are not live on Solana, we have a single confidence reading each: 0.7 to 7.8 bp for SPY, QQQ, AAPL, NVDA, GOOGL and TSLA, and 10 to 29 bp for MSTR, COIN, CRCL and HOOD (USDC's adds 1 to 3 bp). We use 10 and 20 bp. In our SOL replay a keeper that posted on every 5 bp move or at least every

Guard pair	Confidence sum, median	99th pct	max	Pyth age, median	max
SOL/USDC	5 bp	6 bp	6 bp	28 s	58 s
JUP/USDC	9 bp	12 bp	12 bp	28 s	59 s
JTO/USDC	9 bp	10 bp	10 bp	27 s	57 s
PYTH/USDC	5 bp	9 bp	9 bp	28 s	59 s

Table 3: The guard's confidence check on mainnet's live push feeds (each interval rounded up to whole basis points, then summed).

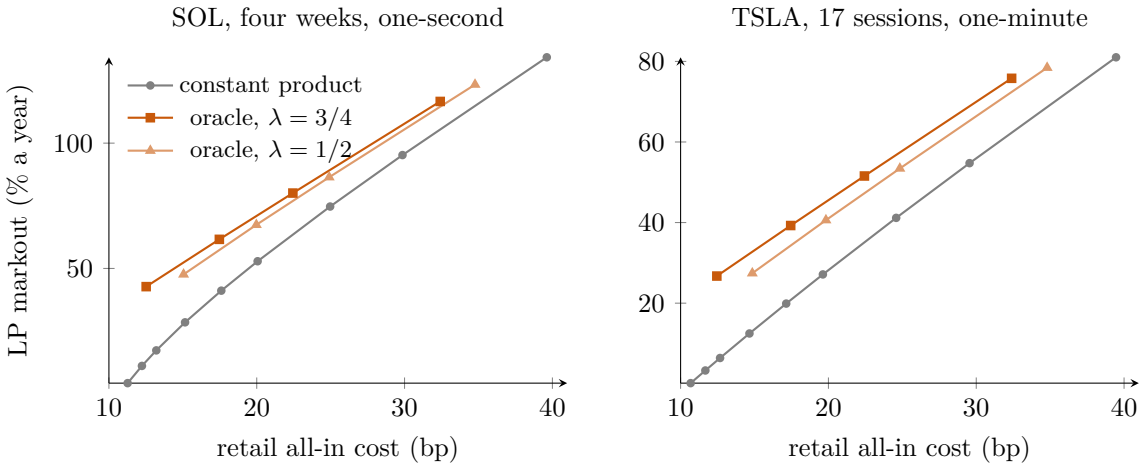


Figure 3: LP profit against the price traders pay, with daily volume equal to the pool. Constant product at fees from 1 to 30 bp traces the grey frontier; the oracle pools, from the band floor to 30 bp, sit above it throughout.

30 seconds posted about 4,400 times a day, 0.11 SOL (about $8) at 25,480 lamports a post; posting on 2 bp or every 10 seconds cost four times as much and moved LP results by less than 0.2 points a year.

6 The mechanism against constant product

This section isolates the mechanism. We replay four weeks of Binance one-second prices (20 October to 16 November 2026) for SOL, JUP, JTO, PYTH, WIF and TRUMP, and about seventeen regular sessions of one-minute prices for ten US stocks, through $1M pools. Each step an arbitrageur trades first against the true price, then the step's retail trades arrive (Poisson, 5,000 a day, lognormal sizes capped at $2,000, buy or sell at random), with daily volume equal to the pool in one case and a quarter of it in the other. The oracle pool's keeper posts the previous second's price with a 1.5 bp error; the measured gap between Pyth and the exchange is 1 bp at the median. BONK is left out because Binance quotes it in steps of 35.6 bp, so its one-second closes are tick bounce rather than a price.

Retail here arrives whatever the quote, so a higher fee always pays LPs more. To compare fairly we run constant product at nine fees from 1 to 30 bp, which traces a frontier of LP profit against the all-in cost retail pays, and read that frontier at the oracle pool's cost. LP profit is the markout, what retail paid against the true price minus what arbitrage took, net of the keeper's cost. Absolute returns scale with the assumed flow; the ratio at a matched price is the result. Both pools also carry the market's direction in their inventory, which the markout leaves out, and under an accurate oracle the geometric-average pool holds the same reserve ratio as constant product ([1], Example 3.3).

At the same price for traders the oracle pool pays its LPs more on every asset (Table 4,

			oracle, $\lambda = 3/4$			oracle, $\lambda = 1/2$				
Asset	Vol.	Band	Retail	LP	CP, same	Retail	LP	CP, same	Arb. cut	Liar LP
SOL	63%	10 bp	12.5 bp	41.6%	12.9%	15.0 bp	46.5%	27.9%	87%	34.9%
JUP	96%	15 bp	17.4 bp	57.1%	37.2%	19.9 bp	63.8%	49.4%	71%	33.6%
JTO	97%	15 bp	17.6 bp	56.8%	35.7%	20.1 bp	64.2%	48.7%	69%	34.3%
PYTH	109%	15 bp	17.4 bp	54.9%	36.3%	19.9 bp	62.7%	48.6%	60%	27.4%
WIF	82%	15 bp	17.5 bp	59.0%	37.2%	20.1 bp	65.3%	49.8%	80%	43.9%
TRUMP	121%	15 bp	17.6 bp	55.6%	33.7%	20.1 bp	63.4%	47.2%	65%	20.3%
SPY	17%	10 bp	12.5 bp	27.1%	8.1%	14.9 bp	28.5%	15.4%	86%	25.2%
QQQ	30%	10 bp	12.5 bp	27.1%	7.2%	14.9 bp	28.2%	14.6%	86%	25.1%
AAPL	29%	10 bp	12.4 bp	27.0%	7.6%	14.8 bp	28.1%	14.9%	85%	25.1%
NVDA	94%	10 bp	12.6 bp	25.4%	-2.1%	15.2 bp	24.5%	5.8%	83%	23.2%
GOOGL	60%	10 bp	12.5 bp	26.4%	3.5%	15.1 bp	26.8%	11.3%	84%	24.4%
TSLA	50%	10 bp	12.5 bp	26.7%	5.7%	14.9 bp	27.4%	13.1%	85%	24.7%
MSTR	92%	20 bp	22.5 bp	50.5%	27.9%	25.0 bp	51.0%	35.2%	86%	48.2%
COIN	84%	20 bp	22.6 bp	51.0%	29.5%	25.2 bp	51.8%	36.8%	87%	48.6%
CRCL	83%	20 bp	22.6 bp	51.0%	29.3%	25.2 bp	52.0%	36.8%	87%	48.6%
HOOD	77%	20 bp	22.6 bp	51.2%	30.5%	25.1 bp	52.2%	37.7%	87%	48.8%

Table 4: Daily volume equal to the pool. "Vol." is annualised volatility, "Retail" the all-in cost retail paid, "LP" the oracle pool's markout net of keeper cost in % a year, "CP, same" constant product's markout at the same retail cost, "Arb. cut" how much less arbitrage took than from constant product at the same fee ($\lambda = 3/4$), and "Liar LP" the $\lambda = 3/4$ pool with a keeper lying every step. Stocks are annualised over 252 sessions.

Figure 3): 1.5 to 1.7 times as much on the 15 bp crypto assets, 3.2 times on SOL, 1.7 to 1.8 times on the 20 bp stocks, 3.3 to 7.5 times on SPY, QQQ, AAPL, TSLA and GOOGL, and on NVDA it turns a 2.1% yearly loss into a 25.4% gain. On SOL, for instance, the $\lambda = 3/4$ pool charged traders 12.5 bp all-in and earned its LPs 41.6% a year where constant product at that price earned 12.9%. Arbitrage took 60 to 87% less than from constant product at the same fee, and a keeper lying at every step cost LPs between 2 and 35 points a year without ever making them lose money.

With a quarter of the flow the advantage mostly disappears. Most of what the oracle pool saves is the backrunning of retail impact, so with little retail there is little to save: on crypto the oracle pool ends up to about two points below constant product, and arbitrage takes up to 41% more from it, because a pool four times as deep near the price gives every small mispricing more volume. On stocks it stays 1 to 7.5 points ahead. A lying keeper hurts more in thin flow, and on four of the five 15 bp crypto pools it turned the LPs' profit into a loss of 10 to 24% a year, the one-time amount of Proposition 2 collected again every time the price moves. SOL's 10 bp band held at a 4.4% gain. Tight bands, $\lambda = 1/2$, and keepers that LPs can identify are the defences, which is why the application shows each pool's keeper, band and guard.

7 Against today's market

A full-range constant-product pool is an easy opponent. Solana's real competition is concentrated pools and prop AMMs, reached through Jupiter, so we measured that directly. From 14:51 UTC on 18 November 2026, during US market hours, we asked Jupiter's public API for round trips on 21 assets at \$100, \$1,000, \$10,000 and \$50,000, USDC into the token and the proceeds back. Half the round-trip loss is the one-way all-in cost of the best route Jupiter found, with no outside price needed. Table 5 gives the median of 1,144 round trips over 0.8 hours next to the size of an oracle pool ($\lambda = 3/4$, fee 10 bp) that would quote the same trade at least as well. For that pool we add 2.5 bp to its exact quote, the excess its realised cost showed over the ideal in the backtests.

Asset	Best route on Solana today (bp) $1,000	$10,000	$50,000	Oracle pool that matches it $1,000	$10,000	$50,000
SOL	0.1	0.1	0.6	–	–	–
JUP	2.4	3.5	6.9	–	–	–
TRUMP	2.9	7.6	35.7	–	–	$10.7M
PENGU	2.6	3.8	29.7	–	–	$14.5M
BONK	8.8	44.2	202.1	–	$1.6M	$1.3M
RAY	5.9	19.4	67.2	–	$7.2M	$4.5M
FARTCOIN	21.6	44.8	125.5	$0.6M	$1.5M	$2.2M
JTO	38.7	88.9	362.7	$0.2M	$0.6M	$0.7M
PYTH	25.1	74.3	326.3	$0.4M	$0.8M	$0.8M
WIF	30.1	69.6	245.5	$0.3M	$0.9M	$1.0M
W	67.5	378.1	1813.4	$90k	$0.1M	$0.1M
SPYx	3.2	11.2	35.6	–	–	$10.8M
QQQx	11.1	19.0	56.7	–	$7.7M	$5.6M
NVDAx	10.4	18.6	48.3	–	$8.2M	$7.0M
TSLAx	9.1	13.5	51.9	–	$50.6M	$6.3M
AAPLx	8.2	18.5	102.1	–	$8.3M	$2.8M
GOOGLx	10.3	13.1	90.3	–	$82.8M	$3.2M
MSTRx	11.8	19.0	84.9	–	$7.7M	$3.4M
CRCLx	10.6	22.4	72.9	–	$5.0M	$4.1M
COINx	24.3	61.9	460.5	$0.4M	$1.0M	$0.5M
HOODx	7.6	46.1	n/a	–	$1.5M	n/a

Table 5: Jupiter's best one-way cost (median) and the oracle-pool size that matches it. "–" means the fee plus the 2.5 bp excess is already above the best route at any size; "n/a" means Jupiter found no route back.

SOL, JUP, TRUMP and PENGU already trade at 0 to 8 bp up to $10,000, under a 10 bp fee, and so do most xStocks below $10,000; no oracle pool of any size beats those routes. The long tail is different. A pool of $90k to $2.2M matches the best route for $1,000 to $50,000 trades in JTO, PYTH, WIF, W, FARTCOIN and COINx, and for $50,000 trades in the xStocks $0.5M to $10.8M does, against an entire Jupiter-reported liquidity of $69k to $2.6M per stock.

Matching a quote is not the same as earning money, so we ran the backtest once more with the competition built in. Each token's flow is its real public-pool flow over the previous 24 hours from DexScreener (every Solana pair, trades and volume), arriving with lognormal sizes ($\sigma = 1.5$) around the observed mean trade. A trade fills in our pool only if our all-in cost for it is no worse than Jupiter's best route for that size; otherwise it goes elsewhere. The oracle pool runs under the live mainnet configuration, a 10% protocol share and the smallest base fee rule (2) allows, and for comparison we put a Raydium-style constant-product pool at 25 bp, with its 16% protocol share, through the same routing. Table 6 shows the share of the flow each pool wins and what its LPs earn, net of arbitrage, the protocol's share and the keeper.

Three kinds of market show up. Where the existing routes are thin, the oracle pool takes almost everything: a $250,000 JTO pool wins three quarters of the token's public flow and earns its LPs 60% a year, four times what a constant-product pool of the same size earns. Where prop AMMs and deep concentrated pools already quote inside the fee floor (SOL, JUP, TRUMP), it wins nothing and bleeds slowly to arbitrage, a little faster than constant product because it is deeper. Tokenized stocks sit in between. The oracle pool wins a few percent of the flow and ends close to break-even, while a constant-product pool of the same size loses 0.1 to 7.8% a year to arbitrage. Lowering the band to 6 bp for the three ETFs with the tightest confidence (SPY, QQQ, AAPL) lets a $1M QQQx pool win a fifth of the flow, at a 1.2% yearly return. So an oracle pool is not a machine that prints yield on every pair. It is a public pool that stops losing to arbitrage, and it earns where public liquidity is scarce.

Asset	Flow/day	Fee	$250,000 pool Oracle wins	Oracle LP	CP LP	$1,000,000 pool Oracle wins	Oracle LP	CP LP
JTO	$0.36M	17 bp	76.2%	60.7%	14.5%	95.1%	16.0%	7.6%
PYTH	$0.47M	17 bp	3.9%	-4.0%	-2.6%	18.9%	-1.1%	-2.5%
WIF	$0.22M	17 bp	8.8%	0.9%	-1.1%	27.9%	1.3%	-1.1%
JUP	$3.29M	17 bp	0.1%	-4.2%	-2.7%	0.2%	-3.3%	-2.7%
TRUMP	$1.90M	17 bp	0.1%	-6.6%	-3.9%	0.2%	-5.2%	-3.8%
SOL	$167.36M	12 bp	0.0%	-2.1%	-1.1%	0.0%	-1.2%	-1.1%
COINx	$1.53M	23 bp	2.2%	0.2%	-4.1%	16.7%	2.1%	-4.1%
MSTRx	$0.90M	23 bp	2.3%	-0.8%	-6.3%	7.1%	-0.9%	-6.5%
CRCLx	$2.72M	23 bp	0.6%	-0.6%	-4.7%	2.6%	-0.5%	-4.7%
NVDAx	$24.82M	12 bp	0.1%	-1.6%	-7.8%	0.5%	-1.2%	-7.8%
GOOGLx	$1.49M	12 bp	0.9%	-0.6%	-3.3%	4.2%	-0.6%	-3.3%
AAPLx	$0.63M	12 bp	1.0%	-0.1%	-0.5%	3.8%	-0.1%	-0.5%
QQQx	$1.72M	12 bp	0.4%	0.0%	-0.5%	3.1%	0.1%	-0.5%
SPYx	$4.48M	12 bp	0.0%	0.0%	-0.1%	0.2%	0.0%	-0.1%
TSLAx	$2.18M	12 bp	0.8%	-0.2%	-1.7%	3.6%	-0.1%	-1.7%
HOODx	$0.04M	23 bp	8.4%	-0.8%	-4.7%	14.8%	-0.8%	-4.8%

Table 6: Under real competition. "Flow/day" is the token's public-pool volume over 24 hours; "Fee" the oracle pool's base fee under rule (2); "wins" the share of that flow routed to the oracle pool; "LP" net LP return in % a year. DexScreener's counts include arbitrage swaps; with only half the flow treated as retail the conclusions are the same and the returns roughly halve.

8 Implementation and deployment

The program is written in Anchor 0.30.1, has 18 instructions, and has run on Solana mainnet since 18 November 2026 at `CCn6P5NHvC2snSnTKZrSLvyy7Gr8TMXCGXzd8iZabV2P`, deployed at its exact size with an upgrade authority we hold. Every oracle swap in the test suite paid exactly what an operation-for-operation JavaScript copy of the curve computes, to the raw unit, and the application's client matched the engine on 400 random trades.

Instruction	Compute units	Wallet transaction bytes
Swap, oracle pool, $\lambda = 3/4$	47,700 to 49,153	686
Swap, oracle pool, $\lambda = 1/2$	32,031 to 32,412	686
Swap, oracle pool, $\lambda = 3/4$, with a hook	53,061 to 55,848	750
Swap, constant product, same pair	21,416 to 21,432	n/a
Price update	3,198	279
Create price feed	19,986	466
Create oracle pool	51,091 to 58,591	602

Table 7: Measured on a local validator; swap sizes include creating the output token account.

The local-validator suite covers feed and pool creation (including rule (2) at its edge), exact quotes and repricing, five guard failures that refuse the swap, eight forged-account cases, the hostile keeper, the fallback drain, the update sandwich, a hook on an oracle pool, transaction sizes, and closing every account back to zero rent. Unit tests check that roots bracket the true value for over 2,000 random 128-bit inputs, that the invariant never drops across 3,000 swaps with moving prices, and that round trips never profit even at zero fee. With rule (2) the suite passes 26 of 26. The deployed program is 612,864 bytes, and its 3.114 SOL of rent can be recovered by closing it; the build with the new rule is 320 bytes larger.

9 Related work

Oracle-dependent market makers predate the theory we build on: DODO's proactive market maker [9], Curve v2's internally pegged curve [10], UAMM [11] and the dynamic curves of Krishnamachari et al. [12]; Bergault et al. treat price-aware market making as an optimisation problem [13]. Amini and Feinstein supply the axioms, the decomposition of loss-versus-rebalancing with oracle error and the analysis of a sandwiched update [1]; our potential argument extends the last to a keeper that moves the price at will inside a band. Loss-versus-rebalancing is due to Milionis et al. [2], with fees and discrete blocks in [3]. CFMMs as price oracles appear in [5, 6], pools that auction the right to set fees in [7], and concentrated liquidity in [8]. Solmaz, Heimbach and Milionis document prop AMMs across chains [4], and oracle manipulation is surveyed in [14]. On Solana, Lifinity runs oracle pools closed to deposits, Jupiter's lending AMM takes public liquidity with an optional external centre price for correlated pairs, and Meteora's dynamic pools set fees from the pool's own history. On EVM chains Metric opened oracle-guided pools to outside depositors in October 2026 and reported about $1.2B of volume in its first five days. As far as we know OniSwap is the first public-deposit oracle-parametrized pool on Solana.

10 Limitations

The backtests are models. In the mechanism comparison retail arrives whatever the quote; comparing at equal prices removes most of the effect but not the volume a better price would attract. The market test fixes that with real routing but takes its flow from one day of DexScreener counts, which include arbitrage, and its competitor costs from one US session of Jupiter quotes; both change daily. The stock data is one-minute, which hides sub-minute moves and understates arbitrage for every pool, and the stock confidence figures are single readings. The refreshed guard needs signed Pyth updates, which Pyth's public service now gives out only with an access key, and when an equity market is closed Pyth stops publishing and the pool stops swapping (deposits and withdrawals continue).

On the security side, the theorem is exact and covers every trade size. The exact computation is certified for its grid and allows trades up to a thousand times the reserves. Our lying keeper in the backtests lies around a current Pyth price; a keeper that refreshes Pyth itself can also delay the refresh up to the age limit and so shift the window by Pyth's lag c, which Proposition 2, Corollary 1 and Table 1 account for and the backtests do not. One feed may serve several pools, and a lie reaches all of them, each within its own bound. The deployed program checks the gross fee rather than rule (2) until its next upgrade, and it has not been audited.

11 Conclusion

A pool that sees the market can be deeper and lose less, if its oracle is good, and on Solana "good" turns into concrete numbers: the fee the pool keeps must cover the oracle's whole band, the band is set by how fresh and confident the oracle is, and a guard that fails must stop the pool rather than switch it to another curve. Built that way, the pool earns its LPs several times what a public constant-product pool earns at the same price for traders, does not lose to arbitrage where a constant-product pool would, and takes most of the flow in markets where public liquidity is thin.

A Proofs

Proposition 2. Near balance, buying da of the risky asset moves x up by about $2\,da/a$, so $V/4$ dollars move x by one unit. Using $e^z \geq 1 + z$, a buy at state x and keeper offset y gains

about $P(-\lambda y-(1-\lambda)x-\phi)$ per unit and a sell $P(\lambda y+(1-\lambda)x-\phi)$. Written against dx these give G_{att}, whose reserve term integrates to $-\frac{1-\lambda}{2}(x_{\text{end}}^2-x_0^2)$. With $y\in[c-\beta,c+\beta]$, $-\lambda\int y\,dx\le-\lambda c(x_{\text{end}}-x_0)+\lambda\beta\int|dx|$, so

$$G_{\text{att}}\le\tfrac{V}{4}\Big[-\lambda c(x_{\text{end}}-x_0)-(\phi-\lambda\beta)|x_{\text{end}}-x_0|-\tfrac{1-\lambda}{2}(x_{\text{end}}^2-x_0^2)\Big]\\ -\tfrac{V}{4}(\phi-\lambda\beta)\Big(\int|dx|-|x_{\text{end}}-x_0|\Big).$$

Maximising the first line over x_{end} gives G^*, and $D_{\text{buy}}+D_{\text{sell}}=2(\lambda\beta-\phi)<0$.

Theorem 1. A buy of u at price O moves the pool along $(Oa')^\lambda+(b+gu)^\lambda=(Oa)^\lambda+b^\lambda$ and leaves $b'=b+\rho u$. Hence

$$\begin{aligned}H(a',b')-H(a,b)&=\big[(b+\rho u)^\lambda-b^\lambda\big]-(G/O)^\lambda\big[(b+gu)^\lambda-b^\lambda\big]\\&\ge\big[(b+\rho u)^\lambda-b^\lambda\big]-e^{\lambda\beta}\big[(b+gu)^\lambda-b^\lambda\big],\end{aligned}$$

since $O\ge Ge^{-\beta}$. Put $v=\rho u/b$ and $k=g/\rho\le e^{-\beta}$. Because the right side falls as k grows it suffices to take $k=e^{-\beta}$, and after dividing by $b^\lambda e^{\lambda\beta}$ the claim is $\psi(v)\ge0$ for

$$\psi(v)=k^\lambda\big((1+v)^\lambda-1\big)-\big((1+kv)^\lambda-1\big).$$

Now $\psi(0)=0$ and $\psi'(v)=\lambda k^\lambda\big[(1+v)^{\lambda-1}-(1/k+v)^{\lambda-1}\big]>0$, because $1/k>1$ and $\lambda-1<0$. A sell of d at price O moves the pool along $(O(a+gd))^\lambda+b'^\lambda=(Oa)^\lambda+b^\lambda$ and leaves $a'=a+\rho d$, so

$$\begin{aligned}H(a',b')-H(a,b)&=G^\lambda\big[(a+\rho d)^\lambda-a^\lambda\big]-O^\lambda\big[(a+gd)^\lambda-a^\lambda\big]\\&\ge G^\lambda\Big\{\big[(a+\rho d)^\lambda-a^\lambda\big]-e^{\lambda\beta}\big[(a+gd)^\lambda-a^\lambda\big]\Big\},\end{aligned}$$

using $O\le Ge^{\beta}$, and the same function ψ with $v=\rho d/a$ finishes the proof.

Corollary 1. The attacker's profit is W_0-W_{end} and every reachable state has $H\ge H_0$. Minimising $Pa+b$ over $(Ga)^\lambda+b^\lambda\ge H_0$ with a Lagrange multiplier puts the optimum where $b/a=(PG^{-\lambda})^{1/(1-\lambda)}$, which gives the closed form.

The two facts behind the exact computation. For a buy to $a'<a$, write $B^\lambda=O^\lambda(a^\lambda-a'^\lambda)+b^\lambda$, so that $b'=b+(B-b)/g$ (the whole fee retained). Then $dB/db=(B/b)^{1-\lambda}\ge1$ and $db'/db=1+(dB/db-1)/g\ge1$; and B, hence b', increases with O, so a buyer wanting b' low uses the low edge. For a sell of $d=a'-a$, $b'^\lambda=b^\lambda-O^\lambda\big((a+gd)^\lambda-a^\lambda\big)$, so $db'/db=(b'/b)^{1-\lambda}>0$, and b' decreases with O, so a seller uses the high edge.

B Reproduction

Every number in this paper comes from a script in the OniSwap repository, and `whitepaper/make_tables.py` builds the tables and figures from their outputs.

- Security: `spike/oracle/exact_attack.py`, `fee_threshold.py`, `pump_curve.py`, `potential.py`; method and results in `spike/oracle/EXACT.md`.
- Pyth on mainnet: `spike/win/pushfeeds.mjs`, `pythcost.mjs`, `conf_sampler.mjs`; sponsored regime `spike/oracle/measure.py` and `replay.py`.
- Mechanism comparison: `spike/win/fetch_klines.py`, `fetch_equity.py`, `run_all.sh` (`backtest.mjs`), `analyze.py`.

- Market test: `spike/win/jup_sampler.py`, `compete.py`, `dexflow.py`, `run_route.sh` (`route.mjs`), `route_summary.py`.
- Engine: `bash engine/tests/run.sh`; hostile hook: `spike/hooksec/run.sh`.

References

[1] H. Amini and Z. Feinstein. Oracle-parametrized constant function market makers: from price feeds to pricing rules. arXiv:2609.33799, 2026.

[2] J. Milionis, C. C. Moallemi, T. Roughgarden and A. L. Zhang. Automated market making and loss-versus-rebalancing. arXiv:2208.06046, 2022.

[3] J. Milionis, C. C. Moallemi and T. Roughgarden. Automated market making and arbitrage profits in the presence of fees. arXiv:2305.14604, 2023.

[4] O. Solmaz, L. Heimbach and J. Milionis. Active liquidity on chain: evidence from propAMMs across chains. arXiv:2609.38056, 2026.

[5] G. Angeris and T. Chitra. Improved price oracles: constant function market makers. arXiv:2003.10001, 2020.

[6] G. Angeris, H.-T. Kao, R. Chiang, C. Noyes and T. Chitra. An analysis of Uniswap markets. arXiv:1911.03380, 2019.

[7] A. Adams, C. C. Moallemi, S. Reynolds and D. Robinson. am-AMM: an auction-managed automated market maker. arXiv:2403.03367, 2024.

[8] H. Adams, N. Zinsmeister, M. Salem, R. Keefer and D. Robinson. Uniswap v3 core. Technical report, Uniswap Labs, 2021.

[9] DODO Team. DODO: a next-generation on-chain liquidity provider powered by the proactive market maker algorithm. Technical report, 2020.

[10] M. Egorov. Automatic market-making with dynamic peg. Technical report, Curve Finance, 2021.

[11] D. J. Im, A. Kondratskiy, V. Harvey and H.-W. Fu. UAMM: price-oracle based automated market maker. arXiv:2308.06375, 2023.

[12] B. Krishnamachari, Q. Feng and E. Grippo. Dynamic curves for decentralized autonomous cryptocurrency exchanges. arXiv:2101.02778, 2021.

[13] P. Bergault, L. Bertucci, D. Bouba, O. Guéant and J. Guilbert. Price-aware automated market makers: models beyond Brownian prices and static liquidity. arXiv:2405.03496, 2024.

[14] S. Eskandari, M. Salehi, W. C. Gu and J. Clark. SoK: oracles from the ground truth to market manipulation. In Proc. ACM AFT, 2021.