

UNIVERSITY OF
CAMBRIDGE
Department of Computer
Science and Technology

Proprietary AMMs as Latency-Aware Dealers: Liquidity Provision under Latency and Inventory Risk

Florian Klein



Peterhouse

June 2026

Submitted in partial fulfillment of the requirements for the
Master of Philosophy in Advanced Computer Science

Total page count: 93

Main chapters (excluding front-matter, references and appendix): 53 pages (pp 10–62)

Main chapters word count: 14812

Methodology used to generate that word count:

```
texcount -1 -sum include/chap{1..7}*.tex
```

Declaration

I, Florian Klein of Peterhouse, being a candidate for the Master of Philosophy in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this report, I adhered to the Department of Computer Science and Technology AI Policy. I am content for my report to be made available to the students and staff of the University.

Signed:



Date: June 5, 2026

Abstract

Traditional analyses of automated market makers (AMMs) assume that prices are determined by a fixed on-chain invariant. However, on modern blockchains such as Solana, the majority of liquidity is provided by proprietary AMMs (propAMMs) whose quotes are updated continuously rather than derived from a static invariant. These propAMMs often provide quotes better than those on centralised exchanges like Binance. Yet, their pricing logic is deployed as closed-source sBPF bytecode, preventing independent analysis. This leads to the behavior and quoting logic being opaque to users and researchers and raises concerns about the potential for hidden risks and unfair advantages in the market.

This dissertation introduces the first semantic tracing framework for Solana's sBPF virtual machine, which recovers closed-form quoting formulas from on-chain binaries via backward data-flow analysis and control-dependency resolution. Applied to seven deployed propAMM protocols, we recover the protocols' quoting formulas and provide the first empirical study of propAMM mechanics and behavior across $90.1M in trading volume. The study identifies two distinct architectural families and finds that profitability is driven mainly by the composition of order flow a protocol attracts.

Acknowledgements

I am grateful to my supervisor, Dr Ferdinando Samaria, for his valuable input and feedback throughout this project, and for the weekly meetings that kept the work on track. I also thank my other supervisor Prof. Frank Stajano for his guidance and strategic perspectives on approaching the thesis. Finally, I thank my friends at Cambridge and Munich for their support and the many coworking sessions that made this year both productive and enjoyable.

Contents

1 Introduction **10**

1.1 Motivation . . . 10
1.2 Research Gap . . . 11
1.3 Research Questions . . . 11
1.4 Contributions . . . 12
1.5 Dissertation Outline . . . 12

2 Background **13**

2.1 Automated Market Makers . . . 13
2.1.1 Constant-Function Market Makers . . . 13
2.1.2 Capital Efficiency and Concentrated Liquidity . . . 14
2.1.3 Impermanent Loss and Liquidity Provider Profitability . . . 15
2.1.4 Proprietary AMMs . . . 15
2.1.5 DEX Aggregators . . . 16
2.2 Solana Architecture . . . 16
2.2.1 Slot-Based Execution Model . . . 16
2.2.2 Account Model and Program Runtime . . . 17
2.2.3 BPF Bytecode and On-Chain Binaries . . . 18
2.3 Market Microstructure . . . 20
2.3.1 Adverse Selection and the Bid-Ask Spread . . . 20
2.3.2 Optimal Market Making . . . 20
2.3.3 Spread Decomposition and Flow Toxicity . . . 21
2.4 Program Analysis Foundations . . . 21
2.4.1 Program Slicing . . . 21
2.4.2 Dynamic Taint Analysis . . . 22
2.4.3 Dominator Trees . . . 22

3 Related Work **24**

3.1 LP Returns and AMM Profitability . . . 24
3.2 MEV and Order Flow in DeFi . . . 25
3.3 Blockchain Binary Analysis . . . 26
3.4 Gap Analysis . . . 27

4 Approach **29**
4.1 Toolchain for Reverse-Engineering . . . 29
4.1.1 Local Transaction Simulation . . . 30
4.1.2 Instrumented Execution . . . 30
4.1.3 Backward Formula Reconstruction . . . 32
4.1.4 Control Dependency Resolution . . . 33
4.1.5 Soak Testing Harness . . . 35
4.2 Formula Recovery . . . 35
4.2.1 From Trace to Formula . . . 35
4.2.2 From Test Failure to Fix . . . 37
4.3 On-Chain Observation . . . 37

5 Implementation **38**
5.1 Tracing Infrastructure . . . 38
5.2 Simulator . . . 40
5.3 Protocol-Specific Adaptations . . . 40
5.4 Analysis Toolchain . . . 41

6 Results **43**
6.1 Experimental Setup . . . 43
6.1.1 PropAMM Selection . . . 43
6.1.2 Dataset, Methodology and Market Conditions . . . 43
6.2 Reverse-Engineering Results . . . 44
6.2.1 Recovered Quoting Formulas . . . 44
6.2.2 Validation . . . 48
6.3 Empirical Analysis . . . 49
6.3.1 Quoting Behaviour . . . 49
6.3.2 Risk Management and Inventory Dynamics . . . 51
6.3.3 Order Flow Composition . . . 53
6.3.4 LP Profitability . . . 55
6.3.5 Routing Efficiency . . . 57
6.3.6 Alternative Explanations . . . 58

7 Discussion and Conclusion **60**
7.1 Interpretation of Findings . . . 60
7.2 Limitations . . . 60
7.3 Implications and Future Work . . . 61
7.4 Summary of Contributions . . . 62

References **63**

References **63**

A Recovered propAMM Quoting Specifications **68**
A.1 Worked Example: Reverse-Engineering BisonFi 68
A.2 AlphaQ 77
A.3 SolFi V2 79
A.4 GoonFi V2 81
A.5 Humidifi 83
A.6 BisonFi 86
A.7 Tessera V 88
A.8 ZeroFi 89

List of Abbreviations **92**

List of Figures

1.1 PropAMM share of Solana DEX volume 11

2.1 CFMM invariant curves 14
2.2 Full-range vs. concentrated liquidity 15
2.3 Solana program and account architecture 17
2.4 Solana account structure. 18
2.5 sBPF VM memory layout 19
2.6 Static vs. dynamic program slicing 22

4.1 Methodology overview 30
4.2 Reverse engineering pipeline 31
4.3 Per-instruction semantic tracing 32
4.4 Backward value tracing example 33
4.5 DTA++ branch classification 34

5.1 Feature flag dependency chain 39
5.2 Example backward trace output: BisonFi B2Q swap 41
5.3 Example DTA++ control-dependency output 42

6.1 Three propAMM quoting styles compared 50
6.2 Market share by swap size bin 51
6.3 Effective spread vs. trade size 52
6.4 Inventory imbalance time series 53
6.5 Inventory mean reversion half-lives 53
6.6 LP markout curves by horizon 54
6.7 Markout by trade size quintile 55
6.8 Realised spread decomposition 56
6.9 Continuous-hedging LP PnL by protocol 56
6.10 Static hedge LP PnL by protocol 57
6.11 Cross-protocol routing efficiency matrix 58

List of Tables

4.1 Recurring assembly idioms across the seven Proprietary Automated Market Makers (propAMMs) studied. 36

5.1 Engineering summary 38

6.1 Recording session dataset summary 44
6.2 PropAMM architectural feature comparison 45
6.3 Quoting summary by median spread 51

Chapter 1

Introduction

1.1 Motivation

Decentralised Exchanges (DEXs) on blockchains are increasingly becoming an important component of the global financial infrastructure. Solana processed approximately $968 billion in DEX volume in 2024 and approximately $1.95 trillion in 2025, surpassing Ethereum's DEX volume for much of that period [43]. The majority of this volume is routed through aggregators, most notably Jupiter.

A new class of Automated Market Makers (AMMs) has emerged on Solana that differs from the open-source, constant-function designs that dominate the academic literature. These *propAMMs* are closed-source programs whose pricing logic is compiled to Solana BPF (sBPF) (Solana's bytecode format) and deployed on-chain without source code or documentation. An off-chain operator continuously updates quoting parameters in response to market conditions, similar to a centralised market maker, but the quoting logic itself cannot be inspected directly.

Despite their opacity, propAMMs have come to account for a large share of Solana's DEX activity. As shown in Figure 1.1, propAMMs grew from negligible volume before October 2024 to over 50% of total Solana DEX volume by late 2025, peaking above 65% in early 2026 [43]. On heavily traded pairs such as Solana (SOL)/USD Coin (USDC), propAMM share has been even higher, and on-chain spreads have compressed from approximately 30 basis points to below 1 basis point, matching and in some cases surpassing centralised-exchange execution quality.

On centralised exchanges, market-maker algorithms are entirely proprietary and unobservable; their strategies can only be inferred indirectly from public order book data. PropAMMs, by contrast, execute on a public ledger: while their source code is closed, every state transition, parameter update, and trade is recorded on-chain. This makes them the first class of market-making algorithms whose behaviour can be studied systematically, provided their logic can be recovered from the deployed binaries.

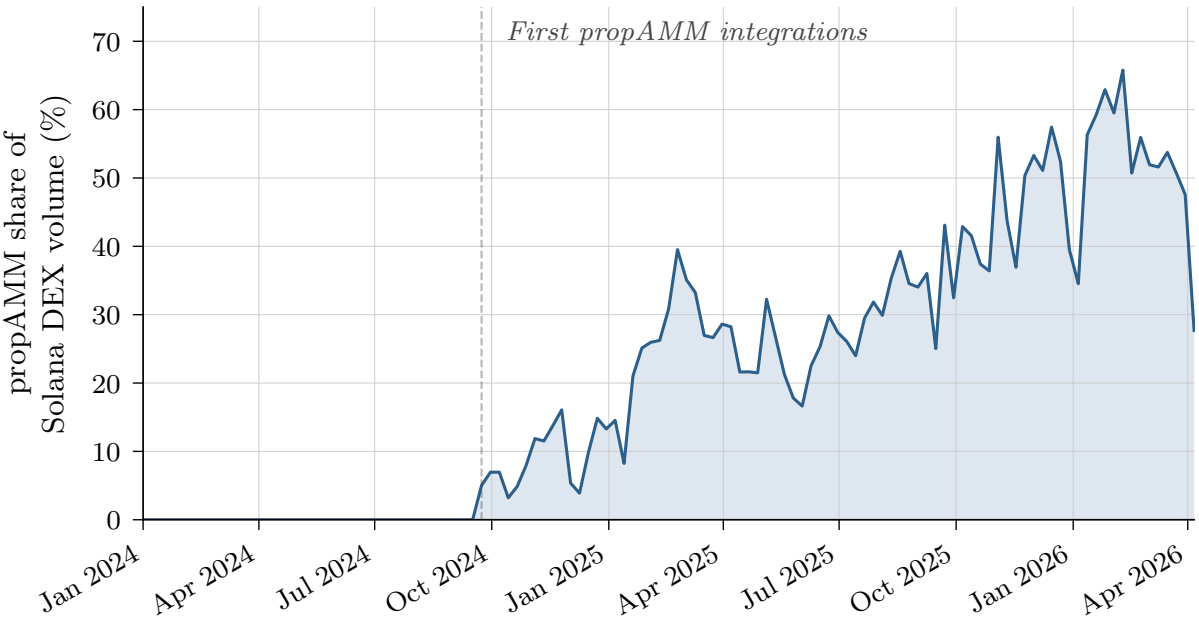


Figure 1.1: Biweekly propAMM share of total Solana DEX volume, January 2024 to early April 2026. PropAMMs grew from zero before October 2024 to regularly exceeding 50% of all Solana DEX volume by late 2025. The series ends at the last fully aggregated window; the still-incomplete final week is omitted. Data: DefiLlama [17].

Economic structure. Unlike a Uniswap pool, where public liquidity providers deposit capital and earn swap fees, a propAMM is funded by an off-chain operator. This operator also sets the quoting parameters and receives the entire Profit and Loss (PnL): it earns the spread in each quote, loses to adverse selection when inventory drifts, and has its own operating costs (e.g. transaction fees for parameter updates, off-chain infrastructure).

1.2 Research Gap

Existing academic work on AMMs assumes that the pricing function is publicly known. Studies of Liquidity Provider (LP) profitability [12, 25, 31], adverse selection costs [34], and market quality [8] all analyse open-source protocols. PropAMMs fall outside this scope: their quoting logic must first be *recovered* before any empirical analysis is possible.

On the program analysis side, existing tools for Solana binary analysis focus on security: vulnerability detection [15] and fuzzing [41]. To our knowledge, no prior work uses execution tracing and backward dataflow analysis to recover closed-form mathematical expressions from deployed sBPF programs.

1.3 Research Questions

This gap motivates two research questions:

RQ1 *Can closed-form quoting formulas be recovered from closed-source sBPF programs?*

We ask whether instrumented execution tracing and backward dataflow can reconstruct the complete pricing logic of deployed propAMM binaries, validated against the live on-chain program.

RQ2 *How do proprietary AMMs behave, and what determines their profitability?* Given recovered formulas, we ask how propAMMs differ in quoting strategy, risk management, order flow composition, and how these factors relate to the observed profitability of each protocol.

1.4 Contributions

A reverse-engineering methodology for sBPF programs. We develop a systematic, repeatable methodology for recovering closed-form quoting formulas from closed-source Solana programs. The pipeline combines instrumented sBPF execution with per-instruction tracing, backward dataflow analysis that reconstructs computation graphs from output values to input sources, and control dependency resolution via post-dominator analysis and targeted alternative path execution [28]. The recovered formulas are validated through continuous soak testing against the live on-chain binary.

An empirical study of proprietary AMMs. Using the recovered formulas, we conduct the first systematic analysis of propAMM behaviour under live market conditions. We record on-chain state for SOL/USDC pools across seven protocols over a continuous 10-hour session and apply market microstructure tools to analyse them with respect to multi-horizon markout analysis [31], spread decomposition [26], and Loss-Versus-Rebalancing (LVR)-based continuous-hedging PnL [34]. The analysis reveals that propAMM profitability is associated less with the sophistication of the quoting formula than with the composition of order flow each protocol attracts. Among the six active protocols, per-trade markout ranged from 0.1 to 1.7 basis points, with the tightest-spread measured protocol (Humidifi) capturing the most favourable flow composition.

1.5 Dissertation Outline

Chapter 2 introduces the necessary background on automated market makers, Solana's architecture, market microstructure theory, and program analysis, and Chapter 3 surveys related work and identifies the gaps this dissertation addresses. Chapters 4 and 5 then develop the reverse-engineering methodology and its implementation, covering local transaction simulation, instrumented execution, backward formula reconstruction, and validation. Chapter 6 applies this methodology to seven propAMM protocols and presents the recovered formulas alongside an empirical analysis of their behaviour. Chapter 7 interprets the findings, discusses limitations, and outlines future work.

Chapter 2

Background

This chapter introduces the background for our analysis of proprietary automated market makers: on-chain market making mechanisms, Solana's architecture and virtual machine, and the program analysis concepts used in our reverse-engineering methodology.

2.1 Automated Market Makers

Traditional financial markets rely on market makers to provide liquidity, typically on orderbook-based exchanges. The orders on those exchanges are actively managed, updated and cancelled in response to market conditions [10]. On blockchains, maintaining an on-chain order book is expensive: every order placement, cancellation, and update requires a state-changing transaction that has gas costs, and the latency of block confirmation introduces additional risk [48]. Based on academic research on prediction markets [23], AMMs were implemented that passively provide liquidity according to a fixed pricing function, without the need for an order book or active management. This is often called a conservation function or invariant, which also motivates the term Constant-Function Market Makers (CFMMs) [4].

2.1.1 Constant-Function Market Makers

The simplest and most widely deployed CFMM is the constant-product market maker, used for example in Uniswap v2 [1] on Ethereum and a variety of other exchanges like Orca [36] or Raydium [38] on Solana. Each pool maintains reserves of two assets x and y such that their product remains constant:

$$x \cdot y = k \tag{2.1}$$

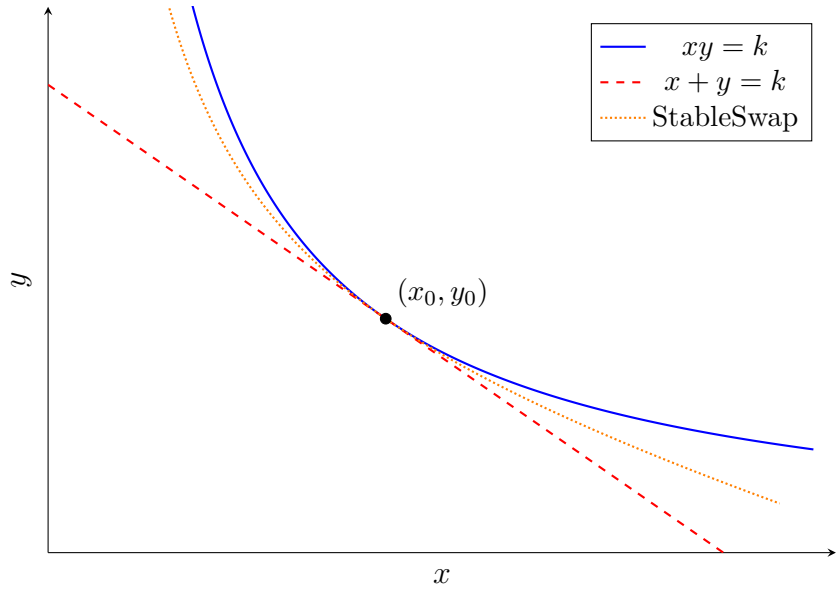


Figure 2.1: Invariant curves for three CFMM designs. The constant-product curve spreads liquidity across all prices. The constant-sum line (dashed) has zero slippage (e.g. for fixed-price stablecoins). The StableSwap hybrid (dotted) interpolates between the two; liquidity is more concentrated near parity [19].

This invariant determines the exchange rate between the two assets. For a given input amount Δx, the output Δy is:

$$\Delta y = y - \frac{k}{x + \Delta x} \tag{2.2}$$

The marginal price is given by the ratio of reserves, $p = \frac{y}{x}$, which adjusts continuously as trades move along the invariant curve. Angeris and Chitra [4] show that arbitrageurs are incentivised to trade CFMMs to the point where the reported price matches the external market price, establishing CFMMs as reliable on-chain price oracles. The constant-product design is one of a broader CFMM family using different invariants for different asset pairs (Figure 2.1). The *constant-sum* variant ($x + y = k$) has zero slippage and is used for assets that should trade at parity (e.g. stablecoins), while Curve's *StableSwap* [19] interpolates between constant-sum and constant-product via an amplification parameter, concentrating liquidity near parity for stablecoin pools.

The constant-product invariant is simple but has limitations that shaped the evolution of AMMs: low capital utilisation and impermanent loss.

2.1.2 Capital Efficiency and Concentrated Liquidity

In a constant-product pool, liquidity is spread uniformly across all possible prices, but most trading happens within a narrow band around the current market price. Reserves at extreme prices contribute negligible depth and amount to idle capital. Concentrated Liquidity Market Makers (CLMMs), introduced by Uniswap v3 [2], improve capital efficiency by allowing LPs to concentrate their capital within a chosen price interval $[p_a, p_b]$ (Figure 2.2). This design was subsequently adopted by other protocols including Orca

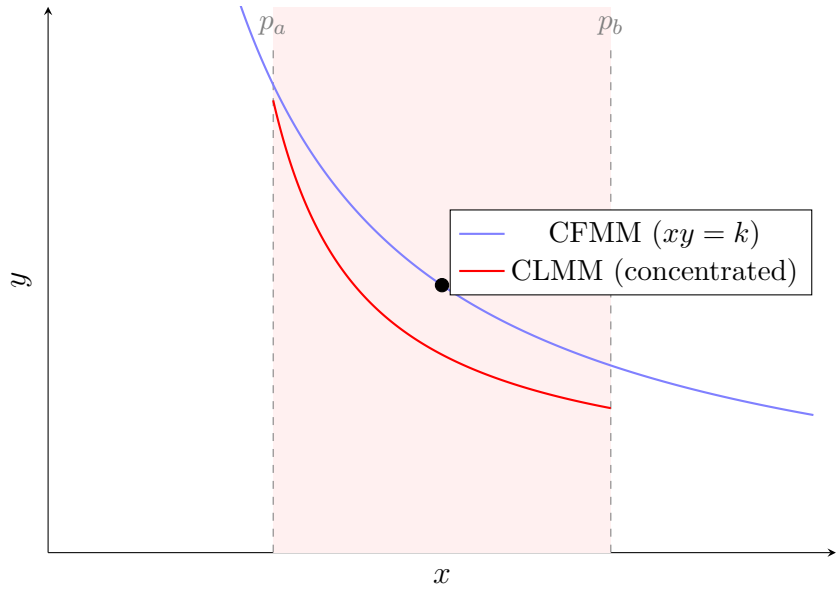


Figure 2.2: Comparison of a full-range CFMM (blue) and a concentrated liquidity position (red) active only within the tick range $[p_a, p_b]$. With the same capital, the CLMM curve is steeper inside $[p_a, p_b]$ and zero outside it: restricting the same reserves to a narrower price band raises the marginal liquidity, and hence the depth, available within that band.

Whirlpools [35] on Solana. However, CLMMs still rely on a fixed invariant within each tick: the pricing mechanism remains transparent and deterministic.

2.1.3 Impermanent Loss and Liquidity Provider Profitability

A LP who deposits assets into a constant-product pool is exposed to Impermanent Loss (IL): the difference in value between holding the LP position and simply holding the original assets. When the relative price of the two assets changes by a factor r (i.e. price ratio moves from p_0 to $r \cdot p_0$), the IL for a constant-product pool is:

$$\mathrm{IL}(r) = \frac{2\sqrt{r}}{1+r} - 1 \tag{2.3}$$

To compensate for impermanent loss, AMMs typically charge a fee on every swap (e.g. 0.3% for Uniswap v2 pools). Milionis et al. [34] formalise this cost as LVR, discussed further in Section 3.1.

2.1.4 Proprietary AMMs

All variants discussed so far share a further limitation: *passivity*. The pricing function is fixed at deployment and cannot adapt to changing market conditions, so external arbitrageurs are required to maintain price alignment across venues.

This motivates a different approach: *propAMMs*, which replace the fixed invariant with a dynamic, operator-controlled quoting function. A propAMM program accepts buy/sell

instructions like any other AMM, but its pricing logic is determined by parameters that an off-chain operator can update between trades. The operator typically sets the mid-price based on external market data and adjusts the spread, fee structure, and depth schedule in response to market conditions. While those programs can occasionally be updated to change the quoting logic itself, the regular case is that only the parameters stored in on-chain accounts are updated, while the underlying code remains unchanged.

2.1.5 DEX Aggregators

On Solana, the majority of swap volume is not submitted directly to individual AMMs but routed through *DEX aggregators* such as Jupiter [27]. An aggregator accepts a user's swap intent (input token, output token, amount) and queries all available on-chain venues (such as CFMMs, CLMMs, propAMMs, and order book DEXs) to find the execution path that tries to maximise the user's output. The aggregator may split a single swap across multiple venues or route through intermediate tokens if this yields a better price.

This architecture has two consequences relevant to our analysis. First, propAMMs do not attract order flow organically; they must *compete* for aggregator-routed flow by offering the best execution price at the moment the off-chain routing decision is made, before on-chain settlement. Unlike a CFMM, where LPs earn a fixed swap fee on top of the invariant price, a propAMM has no separate fee; the spread the operator uses in the quote is its only per-trade revenue. The composition of order flow a protocol receives is therefore a function of its quoted spread relative to all other venues. Second, aggregator routing also gives us a basis for cross-venue comparison: for every swap, the aggregator briefly considers all venues at once, so we can replay the same input against every protocol's quoting function and compare the chosen route against what each unselected venue would have offered at that moment (Section 6.3.5).

2.2 Solana Architecture

Solana [40, 42] is a high-performance blockchain widely used for decentralised finance. This section covers the architectural details relevant to our methodology.

2.2.1 Slot-Based Execution Model

Similar to other blockchains like Ethereum, Solana organises progress into discrete time intervals called *blocks* or *slots*, which are produced by validators at a target rate of 400 ms per slot. While the ordering of transactions on Ethereum is only finalised at the end of each block, block production on Solana is continuous: transactions are included on a first-come-first-served basis in each slot and this progress is continuously gossiped to the network. This means traders can react to changes within the current slot, making

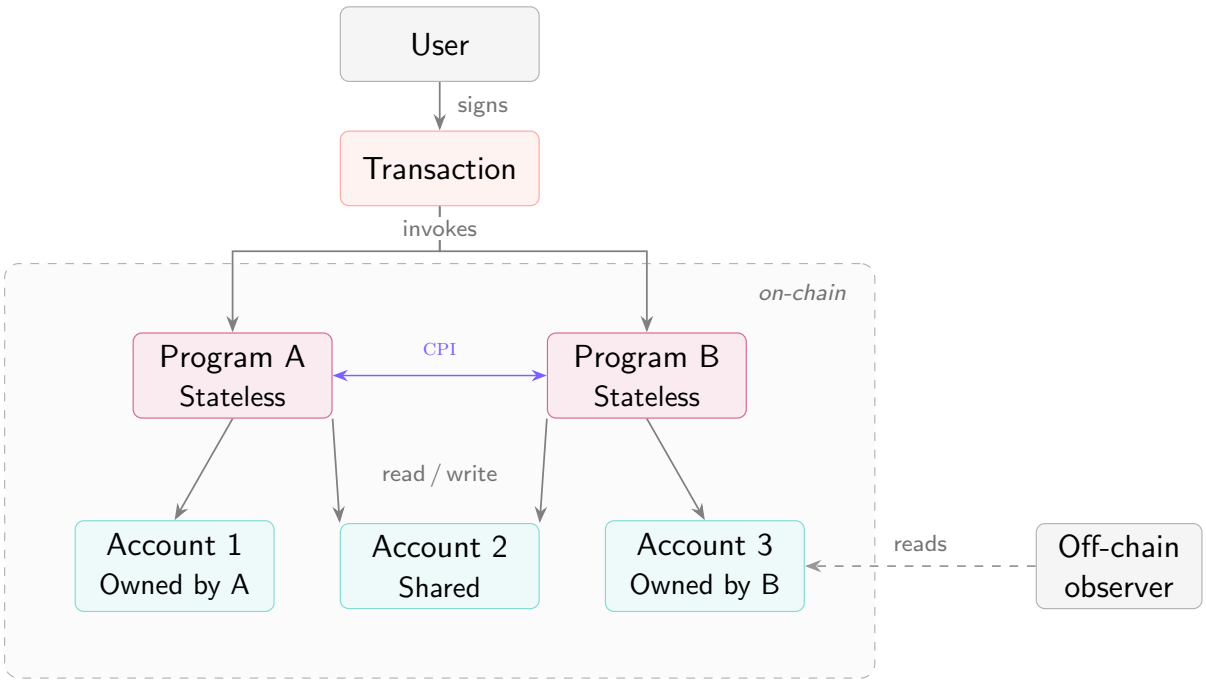


Figure 2.3: Simplified Solana program and account architecture. Programs are stateless and read/write state through separate on-chain accounts.

Solana's execution model closer to a continuous-time market than Ethereum's block-by-block processing.

2.2.2 Account Model and Program Runtime

Solana programs (smart contracts) are stateless: they do not have their own storage but instead read and write state through separate on-chain accounts (Figure 2.3). Programs are invoked through user-signed transactions and can call other programs via cross-program invocation. Crucially, all account data is publicly readable, meaning that propAMM state (pool balances, configuration parameters, quoting parameters) can be directly inspected and recorded over time if the account layout and field semantics are known.

Account structure. Each account is represented as an `AccountSharedData` struct with five fields (Figure 2.4). The three fields most relevant to our analysis are:

- `lamports` (`u64`, 8 bytes): the account's SOL balance, used to pay transaction fees (1 SOL = 10^9 lamports).
- `owner` (`Pubkey`, 32 bytes): the public key of the program that owns this account. Only the owning program may modify the account's `data` field.
- `data` (`Vec<u8>`, variable, up to 10 MiB): an opaque byte array whose interpretation is determined entirely by the owning program.
- `executable` (`bool`: 1 byte): special flag indicating whether the account contains executable code (i.e., a program) or is a data account.

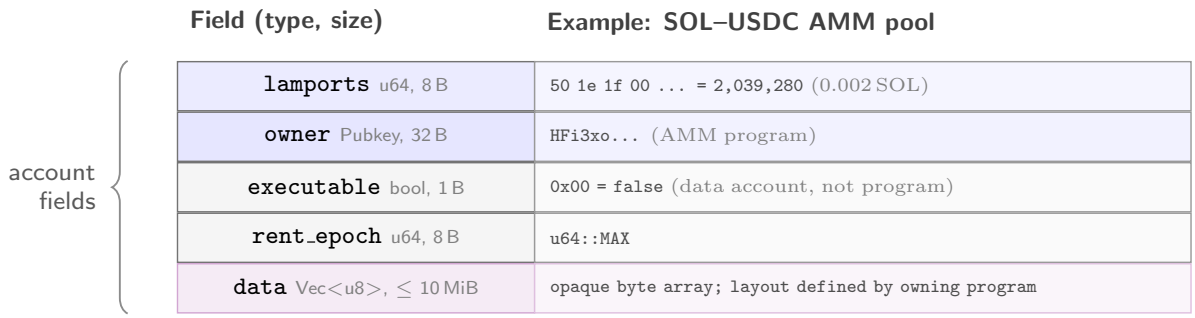


Figure 2.4: Solana account structure.

The remaining field `rent_epoch` is a legacy Solana feature and not relevant for our analysis of propAMM logic.

Transaction execution. A Solana transaction is a signed message containing one or more *instructions*, each specifying a program to invoke and a list of accounts used as input for the program with read or write flags. Execution proceeds through several stages. First, the *account loader* acquires reader–writer locks on all referenced accounts, enabling transactions with disjoint account sets to execute in parallel. Next, the *program cache* resolves the invoked program: on a cache hit the compiled code is reused; on a miss the Executable and Linkable Format (ELF) binary is loaded, verified, and JIT-compiled. The program then runs inside the sBPF Virtual Machine (VM), which provides four memory regions (code, stack, heap, and a serialised input buffer containing account data) and enforces a compute unit (CU) budget that caps execution cost. Finally, the *verify* stage checks that lamport balances are conserved, and ownership rules are respected; if any check fails, all state changes are rolled back atomically.

For AMM transactions, this means swaps on different pools execute concurrently, while swaps on the same pool are serialised by the write lock on the pool's state account.

2.2.3 BPF Bytecode and On-Chain Binaries

The Solana runtime executes programs compiled to a subset of eBPF (extended Berkeley Packet Filter (BPF)) [46] bytecode, known as sBPF, using a custom virtual machine and JIT compiler [6]. Users write programs in high-level languages like Rust or C, which are then compiled to sBPF bytecode and deployed on-chain [18]. Once deployed, the program's bytecode is stored on-chain and can be downloaded and disassembled by anyone. This publicly available bytecode enables reverse-engineering of the quoting functions embedded in propAMM programs. Moreover, the following details of the execution model are relevant to this dissertation:

ELF loading. On-chain programs are stored as ELF binaries in the `data` field of an executable account. When invoked, the runtime parses the ELF and verifies the bytecode

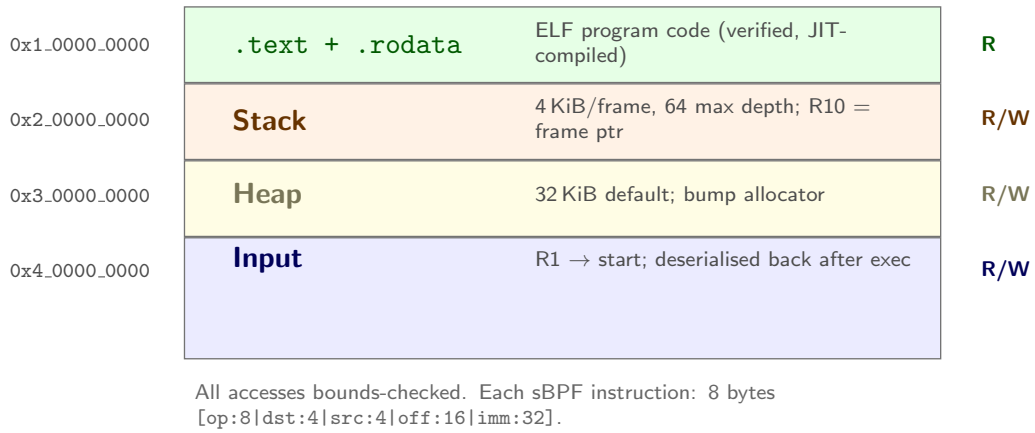


Figure 2.5: SBPF virtual machine memory layout. Each program invocation receives four memory regions: program code, stack, heap, and serialised input accounts.

against the legal sBPF opcode set. For upgradeable programs, the ELF is stored in a separate *programdata* account, allowing updates without changing the program's address.

Virtual machine memory layout. The sBPF VM provides each invocation with a 64-bit virtual address space divided into four regions (Figure 2.5): a read-only program region containing the ELF code, a stack and heap for execution state, and an *input region* at `0x4_0000_0000` that contains the serialised account metadata and instruction data. On entry, register `R1` points to the start of this input region. All memory accesses are bounds-checked; out-of-bounds accesses terminate the program.

Account serialisation. Before invoking a program, the runtime serialises each account's metadata and data into a contiguous buffer with 8-byte alignment, laid out as a fixed-format record per account (duplicate flag, signer flag, writable flag, executable flag, public key, owner, lamports, data length, data, rent epoch). After all accounts, the buffer contains the instruction data length, the instruction data bytes, and the 32-byte program ID. This layout means a program can compute the virtual address of any account's data field by walking the serialised buffer from `R1`. After execution completes, the runtime deserialises the input region back into account state, applying any changes the program made to lamports or data fields.

Cross-program invocation. Programs interact with the runtime through syscalls. The most relevant for AMMs is Cross-Program Invocation (CPI): the `sol_invoke_signed` syscall dispatches an instruction to another program, for example an AMM invoking the Solana Program Library (SPL) Token program for a transfer.

2.3 Market Microstructure

Market microstructure theory studies how prices form and liquidity is provided in financial markets.

2.3.1 Adverse Selection and the Bid-Ask Spread

A market maker quotes two prices: a *bid* (the price at which it will buy) and an *ask* (the price at which it will sell). The difference between them is the *bid-ask spread*, which represents the cost of trading and the market maker's primary source of revenue. *Adverse selection* arises when some counterparties possess superior information about the asset's true value, causing the market maker to systematically lose on trades against them.

Glosten and Milgrom [21] show that in a quote-driven market with competitive dealers, the presence of informed traders generates a strictly positive bid-ask spread even when dealers earn zero expected profit. The intuition is that dealers set the ask (bid) equal to the expected asset value conditional on the next trade being a buy (sell), and since informed traders select the direction, the spread compensates for this adverse selection. The spread widens with the fraction of informed traders.

Kyle [30] formalises a complementary model in which a single informed insider, noise traders, and competitive market makers interact through a batch auction. The market maker sets prices according to a linear rule $P(y) = \mu + \lambda y$, where μ is the unconditional expected asset value, y is total order flow, and λ is the *price impact coefficient* that measures how much each unit of net flow moves the price. Higher λ implies lower market depth (i.e., less liquidity).

These models establish the fundamental trade-off that all market makers face, including propAMMs: quoting tighter spreads attracts more flow but increases exposure to informed counterparties who trade on superior information.

2.3.2 Optimal Market Making

Avellaneda and Stoikov [7] derive the optimal quoting strategy for a risk-averse dealer who faces stochastic order arrival. The model produces two key results: the dealer's *reservation price* shifts away from mid-price in proportion to inventory, encouraging mean-reversion of the position; and the optimal spread decomposes into an inventory-risk component and an order-arrival component that trades off spread revenue against execution probability.

This framework maps directly to propAMM design: the reservation price adjustment corresponds to the inventory-dependent quote skewing observed in several protocols (Section 6.3.2), and the spread decomposition provides a methodology for interpreting the diverse spread strategies we recover.

2.3.3 Spread Decomposition and Flow Toxicity

Huang and Stoll [26] decompose the bid-ask spread into three components: adverse selection (the permanent price impact of a trade, reflecting information content), inventory holding cost (transient quote adjustment to manage risk), and order processing cost (the fixed cost of providing quotes). Their empirical methodology uses the serial correlation of trade directions to separate the permanent (adverse selection) component from the transient (inventory) component.

For measuring adverse selection at the per-trade level, the *markout* approach compares the execution price against a reference price observed at a future horizon τ: positive markout indicates the liquidity provider profited on the trade, while negative markout indicates the price moved against the provider, i.e. the trader was better informed than the operator at the time of the quote. Flow that consistently produces negative markout is termed *toxic*. Lehar and Parlour [31] apply this methodology to Uniswap, and we adapt it for propAMM evaluation in Section 6.3.3.

2.4 Program Analysis Foundations

The reverse-engineering methodology in Chapter 4 draws on the program analysis techniques introduced here.

2.4.1 Program Slicing

Weiser [45] introduced *program slicing* as a decomposition technique that reduces a program to the minimal subset of statements that can affect a specified variable at a specified location. A *slicing criterion* $C = (i, V)$ consists of a program point i and a set of variables V; the *slice* with respect to C is the transitive closure of all statements that influence V at i through data or control dependencies.

Korel and Laski [29] introduced *dynamic program slicing*, where the slice is computed with respect to a single concrete execution rather than all possible executions. Because the actual values of array indices, pointer targets, and branch conditions are known from the execution trace, dynamic slices are significantly smaller and more precise than static slices (Figure 2.6). Agrawal and Horgan [3] formalised this as the Dynamic Dependence Graph (DDG), where each execution instance of a statement receives its own node, enabling precise tracking of dependencies across loop iterations. Zhang, Gupta, and Zhang [47] further developed algorithms for computing precise dynamic slices efficiently by traversing the DDG backward from the slicing criterion.

Our backward value tracing (Section 4.1.3) follows the same approach: given an execution trace and a target output value, we compute the set of instructions that contributed to that value by following explicit dataflow edges backward through the trace.

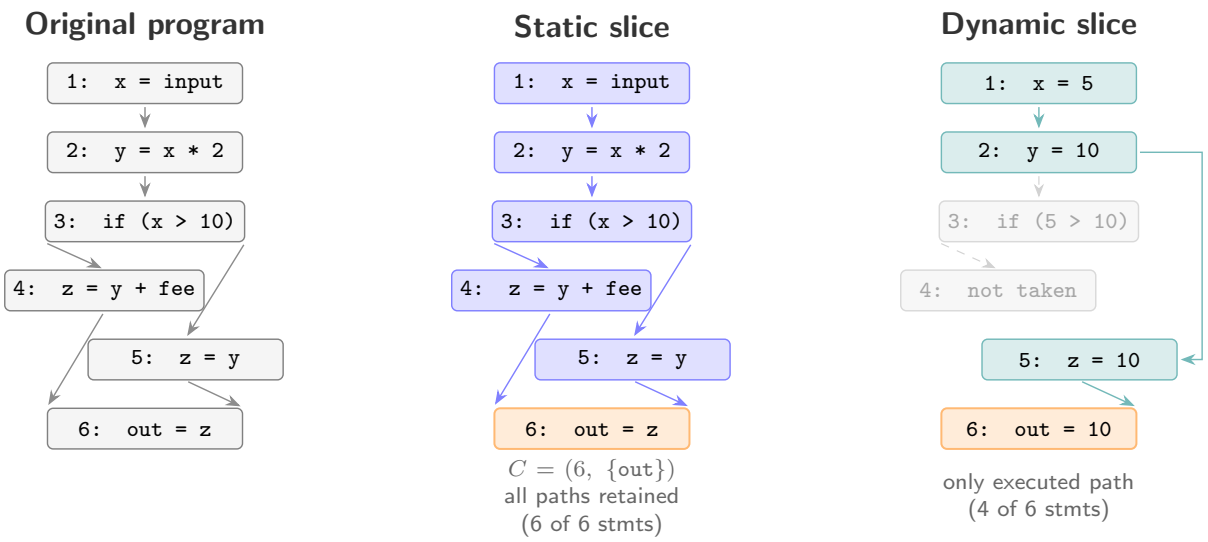


Figure 2.6: Static vs. dynamic program slicing for criterion $C = (6, \{\texttt{out}\})$. The static slice keeps all statements on any path that could affect the output. The dynamic slice, computed for a specific execution (input = 5), keeps only the statements actually executed, leading to a smaller and more precise slice.

2.4.2 Dynamic Taint Analysis

Dynamic Taint Analysis (DTA) tracks how data flows through a program by marking certain values as *tainted* (originating from a specified source) and propagating the taint label through arithmetic, memory, and control operations. Standard DTA propagates taint only along explicit dataflow edges (register-to-register, memory load/store), which can miss *implicit flows* through control dependencies: when a tainted value determines which branch is taken, the values computed on the taken path are influenced by the taint but are not marked as tainted by standard rules.

Kang et al. [28] address this with DTA++, which uses *targeted control-flow propagation*: for each branch where the condition is tainted, the not-taken path is concretely executed to determine whether the branch actually influences the output. Only branches that produce different outputs on the alternative path have their taint propagated, avoiding the overtainting that results from propagating through all control dependencies. We adapt this technique for control dependency resolution in Section 4.1.4.

2.4.3 Dominator Trees

A node d *dominates* node n in a control flow graph if every path from the entry to n must pass through d. The *immediate dominator* of n is the closest strict dominator. *Post-dominators* are the dual concept on the reversed Control Flow Graph (CFG): node p post-dominates n if every path from n to the exit passes through p. The Immediate Post-Dominator (IPDOM) of a conditional branch is the earliest point where both the taken and not-taken paths reconverge.

Cooper, Harvey, and Kennedy [14] present a simple iterative algorithm for computing dominator trees that is faster in practice than the asymptotically superior Lengauer–Tarjan algorithm [32]. We use this algorithm to compute post-dominator trees for sBPF programs (Section 4.1.4), providing the stopping condition for alternative path execution in our control dependency analysis.

Chapter 3

Related Work

This dissertation makes two core contributions: a reverse-engineering methodology for recovering quoting formulas from closed-source programs, and an empirical analysis of proprietary AMM behaviour using market microstructure tools. This chapter analyses three related research areas: empirical analysis of AMM-based decentralised exchanges (Section 3.1), Miner Extractable Value (MEV) and order flow dynamics (Section 3.2), and blockchain binary analysis (Section 3.3).

3.1 LP Returns and AMM Profitability

The empirical AMM literature mostly focuses on CFMMs, where the contract distributes swap fees pro rata to depositors who carry the inventory risk. This differs structurally from the operator-as-principal model of propAMMs, where a single off-chain operator funds the pool, sets quotes, and accrues the entire trading PnL (Section 2.1.4). Milionis et al. [34] formalise the adverse selection cost borne by AMM liquidity providers as LVR. They show that the instantaneous LVR rate takes the form

$$\ell(\sigma, P) = \tfrac{1}{2}\,\sigma^2\,P^2\,\big|x^{*\prime}(P)\big|, \tag{3.1}$$

where σ is the instantaneous volatility of the marginal price P and $x^{*\prime}(P)$ is the slope of the AMM's reserve curve (the marginal liquidity). LVR is non-negative, non-decreasing, and predictable, meaning it represents a running cost that fee income must offset for LP provision to be profitable.

Lehar and Parlour [31] provide a comprehensive empirical study of the Uniswap protocol, analysing 95.8 million interactions from its inception in November 2018 through December 2022. They use markout analysis to decompose LP returns into fee income and adverse selection costs. Their key finding is that conditions exist under which AMMs dominate traditional limit order books, specifically when adverse selection costs are moderate and centralised-exchange transaction costs are non-trivial.

Heimbach, Schertenleib, and Wattenhofer [25] study LP risks and returns on Uniswap V3 and conclude that concentrated liquidity provision has effectively become a sophisticated-investor activity: meaningful returns require active position management and entail elevated financial risk. Cartea, Drissi, and Monga [12] extend the LVR framework with the concept of *predictable loss* (PL), which decomposes into a convexity cost (equivalent to LVR) plus an opportunity cost of locking capital in the pool. They derive closed-form optimal LP strategies for concentrated liquidity and show empirically that LPs on Uniswap V3 have on average traded at a loss.

Fritsch and Canidio [20] empirically measure LVR across Uniswap pools and find that V2 pools are more profitable than V3 for passive LPs. They also show that arbitrage losses decrease by 20–70% with faster block times, a finding directly relevant to Solana's 400 ms slots. Barbon and Ranaldo [8] compare market quality across centralised and decentralised exchanges, finding that gas fees are the principal friction limiting DEX execution quality. Park [37] shows that any invariant-based pricing function permits sandwich attacks and that linear pricing is not regret-free. These results highlight fundamental limitations of open AMM designs that propAMMs may avoid by not committing to a fixed invariant. Hasbrouck, Rivera, and Saleh [24] model concentrated liquidity positions as covered calls, providing a theoretical bridge between AMM LP economics and options theory. From this literature and the market microstructure foundations described in Section 2.3, we adopt three analytical tools for our empirical analysis (Chapter 6): the multi-horizon markout methodology of Lehar and Parlour, the spread decomposition framework of Huang and Stoll [26], and the LVR-based continuous-hedging PnL metric of Milionis et al.

Bergault et al. [9] propose an alternative AMM design where the pricing function incorporates an external price signal, showing via mean-variance analysis that even a lagged oracle-based pricing function outperforms passive CFMMs in reducing adverse selection for LPs. Their work motivates the oracle-driven pricing adopted by the propAMMs studied here.

3.2 MEV and Order Flow in DeFi

Daian et al. [16] introduced the term MEV and provided the first systematic study of the profit a block producer can extract by including, excluding, or reordering transactions. They documented widespread deployment of arbitrage bots on Ethereum DEXs that exploit AMM price staleness through frontrunning, and argued that sufficiently large MEV can incentivise chain reorganisations, threatening consensus stability. PropAMMs respond to this: by tracking oracle prices and adjusting quotes dynamically, they reduce the staleness arbitrageurs exploit, internalising price discovery rather than leaving it to external searchers. On Solana, the MEV landscape differs from Ethereum (continuous block production, no public mempool, Jito-based tip mechanisms), but the fundamental dynamic of informed traders extracting value from stale quotes remains.

Chitra et al. [13] analyse *intent-based markets*, where users submit declarative trade specifications and third-party solvers compete to satisfy them. Their model shows that solver competition is subject to congestion externalities, and counter-intuitively, a social planner may prefer to restrict solver entry. PropAMMs on Solana operate within this framework: aggregators like Jupiter route user intents to whichever venue offers the best price, and propAMMs compete to be that venue.

Capponi, Jia, and Zhu [11] study the paradox of just-in-time (JIT) liquidity provision, showing that JIT LPs can detect toxic versus uninformed flow and selectively provide liquidity, potentially crowding out passive LPs. PropAMMs institutionalise the JIT strategy: the operator continuously adjusts quotes to incoming flow, achieving the selective liquidity provision that ad-hoc JIT providers perform per-transaction.

3.3 Blockchain Binary Analysis

Smart contract analysis has been an active area of research, predominantly focused on the Ethereum Virtual Machine (EVM). Grech et al. [22] present Elipmoc, a decompiler for EVM bytecode that achieves 99.5% fully resolved operands through transactional context sensitivity and path-sensitive function reconstruction. EVM decompilation is challenging because deployed contracts are stack-based bytecode with no explicit function boundaries and implicit jump targets.

The sBPF instruction set used by Solana differs structurally from the EVM: it is register-based (11 general-purpose registers), has explicit function call/return conventions, and programs are compiled from Rust or C rather than Solidity. Yuan et al. [49] present the first complete formal semantics of the Solana sBPF instruction set in Isabelle/HOL, including Solana-specific deviations from standard eBPF (such as signed 32-bit addition). Their validation framework uncovered discrepancies in the Solana VM implementation, confirming that sBPF semantics require careful treatment.

On the Solana side, Cui et al. [15] present VRust, the first static analysis tool for Solana programs, operating on Rust MIR (mid-level intermediate representation) rather than compiled bytecode. Smolka et al. [41] develop FuzzDelSol, a coverage-guided fuzzer for Solana programs that operates on compiled binaries without source code and uses dynamic taint tracking for vulnerability detection. Both tools operate on the same sBPF binaries that our methodology targets (Chapter 4), but focus on security (vulnerability detection and fuzzing) rather than recovering the mathematical semantics of deployed programs.

More broadly, the dynamic program slicing literature [3, 47] provides the algorithmic foundation for trace-based backward analysis: given a concrete execution trace and a target variable, a dynamic slice identifies the minimal set of executed statements that contributed to the target's value. Our backward value tracing (Section 4.1.3) can be understood as a dynamic backward slice over an sBPF execution trace. For resolving implicit information

flows through control dependencies, Kang et al. [28] introduced DTA++, which concretely executes the not-taken branch path to determine whether a conditional branch influences the output; we adapt this technique for our setting (Section 4.1.4).

In the broader binary analysis literature, Udeshi et al. [44] present REMaQE, a framework for recovering mathematical equations from native (x86/ARM) executables using symbolic execution and algebraic simplification. REMaQE achieves high accuracy on simple mathematical functions but targets single-equation recovery from native binaries, whereas our setting requires recovering multi-step quoting formulas with control dependencies from sBPF bytecode. Our approach (Section 4.1.3) differs in using concrete execution tracing and backward dataflow analysis rather than symbolic execution, which avoids path explosion in the complex branching logic of real-world AMM programs.

Traditional decompilation and binary lifting frameworks (e.g., angr, Ghidra, Binary Ninja) produce control flow graphs and pseudocode but do not extract mathematical formulas. Our problem requires understanding of how account data fields are combined arithmetically to produce outputs, motivating the trace-based backward analysis of Chapter 4.

3.4 Gap Analysis

The literature reveals three gaps that this dissertation addresses:

1. **No formula recovery from sBPF bytecode.** Existing binary analysis for Solana focuses on security (vulnerability detection, fuzzing). To our knowledge, no prior work addresses the recovery of closed-form mathematical expressions from deployed sBPF programs, whether through execution tracing or other techniques. Our methodology (Chapters 4 and 5) fills this gap by combining instrumented execution, backward value tracing, and DTA++-inspired control dependency resolution.

2. **No empirical study of proprietary AMMs.** Despite capturing a substantial share of DEX volume on Solana, propAMMs have, to our knowledge, received no academic attention. While industry reports discuss specific protocols at a descriptive level, no systematic empirical analysis of their quoting behaviour exists in the academic literature. The existing empirical AMM studies (Lehar and Parlour, Heimbach et al., Cartea et al.) all analyse open-source protocols where the pricing function is known. Our results (Chapter 6) provide what we believe to be the first systematic characterisation of propAMM quoting behaviour, flow composition, and LP profitability.

3. **No analytical pipeline from binary to financial measurement.** Recovering a quoting formula and applying microstructure tools (markout analysis, spread decomposition, inventory dynamics) are independent capabilities; on closed-source on-chain market makers, neither has been used to enable the other. This dissertation

uses the recovered quoting functions as the analytical instrument that converts raw on-chain state into spread decompositions, markouts, and operator-PnL estimates.

Chapter 4

Approach

We aim to provide a comprehensive analysis of propAMM behaviour on Solana. This requires understanding i) the functional properties and parameters of propAMMs (i.e. their quoting curves and parameters), and ii) the empirical patterns of parameter updates over time and how they relate to observable on-chain signals such as inventory imbalance, oracle price moves, and incoming order flow.

To address (i), Section 4.1 describes the toolchain we built for reverse-engineering propAMM bytecode, and Section 4.2 describes the formula-recovery workflow we apply on top of it to recover closed-form quoting formulas and validate them against the binary; the overall pipeline is shown in Figures 4.1 and 4.2. To address (ii), Section 4.3 describes the infrastructure for recording on-chain state updates that feed the empirical evaluation. A detailed walkthrough of the recovery process for one protocol is provided in Appendix A.1.

4.1 Toolchain for Reverse-Engineering

As described in Section 2.2.3, Solana programs are compiled to sBPF bytecode and deployed as opaque binaries with no source code or documentation. We aim to recover the quoting function of each propAMM, which we define as a function

$$Q\colon \mathcal{S} \times \{A \to B,\ B \to A\} \times \mathbb{N} \ \to\ \mathbb{N},$$

where A and B are the pool's two tokens (e.g. SOL and USDC), so $\{A \to B,\ B \to A\}$ is the set of swap directions and $\mathbb{N}$ represents non-negative integer token amounts. The state space $\mathcal{S} = \mathcal{S}_v \times \mathcal{S}_p$ is the observable on-chain state, comprising vault balances $s_v \in \mathcal{S}_v$ and pool parameters $s_p \in \mathcal{S}_p$. Given a state $s \in \mathcal{S}$, a swap direction d, and an input amount x, the function returns the output amount $y = Q(s, d, x)$.

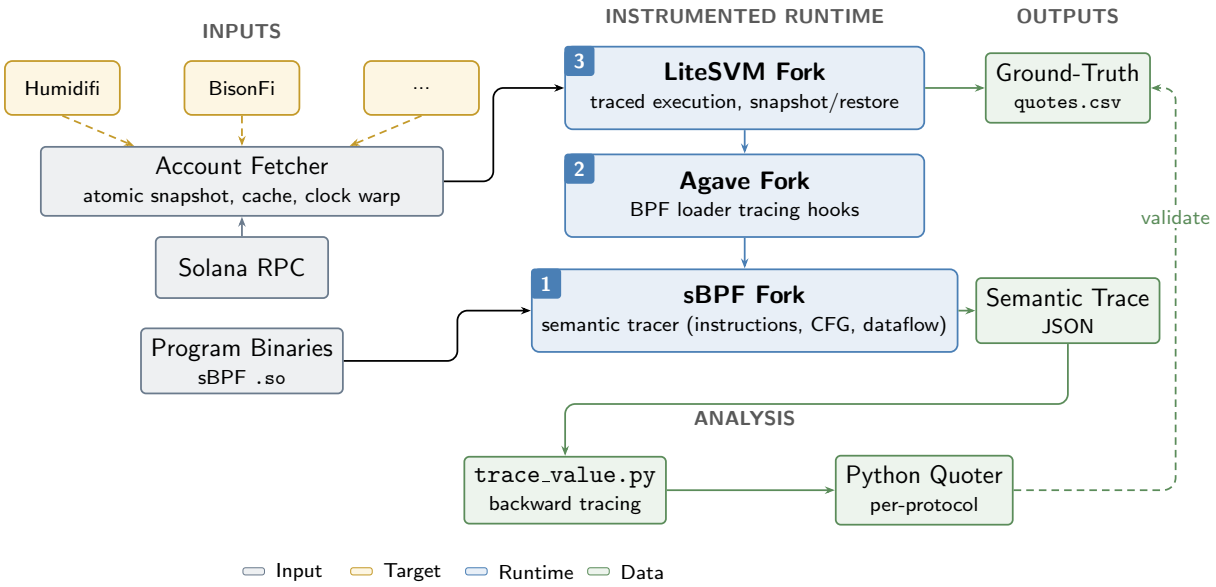


Figure 4.1: Methodology overview. PropAMM program binaries and account state are fetched from Solana mainnet. Binaries are executed by an instrumented sBPF interpreter (1), driven through patched Agave loader hooks (2) inside a forked LiteSVM [33] runtime (3) that supplies the account state. The runtime captures semantic execution traces, from which a backward analysis tool reconstructs the quoting function; the result is validated against the ground-truth output, with mismatches driving further iteration.

4.1.1 Local Transaction Simulation

Recovering Q requires executing the on-chain program with controlled inputs and real account state (Figure 4.1, left column). We cannot do this on mainnet, since each execution would require a real buy or sell transaction, so we simulate locally using a fork of LiteSVM [33], a lightweight Solana virtual machine designed for testing.

The simulation pipeline works as follows. First, all accounts required by the target pool (pool state, token vaults, mints, oracles, and any protocol-specific accounts) are fetched from mainnet Remote Procedure Call (RPC) in a single atomic call. These accounts are injected into the local VM together with the program's compiled `.so` binary, which is obtained via `solana program dump`. The VM's clock is warped to match the slot at which accounts were fetched, since several protocols include staleness checks that reject transactions when the clock slot does not match internal timestamps.

Each simulated swap is constructed as a transaction that routes through an intermediary program (via CPI), matching the production execution path. To generate a set of (x, y) ground-truth pairs, we sweep over a range of input amounts. Between each swap, all account state is restored to ensure each quote reflects the same initial pool state.

4.1.2 Instrumented Execution

We instrument the sBPF interpreter to capture an *execution trace*: a per-instruction record of operand values, register state changes, memory accesses, and dataflow.

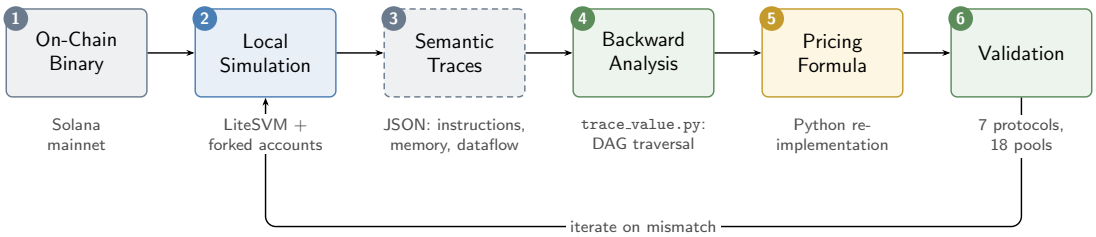


Figure 4.2: End-to-end reverse engineering pipeline (circled numbers indicate order). On-chain binaries (1) are simulated locally (2), producing semantic traces (3). A backward dataflow tool (4, `trace_value.py`) traverses the trace's def–use DAG to produce a computation skeleton, which we simplify and re-implement as a closed-form Python pricing function (5). The result is validated (6) against live pool state, with mismatches driving further iteration.

The instrumentation wraps every call to the interpreter's `step()` function. Before each instruction executes, the tracer captures the pre-execution register state. After execution, it captures the post-execution state, computes the register diff, and records the instruction's operand values. Each instruction receives a monotonically increasing *execution sequence number* (`exec_seq`) that uniquely identifies every instruction execution, even across iterations of the same loop at the same program counter.

We also add a `source_def` field on each operand. For every value consumed by an instruction, the tracer records the `exec_seq` of the instruction that *produced* that value. This is implemented by maintaining a `register_defs` array that tracks, for each register, the identifier of the most recent instruction that wrote to it. When an instruction's operands are extracted, `register_defs` is read *before* the current instruction's definition is recorded, ensuring the backward link points to the previous producer.

A parallel `DataFlowAnalyzer` classifies each value's origin as one of: `Computed` (result of an arithmetic or bitwise operation), `MemoryLoad` (read from a VM memory address, with a link to the most recent store at that address), `RegisterCopy`, `Constant` (immediate or load from read-only data), or `EntryParameter`. Memory stores are tracked in an address-keyed table, enabling alias analysis: when a load instruction reads from an address, the analyser resolves it to the most recent store, linking loads to stores and stores to their value sources. Figure 4.3 illustrates this pipeline, showing three consecutive instruction executions with their `source_def` backward edges, the value origin classification, and account address resolution.

After execution, the runtime populates an `input_accounts` table that maps each VM memory address range to its corresponding on-chain account public key, owner, data length, and access flags based on the order of accounts passed to the program as described in Section 2.2.3. This enables the backward analysis to resolve a raw memory access at address `0x500001A0` to "account `51FQwj...` at byte offset `0x120`". The result is a `TraceContext` containing the full hierarchical execution trace, the dataflow state, and the account layout. Trace files are typically 8–60 MB of JSON per execution, depending

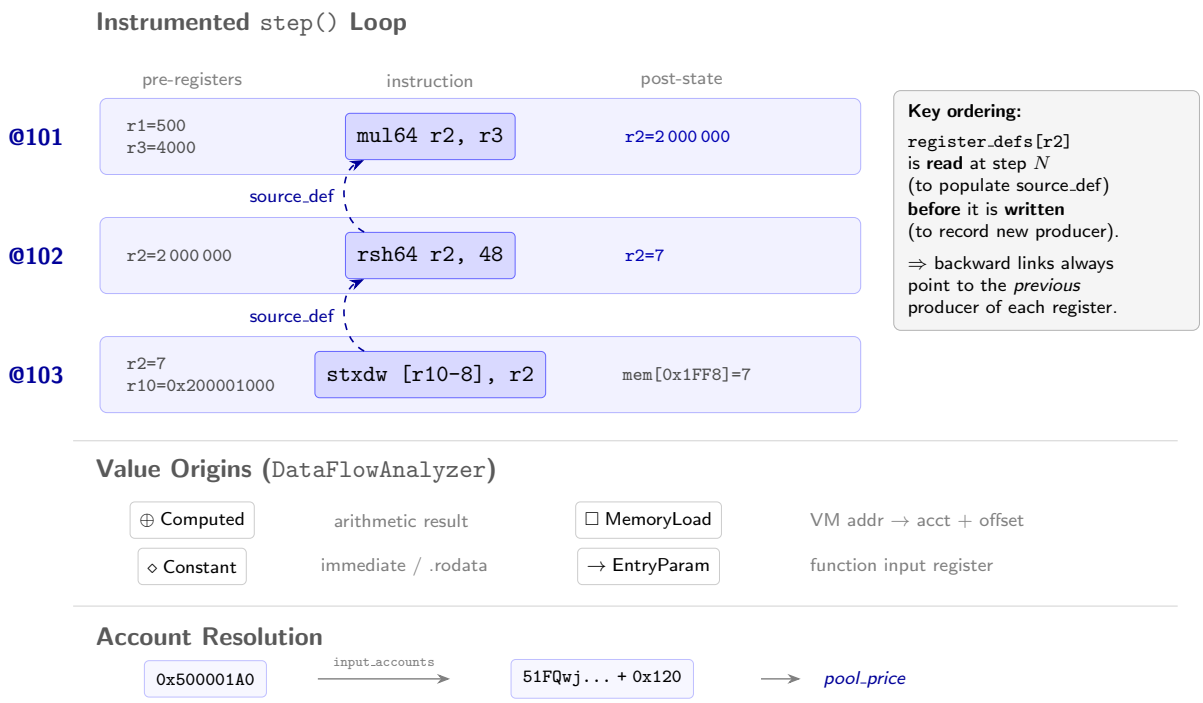


Figure 4.3: Per-instruction semantic tracing. Top: three consecutive instruction executions with pre/post register state and source_def backward edges. Right: the read-before-write ordering of `register_defs` that ensures backward links point to the previous producer. Bottom: value origin classification and account address resolution.

on program complexity.

4.1.3 Backward Formula Reconstruction

Given a semantic trace and a target output value y, the backward analysis tool produces a topologically sorted pseudocode reconstruction of the computation that produced y, by walking `source_def` edges backward from output to leaves. Since every value in the trace is either derived from (i) the input data, (ii) read from an account or from (iii) a hard-coded constant in the program, we can treat (i), (ii), and (iii) as leaf nodes in the computation graph. The algorithm starts at the instruction with the highest `exec_seq` that wrote y via an arithmetic operation, then recursively dispatches on operand type. Every case falls into one of three actions:

Recursive descent. Arithmetic and bitwise operations (add, sub, mul, div, shifts, etc.) have both operands traced recursively via their respective `source_def` edges. These are the only cases that add new nodes to the computation graph.

Leaf nodes. Memory loads from account data are resolved through the `input_accounts` table to an (account public key, byte offset) pair and recorded as a leaf; semantic names such as "pool price" are assigned later; the intuition and methodology behind this are covered in Section 4.2. Constants and immediates are recorded as leaves directly. In both cases the traversal terminates.

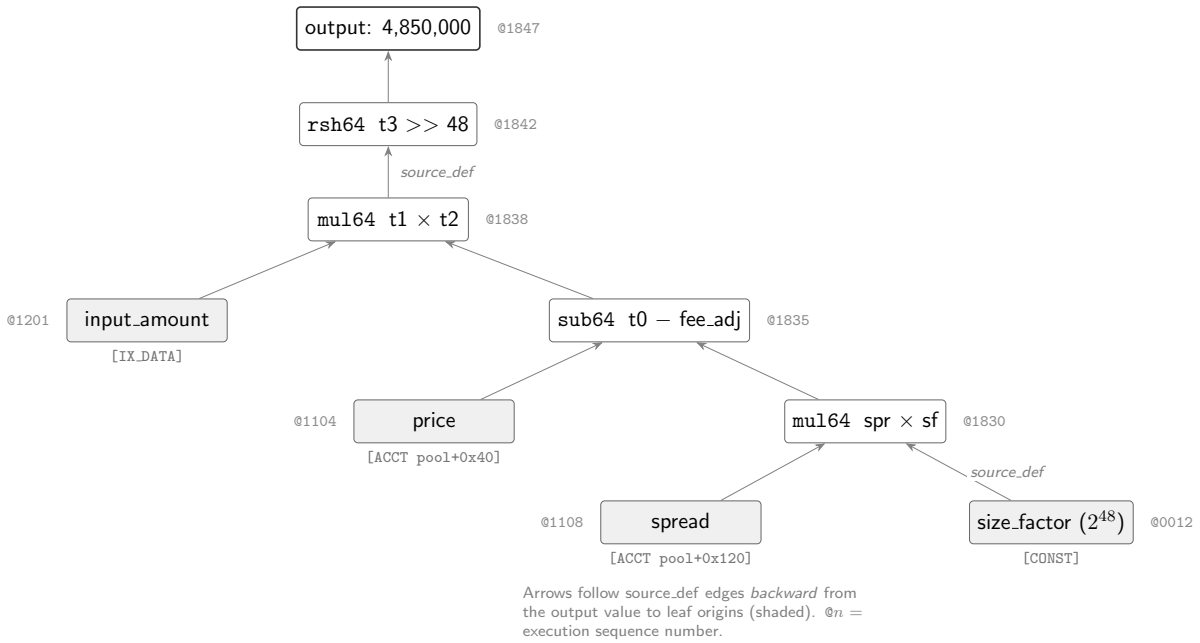


Figure 4.4: Backward value tracing through an sBPF execution trace. Starting from the output value, source_def edges are followed to reconstruct the arithmetic expression tree, terminating at leaf values loaded from account data, instruction data, or constants.

Transparent indirection. Memory loads from the stack are resolved by finding the most recent store at that stack address and following the stored value's `source_def`. Register moves are followed to the source register's producer. Function returns (`exit`) emit r0's `source_def`, so the return value is traced like any other register; callee-saved registers (r6–r9) are similarly traced via their pre-call `source_def`. None of these cases produce a new node in the output: they are pass-throughs that the traversal walks across.

Edge cases. Values that pass through `sol_memcpy_` syscalls are followed through the memcpy source and destination addresses, up to a chain depth of 10. Misaligned reads (e.g. a 4-byte load from the middle of an 8-byte store) are resolved by searching for overlapping wider stores and extracting bytes at the correct offset.

Output and validation. The result is a topologically sorted pseudocode listing from leaves to y. A built-in verification step re-executes the listing under 64-bit unsigned overflow semantics, confirming that every intermediate value matches the trace; this catches recovery bugs immediately. Figure 4.4 shows a simplified example.

4.1.4 Control Dependency Resolution

The backward dataflow analysis recovers the arithmetic computation graph but misses *control dependencies*: conditional branches that determine which computation path executes without producing values directly. For example, a tier-selection branch picks which fee multiplier is used, but only the multiplier's value appears in the dataflow; the branch

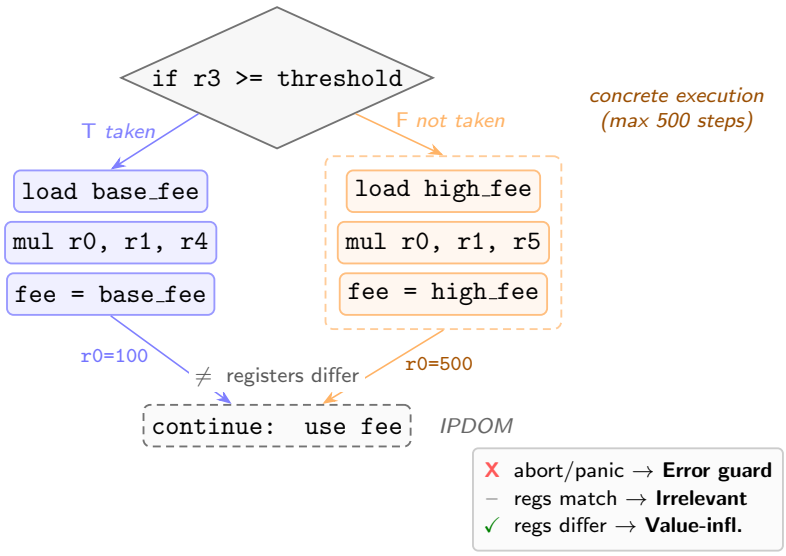


Figure 4.5: DTA++ branch classification for control dependency resolution. When backward tracing reaches a tainted conditional branch, the not-taken path is concretely executed up to the immediate post-dominator (IPDOM) using `sbpf_interp.py`.

condition that selected it does not. We resolve control dependencies using two complementary techniques from the program analysis literature.

Static post-dominator analysis. For each conditional branch in the program, we compute its IPDOM: the nearest program point that all paths from the branch must pass through. The IPDOM is computed using the iterative dominator algorithm of Cooper, Harvey, and Kennedy [14] applied to the reversed CFG. The ELF binary is parsed to extract the `.text` section, instructions are decoded into basic blocks, functions are identified by reachability from call targets, and the post-dominator tree is computed per function. The IPDOM serves as the stopping condition for alternative path execution (below).

Dynamic alternative path execution. To determine which branches actually influence the output, we adapt the alternative path execution concept from DTA++ [28] to our backward tracing setting. Where the original DTA++ propagates taint forward through x86 binaries, we use the same core idea (concretely executing the not-taken path and comparing outcomes) but apply it backward from a known output value. For each conditional branch near a traced computation, a minimal sBPF interpreter executes the alternative path from the branch point, using the register state captured from the trace. Execution continues until the IPDOM is reached, the program aborts, or a step limit (500 instructions) is exceeded. The branch is then classified into one of three categories:

1. **Error guard**: the alternative path triggers an abort or panic. The branch is a runtime safety check and does not influence the output. It is skipped.

2. **Irrelevant**: the alternative path reaches the IPDOM with all register values matching the actual trace. The branch does not affect the computation. It is skipped.

3. **Value-influencing**: the alternative path reaches the IPDOM with at least one register differing. The branch selects between different computation paths and is included in the output with full metadata about which registers differ and what their alternative values would have been.

For value-influencing branches, the condition operands are themselves traced backward through the dataflow graph. To prevent recursive explosion (tracing a condition discovering more branches), control dependency analysis is temporarily disabled during condition operand tracing. Figure 4.5 illustrates the classification process.

4.1.5 Soak Testing Harness

To turn a candidate Python quoter into a validated one, the final piece of the toolchain is an automated soak-testing harness that exercises the quoter against the real on-chain binary (Figure 4.2, stages 5–6). Each iteration fetches a fresh snapshot of all pool accounts from mainnet, executes swaps across a range of input amounts in both directions using the real `.so` binary in LiteSVM, and compares the output against the Python quoter's prediction. The comparison uses an absolute tolerance of ± 1 token unit (the smallest representable amount) to account for integer rounding differences.

A typical run consists of 100 iterations. Because each iteration's snapshot is captured a few seconds apart, successive iterations exercise the quoter against genuinely different live pool states: different vault balances, different operator-set parameters, and different oracle prices. Within each iteration, we test 100 linearly spaced input amounts across both swap directions, for a total of 20,000 comparisons per run. When a mismatch is detected, a self-contained regression snapshot (all account data, input amount, expected output) is saved automatically; these snapshots are the input to the diagnostic recipe in Section 4.2.2. Binary freshness is monitored by periodically re-downloading each program's `.so` binary from mainnet and comparing MD5 hashes; any upgrade triggers a re-validation pass.

4.2 Formula Recovery

4.2.1 From Trace to Formula

The toolchain of Section 4.1 hands us a topologically-sorted pseudocode skeleton of `u64` operations ending in tagged leaves: `[IN]` from accounts (with public key and byte offset), `[ARG]` from instruction data, `[CONST]` from immediates, and `[SYS]` from sysvars. For example, the skeleton for a two-limb Q88 coefficient assembly might look like this. The implementation of this toolchain is described in more detail in Section 5.4.

```
[IN]    x1 = 0x1b89840000000000   @0x400002c00 [pool+0x340]
[IN]    x2 =         0x5d5c7812   @0x400002c08 [pool+0x348]
t1     = (x2 << 64) | x1
```

```
...
result = ((net * t1) >> 88) // 1000
```

However, a closed-form formula is not directly derivable from a trace like this. For this, we need to interpret the arithmetic shape of the trace, assign semantic meaning to each leaf, and hypothesise how they fit together into a formula that matches the trace's structure while also making sense as a quoting function. The remainder of this subsection summarises the heuristics; the worked example in Appendix A.1 is the most useful resource for seeing how they combine in practice.

Pattern recognition. We match the trace's arithmetic shape against a small sample of assembly patterns that reoccur across the seven protocols (Table 4.1). A frequent pattern is Q-format fixed-point multiplication: we read `(x*c)>>K` as $x \cdot c/2^K$, with K often revealed by a shift constant; when a single Q-format value spans more than 64 bits, the binary stores it as several limbs joined by shift-and-or.

Trace shape	Interpretation
`(x*c)>>K`	Q-format multiply $x \cdot c/2^K$
`(hi<<64)|lo`	Multi-limb assembly of a wide coefficient (e.g. `u128`)
`(x+d-1)//d`	Ceiling division $\lceil x/d \rceil$
`x*r//d; x-...`	Fractional deduction (fee shape)
`min(rem, V*m/10^4)`	Inventory-bounded chunk consumption
`(clk-slot)*rate`	Staleness delta scaled into a penalty

Table 4.1: Recurring assembly idioms across the seven propAMMs studied.

Semantic naming. The toolchain identifies leaves as `(account, offset)`; naming them requires interpreting the role of the bytes at that offset in the context of the pool's state layout. The role of the parameters either comes from a) known values that we pattern-match (e.g. the vault balance is a known referent for the pool's inventory, the current slot of the clock sysvar is a known referent for time, etc.) or b) derived from the context during reverse-engineering. Again, the walkthrough of Appendix A.1 shows this in practice.

The iteration loop. We bootstrap the process with a single trace at a small input, which is enough to implement an initial `quoter.py`. We then iteratively refine this quoter by running it against the binary in the soak harness, capturing every disagreement as a regression snapshot, and applying the diagnostic recipe of Section 4.2.2 to turn each snapshot into a quoter patch and a permanent regression test. Every newly observed field is subjected to the same pattern-recognition and naming procedure.

Where judgement is irreducible. While our toolchain greatly simplifies the discovery and validation of the quoting formula, it does not fully automate it. The interpretation

of traces into formulas relies on human judgement and domain knowledge, especially in the early stages when the quoter is still far from the expected output.

4.2.2 From Test Failure to Fix

For every input on which the candidate quoter and the on-chain binary disagree, the soak harness (Section 4.1.5) emits a self-contained regression snapshot, `(pool_state, vault_balances, clock, input, expected)`. We then turn each snapshot into a quoter patch and a permanent regression test as follows: binary-search the input axis within the frozen state to find the smallest failing amount (shorter traces tend to isolate a single faulty branch); re-trace at that input with the simulator's `--trace` flag and extract the pseudocode skeleton via `trace_value.py` (Section 4.1.3); diff against the quoter's `--debug` output, where the first unmatched intermediate value is the bug location; and patch the quoter, saving the snapshot under `regressions_<protocol>/` so subsequent soak runs replay it deterministically. The loop converges empirically rather than by proof; Appendix A.1 shows it running end-to-end on BisonFi.

4.3 On-Chain Observation

The reverse-engineered quoting function Q and account layout from Sections 4.1 and 4.2 tell us what each field in a pool's state means.

We compiled and deployed the Agave validator [5] on a server and used the Yellowstone gRPC plugin [39] to stream account updates in real time for all target pools. The non-voting validator participates in gossip but not consensus, receiving all account state changes with intra-slot timestamps and the transaction signatures that caused each change, allowing us to distinguish swaps from capital events (deposits, withdrawals, vault migrations). Raw account snapshots are decoded using the account layouts recovered during reverse-engineering (Section 4.1.2). The recovered quoting function is then applied to each detected swap to compute the implied spread, execution price, and markout at various horizons.

Chapter 5

Implementation

We implement the tracing across four systems: a patched sBPF interpreter that captures semantic traces, a local Solana VM that executes the program binaries, a Rust-based simulator that orchestrates account fetching and swap execution, and a Python analysis toolchain that performs backward formula reconstruction and validation. Table 5.1 summarises the resulting components and their sizes. All source code is released across two public repositories: the simulator, recovered formulas, and per-protocol soak tests live at `https://github.com/TraceSVM/propamm-simulator`, and the recording pipeline together with the empirical-analysis scripts used in Chapter 6 live at `https://github.com/TraceSVM/propamm-replay`.

Component	Language	LOC	Key files / dependencies
Tracing infrastructure	Rust	∼2,800	forked `solana-sbpf`, `solana-program-runtime`
LiteSVM fork	Rust	∼400	forked `litesvm` 0.5
Simulator	Rust	∼3,500	`solana-sdk` 2.2, `bytemuck`
Analysis toolchain	Python	∼4,200	`trace_value.py` (backward tracer) `sbpf_cfg.py` (static CFG / IPDOM) `sbpf_interp.py` (minimal interpreter) deps: `pyelftools`, `numpy`
Per-protocol quoters	Python	∼2,100	one file per protocol (`humidifi.py`, `alphaq.py`, . . .)
Total		**∼13,000**	

Table 5.1: Engineering summary. LOC counts exclude tests and generated code. Filenames in the right column are referenced individually in Section 5.4.

5.1 Tracing Infrastructure

The tracing infrastructure spans three custom forks of open-source Solana components plus a thin glue layer, all activated together by a single `semantic-tracer` Cargo feature flag (Figure 5.1). The simulator enables the feature on LiteSVM, which in turn enables it on three downstream crates: `solana-bpf-loader-program` (where traced execution is

invoked), `solana-program-runtime` (which threads trace state through the `InvokeContext`), and `solana-sbpf` (where per-instruction capture occurs).

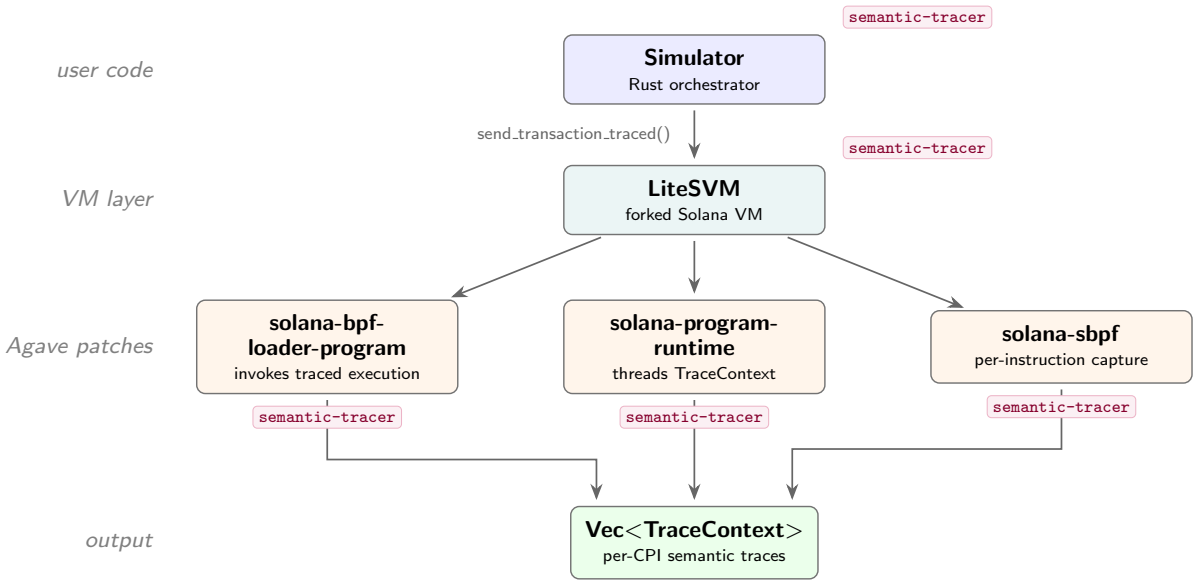


Figure 5.1: Feature flag dependency chain for the semantic tracing infrastructure. The `semantic-tracer` Cargo feature propagates from the simulator through LiteSVM into three patched Agave crates, producing a `Vec<TraceContext>` with per-CPI traces.

sBPF interpreter (per-instruction capture). The core modification is to the sBPF interpreter's execution loop. A `Tracer` struct wraps the interpreter and replaces its free-running `execute_program()` call with a per-instruction `traced_step()` loop that captures pre-execution register state, invokes the unmodified `interpreter.step()`, then records the register diff with the instruction's operand values and their `source_def` backward links into the dataflow analyser. A monotonically increasing `exec_seq` counter tags every instruction execution, so successive iterations of the same loop remain distinguishable.

Dataflow analyser (preserving value provenance). A `DataFlowAnalyzer` runs alongside the tracer and records, for every value produced during execution, a `ValueDefinition` keyed by `DefId` = (pc, seq, exec_seq). Each definition records the value, its `ValueOrigin` (Computed, MemoryLoad, RegisterCopy, Constant, EntryParameter, SyscallReturn, or FunctionReturn), a taint set tracking which input sources (memory regions, entry arguments, syscall results) contributed to it, and the `DefId`s of its inputs. Memory stores are tracked in an address-keyed table; when a load reads from an address, the analyser resolves it to the most recent store, creating the alias chain that enables backward tracing through stack-mediated dataflow. Syscalls to `sol_memcpy_` and `sol_memmove_` are handled by creating synthetic store definitions for each 8-byte chunk, preserving dataflow continuity through memory copies.

LiteSVM fork (driving traced execution). The LiteSVM fork exposes a `send_transaction_traced()` method that returns both the transaction result and a `Vec<Trac`

eContext>, one per CPI in the transaction; the protocol's swap CPI is then selected from the vector.

Account layout resolution (naming the leaves). The dataflow analyser captures memory loads as raw VM addresses; turning those into named account fields requires a second mapping. After traced execution, the Agave runtime populates the trace's `input_accounts` field by iterating over the instruction's accounts and recording each account's public key, owner, VM data address, data length, and access flags. The VM data address is the virtual address where each account's serialised data begins in the sBPF input buffer (memory region 4, starting at `0x400000000`).

5.2 Simulator

The Rust-based simulator wraps the tracing infrastructure and orchestrates the end-to-end pipeline from on-chain state to a traced quote.

Account fetching. All accounts required by a pool are listed in a per-protocol JSON configuration file. The simulator fetches them from mainnet RPC in a single atomic call so all account states share a slot, caching results locally to avoid redundant calls.

Clock warping. Several propAMMs include staleness checks that reject transactions if the runtime clock does not match an internal slot value stored in account data. The simulator therefore exposes a per-protocol `adjust_clock_slot()` hook that extracts this internal slot and sets the VM's slot, epoch, and epoch timestamp to a consistent point on mainnet's epoch schedule.

Transaction construction. Swaps are dispatched through Jupiter via CPI, matching the production execution path used by aggregators rather than calling the protocol directly. This lets a single transaction format work across all seven protocols, with the target identified by a discriminator field.

State isolation. To keep each quote independent, the simulator snapshots all relevant accounts before each swap and restores them afterwards; otherwise each swap would mutate vault balances and successive quotes would reflect a moving pool state.

5.3 Protocol-Specific Adaptations

Each of the seven reverse-engineered protocols required specific adaptations, since they deploy different swap-instruction versions and have unique account layouts, obfuscation schemes, and access-control mechanisms.

XOR obfuscation. Two of the seven protocols obfuscate pool or oracle data by Exclusive OR (XOR)-ing it with a fixed pad embedded in the program's read-only data section. Humidifi applies per-field XOR obfuscation with approximately 30 distinct pads (mostly 64-bit, some 16-bit), each recovered from the program's read-only data section. SolFi V2 XORs the entire oracle account against a 168-byte pad.

Sysvar requirements. ZeroFi reads the `LastRestartSlot` sysvar, which defaults to zero in LiteSVM but must be set to a valid mainnet value for the program to accept transactions. The simulator patches this sysvar during setup.

```
INPUTS (excerpt; 14 leaves total):
  [IN]    fp_val      =   1568301972   @0x400002c08 [8FnX3xo2..+0x348]
  [IN]    fee_mult    =          154   @0x400002c14 [8FnX3xo2..+0x354]
  [CONST] fee_denom   =      2560000                  # inline constant
  [CONST] dec_a       =          100                  # inline constant
  [CONST] dec_b       =           10                  # inline constant
  ; (9 byte-unpacking constants used to assemble t9 and t17 elided)

COMPUTATION (excerpt; 33 steps total):
  ; --- byte unpacking (t0..t17) elided: assembles t9 and t17 ---
  t18  = t9 - t17           = 1000000000        # input amount, reassembled
  ; --- fee deduction ---
  t20  = t18  * fee_mult                        # input * fee multiplier
  t21  = t20  // fee_denom  = 60156             # fee (denominator 2.56e6)
  t22  = t21  + 1           = 60157
  t23  = t18  - t22         = 999939843         # net amount after fee
  ; --- price multiplication ---
  t26  = t23  * fp_val                          # net * fp_val
  t27  = 849183970 + t26                        # rounding bias
  t28  = t27  >> 24                             # Q-format divide by 2^24
  ; --- decimal-scale post-divide ---
  t31  = dec_a * dec_b      = 1000              # decimal scale = 100*10
  result = t28 // t31       = 93472458          # USDC output (6 decimals)

VERIFICATION:
  Status: PASS (33 steps verified, all intermediate values match)
```

Figure 5.2: `trace_value.py` output for a 1 SOL base-to-quote swap on BisonFi pool `8FnX3xo2...` Byte-unpacking steps `t0`–`t17` are elided.

5.4 Analysis Toolchain

Reconstructing a quoting formula from a `TraceContext` uses three Python tools that consume the same JSON output, each contributing different structure.

Backward value tracer (`trace_value.py`). The primary analysis tool implements the backward tracing algorithm described in Section 4.1.3. It indexes the trace by value producers, `DefId` definitions, register write histories, account memory ranges, and memcpy chains, then performs recursive backward traversal via `trace_edges()`. The output is

topologically sorted pseudocode with semantic leaf names (e.g., "pool.reserve_a at offset `0x48`") and dead-end pruning. Figure 5.2 shows a representative excerpt: a 1 SOL base-to-quote swap on the BisonFi pool `8FnX3xo2..`, where the tool extracts the 33-step computation graph yielding 93 472 458 units of USDC ($93.47).

Static CFG analyser (`sbpf_cfg.py`). Parses sBPF ELF binaries, constructs basic blocks, identifies function boundaries, and computes post-dominator trees using the Cooper–Harvey–Kennedy iterative algorithm on the reversed CFG. The IPDOM for each conditional branch is exported for use by the DTA++ classifier.

Minimal sBPF interpreter (`sbpf_interp.py`). A lightweight Python interpreter implementing 69 sBPF opcodes with sparse byte-addressed memory and emulated syscalls. Used exclusively for DTA++ alternative path execution: running the not-taken branch from a snapshotted register/memory state until the IPDOM is reached.

With `--binary <prog.so>`, `trace_value.py` combines the CFG analyser and minimal interpreter to annotate each computation step with the branches that gated it. Figure 5.3 re-runs the example from Figure 5.2 with the DTA++ pipeline enabled. The output now interleaves `# BRANCH:` lines above each gated step, recording the branch condition, which side was taken, and how many interpreter steps the alternative path ran before reaching the IPDOM.

```
COMPUTATION (DTA++ excerpt; 7 control dependencies in the full trace):
                # BRANCH: if 0x400005920 == x1 (0): not taken, alt 4 steps
  t1   = x0 << 8       = 36352
  ; ... byte-unpacking elided ...
                # BRANCH: if x1 (0) == x1 (0): taken, alt 4 steps
  t57  = t46 - t56     = 1000000000     ; input amount, reassembled
                # BRANCH: if 106 <= 90: not taken, alt 13 steps
  t60  = t59 // const18 = 60156         ; fee
                # BRANCH: if t57 (1000000000) >= t63 (60157): taken, alt 4 steps
  t64  = t57 - t63     = 999939843      ; net after fee
                # BRANCH: if t122 (1000) == x1 (0): not taken, alt 6 steps
  t142 = t64 * x29                      ; net * fp_val
  t143 = 849183970 + t142
  t144 = t143 >> 24                     ; Q-format divide
```

Figure 5.3: Same trace as Figure 5.2, re-analysed with `--binary` to enable DTA++ control-dependency annotation. Guard and irrelevant branches are elided.

Per-protocol quoters. Each protocol has a standalone Python quoter that reimplements the recovered quoting formula using arbitrary-precision integer arithmetic. Quoters decode account data and provide a `get_quote()` function that matches the on-chain computation step by step, serving as both the validation reference for soak testing and the spread calculator for the empirical analysis.

Chapter 6

Results

6.1 Experimental Setup

6.1.1 PropAMM Selection

We use *propAMM* and *protocol* interchangeably for a distinct deployed program with its own operator and quoting logic; a *pool* is one market instance (a token pair with its own accounts and parameters) run under it. We selected seven propAMMs by volume share on Solana, availability of the deployed binary, and coverage of the SOL/USDC pair. Table 6.2 summarises their key architectural properties. All seven are closed-source; the quoting formulas presented in Section 6.2.1 were recovered entirely from on-chain bytecode analysis using the methodology described in Chapter 4.

6.1.2 Dataset, Methodology and Market Conditions

PropAMMs cover the most active pairs on Solana; the deepest is SOL/USDC, the focus of our empirical analysis. We selected all SOL/USDC pools across the seven protocols, representing 18 pools in total (some protocols have multiple pools with different parameters).

Infrastructure. For recording the account states, we compiled and deployed the Agave validator [5] on a bare-metal server located in Amsterdam (AMD EPYC 9254 24-core, 378 GB RAM, 2×3.5 TB NVMe). Amsterdam is centrally positioned on the Solana gossip network, giving low-latency account updates. This non-voting node participates in gossip but not consensus and records all account updates in its local ledger. We used the Yellowstone gRPC plugin [39] to stream account updates in real time, to get intra-slot recording data and timestamps. We also recorded the signatures of transactions causing account state changes to link state updates to specific events and distinguish between swaps and capital events (deposits, withdrawals, vault migrations). The 10-hour

Protocol	Pools	Active	Updates	Swaps	Volume	TVL	Upd./hr
Humidifi	5	5	371,916	53,857	$24.1M	$2.2M	37,192
BisonFi	6	4	350,701	19,426	$34.9M	$9.4M	35,070
GoonFi V2	1	1	81,706	33,644	$7.1M	$593K	8,171
AlphaQ	1	1	64,196	15,245	$15.9M	$292K	6,420
SolFi V2	2	1	112,037	10,676	$4.2M	$5.6M	11,204
ZeroFi	2	2	51,437	6,001	$3.8M	$1.2M	5,144
Tessera V	1	1	91,183	–	–	$2.8M	9,118
Total	**18**	**14**	**1.12M**	**138,849**	**$90.1M**	**$22.1M**	–

Table 6.1: Dataset summary for the 10-hour recording session. *Active* counts pools with ≥ 1 detected swap. Tessera V is excluded from swap detection because its vaults are shared across pools (see text). Volume is computed from vault balance deltas.

recording session produced 562 MB of raw state data (Parquet) and 151 MB of derived quoter output. Ten hours yielded enough swaps (138 849) while staying within a single calm market window, avoiding shifting volatility regimes.

Dataset. An overview of the recorded dataset (April 3–4, 2026) is given in Table 6.1. The recording captures 1.12 million state updates, 138,849 detected swaps, and $90.1 million in volume across six active protocols. During the observation window, SOL/USD traded in a range of $78.50–$80.50, representing low-to-moderate volatility conditions. We detected swaps by filtering for state updates where both vault balances changed with opposite signs. In the dataset there was one withdrawal event from one Humidifi pool, which we accounted for in PnL calculations.

6.2 Reverse-Engineering Results

Using the semantic execution tracing methodology from Chapter 4, we recovered closed-form quoting functions for all seven protocols. Each function maps the observable on-chain state $s \in \mathcal{S}$, a swap direction d, and an input amount x to an output amount $y = Q(s, d, x)$, as defined in Chapter 4. We present the recovered formulas below grouped by architectural family; the complete specifications with all parameter definitions, account-field offsets, and rounding rules are given in Appendix A.

6.2.1 Recovered Quoting Formulas

Notation and shared parameters. All seven protocols share the following inputs and conventions. We write x for the input amount, y for the output, P for the operator-set mid-price, and $d \in \{\text{B2Q}, \text{Q2B}\}$ for the swap direction (base-to-quote, e.g. SOL→USDC, or the reverse). v_a and v_q denote the base and quote vault balances; S_{eff} is a decimal-adjusted price scale (typically 10^{10}, varying with token decimals). Several protocols derive an *inventory measure* $\text{inv} = \lfloor v_a \cdot P / S_{\text{eff}} \rfloor$ (the quote-equivalent value of base reserves) and

Protocol	Group	Output structure	Staleness	Inventory	Asymmetric
AlphaQ	A	$y = x \cdot P_{\text{eff}}$	–	✓	✓
Humidifi	A	$y = x \cdot P / D_{\text{eff}}$	–	✓	✓
SolFi V2	A	$y_{\text{raw}} \cdot (1 - \phi)$	✓	✓	✓
BisonFi	B	$\sum_i y_i$ (pivot tiers)	✓	–	✓
Tessera V	B	$\sum_i y_i$ (position tiers)	–	–	✓
GoonFi V2	B	$p \cdot \pi(p)$ (position tiers)	✓	✓	✓
ZeroFi	B	$\sum_i y_i$ (observation tiers)	✓	–	✓

Table 6.2: Architectural features of the seven reverse-engineered propAMMs. **Group A**: a single adjusted price is applied to the full input. **Group B**: input is consumed across discrete depth tiers. All protocols support direction asymmetry.

a *remaining capacity* $r = L - \text{inv}$, where L is an operator-set capacity parameter. All arithmetic is unsigned 64-bit integer; $\lfloor \cdot \rfloor$ denotes floor division, $\lceil \cdot \rceil$ ceiling division, and $\mathsf{isqrt}(x) = \lfloor \sqrt{x} \rfloor$ the integer square root. In the formulas below, Greek letters $(\alpha, \beta, \gamma, \ldots)$ denote operator-configurable parameters whose exact fixed-point definitions are given per protocol in Appendix A.

Shared structure. All seven protocols share a fundamental property that differentiates them from constant-function AMMs: the mid-price P is set by an off-chain operator (via oracle updates or direct parameter writes), not derived from an on-chain invariant. This substantially reduces the need for arbitrage trades to align the pool price with the external market, and with it the LVR cost identified by Milionis et al. [34] that is inherent to passive AMMs. Residual adverse selection may still arise from oracle latency or stale parameter updates. Moreover, most of them also have those core mechanisms:

(i) *Trade-Size Impact*: all protocols widen the effective spread for larger trades, using $\sqrt{x}$, x^2, or piecewise-linear functions of the input amount;

(ii) *Direction Asymmetry*: all protocols maintain separate parameters for buy and sell, allowing the spread to differ by direction;

(iii) *Staleness-Dependent Fees*: several protocols increase the fee when the oracle or pool state has not been updated recently, offering worse prices when the signal is stale;

(iv) *Inventory Sensitivity*: four of seven protocols adjust pricing based on the pool's reserve imbalance, analogous to a dealer skewing quotes to manage inventory risk;

(v) *Post-Trade Imbalance Awareness*: unlike passive AMMs where slippage is a mechanical consequence of the invariant curve, several propAMMs explicitly compute the anticipated post-trade reserve imbalance and price accordingly;

(vi) *Dynamic Parameter Updates*: all parameters can be updated between trades by the pool operator, enabling strategy adaptation to changing market conditions; and

(vii) *Safety Bounds*: several protocols cap their output via flat-formula ceilings or fee

maxima, preventing degenerate quotes under extreme parameters.

The protocols differ in how these mechanisms are composed. We group them into two families by how the output is assembled (Table 6.2).

Group A: Effective-price models. AlphaQ, Humidifi, and SolFi V2 compute the output by applying a single adjusted price to the input amount. AlphaQ and Humidifi include all adjustments (e.g. spread, trade-size impact, inventory) in an effective price P_{eff}, leading to $y = \lfloor x \cdot P_{\text{eff}}(x, d, s) \rfloor$. SolFi separates the computation into a raw output from the oracle price and a multiplicative fee: $y = y_{\text{raw}} \cdot (1 - \phi)$.

AlphaQ. AlphaQ adjusts a direction-specific midpoint multiplier $Q_d \in \{Q_q, Q_b\}$ for trade size and inventory:

$$Q_{\text{adj}} = Q_d \;\mp\; \alpha\sqrt{x} \;\mp\; \beta x^2 \;+\; s \cdot (\gamma\sqrt{|r|} + \delta r^2), \tag{6.1}$$

where $s = \text{sgn}(r)$ and the $\mp$ terms widen the spread with trade size, while the inventory terms incentivise trades that reduce imbalance. The adjusted multiplier is converted to an effective price $P_{\text{adj}} = Q_{\text{adj}} \cdot P/10^{12}$ (floored for B2Q, ceiled for Q2B). A flat safety price P_{flat}, derived from a fixed multiplier Q_{flat}, caps the output:

$$y = \min\big(y(P_{\text{adj}}),\; y(P_{\text{flat}})\big). \tag{6.2}$$

Humidifi. Humidifi forms a direction-dependent fixed-point adjustment accumulator

$$A = \underbrace{A_0}_{\text{base offset}} + \underbrace{S(x, \Delta, d)}_{\text{size term}} + \underbrace{H(x, \Delta, d)}_{\text{large-trade impact}} + \underbrace{I(v_a, v_q, x, P, d)}_{\text{rebalancing rebate}}, \tag{6.3}$$

where Δ is the pool's internal reserve-side deviation state, S is the signed size-dependent adjustment derived from the pre- and post-trade normalised states, H is an additional impact term that activates only above protocol thresholds, and I is applied only when the swap direction reduces reserve imbalance. The accumulator is then mapped to an effective denominator

$$D_{\text{eff}} = 2^{48} + \underbrace{q(A)}_{\text{fixed-point scaling}}, \tag{6.4}$$

and the final quote is obtained by dividing through D_{eff}:

$$y = \begin{cases} \left\lfloor \dfrac{P \cdot x}{D_{\text{eff}}} \right\rfloor, & d = \text{B2Q}, \\ \left\lfloor \dfrac{x \cdot 2^{48}}{\lfloor P \cdot D_{\text{eff}}/2^{48} \rfloor} \right\rfloor, & d = \text{Q2B}. \end{cases} \tag{6.5}$$

Thus, larger trades worsen the quote through S and H, while trades that move the pool toward balance may receive a rebate through I.

SolFi V2. SolFi corrects the oracle price for vault imbalance and applies a fee composed of three main components:

$$P_{\text{adj}} = \left\lfloor P \cdot \frac{10^7 + \text{corr}(v_a, v_q, P)}{10^7} \right\rfloor, \tag{6.6}$$

$$\phi = \delta_d + \left\lfloor \frac{w_{\text{stale}}(t) \cdot w_{\text{tier}}(y_{\text{raw}}) \cdot m}{10^6} \right\rfloor + \sigma_{\text{base}}, \tag{6.7}$$

where δ_d is a direction offset, $w_{\text{stale}}(t)$ widens with oracle age (protecting against informed flow when the price signal is stale), w_{tier} widens with trade size (penalising large trades that consume more depth), m is a base multiplier, and σ_{base} is a static spread floor. The output is $y = y_{\text{raw}} \cdot (10^7 - \phi)/10^7$.

Group B: Tier-consumption models. BisonFi, Tessera V, GoonFi V2, and ZeroFi consume the input across discrete depth tiers, each with a progressively worse price. The output is a summation $y = \sum_i y_i$ over tier contributions. This structure gives the operator direct control over the depth schedule: how much liquidity is available at each price level.

GoonFi V2. GoonFi is the most complex protocol in the dataset: it selects among several quoting paths based on two slot-age counters that measure pool-update staleness and trade staleness, respectively. In each direction, the mechanism either integrates an interpolated tier price $\pi_d(p)$ over a cumulative position interval, possibly after decaying the stored position by $p' = \max(0,\, p_{\text{old}} - v_{\text{dec}}\, t_a)$, or falls back to an input-weighted stale quote. The oracle price is corrected for vault imbalance via

$$P_{\text{adj}} = \left\lfloor P \cdot \frac{10^6 + c_{\text{vault}}}{10^6} \right\rfloor, \tag{6.8}$$

where c_{vault} is either a linear or piecewise correction that steepens beyond a vault-imbalance threshold. The effective spread may additionally include a staleness-scaled fee $\phi_{\text{stale}} = \phi_{\text{base}} \cdot \max(0,\ s_{\text{clk}} - s_{\text{oracle}} - \eta)$.

ZeroFi. ZeroFi uses a two-observation dynamic fee model with BPF-exact soft-float arithmetic:

$$\begin{aligned} \Delta\phi &= o_a - o_b, \\ y_i &= \text{bpf_f64_to_int}\big((1 \pm \phi_i \cdot 10^{-7}) \cdot O_d \cdot \min(x_{\text{rem}}, \text{bp}_i)\big), \\ y &= \textstyle\sum_i y_i, \end{aligned} \tag{6.9}$$

where o_a, o_b are cranker-updated observations, $\phi_i = \Delta\phi - h_i \cdot \psi_i(t)$ is the per-tier fee after staleness scaling, O_d is the oracle bid or ask rate (stored as raw f64 bits), and bp_i are tier breakpoints.

BisonFi. BisonFi uses a multi-tier pivot system. After deducting a fee ϕ (base + staleness + cross-directional penalty), the net input $x_{\text{net}} = x - \lceil \phi \cdot x / 10^5 \rceil$ is consumed across tiers with progressively wider prices:

$$\boxed{y = \sum_{i=0}^{n} \left\lfloor \frac{\min(x_{\text{rem}},\, w_{i,\text{base}}) \cdot P_i}{2^{48} \cdot S} \right\rfloor} \tag{6.10}$$

$$P_i = \left\lfloor \frac{P \cdot (10^4 + \text{adj}_i \cdot \alpha)}{10^4} \right\rfloor, \quad w_i = \left\lfloor \frac{v_q \cdot \text{mult}_i}{10^4} \right\rfloor, \tag{6.11}$$

where α is an adjustment constant, $\text{adj}_i \in \{-4, \ldots, +4\}$ is the signed adjustment index per tier, mult_i determines the tier width as a fraction of reserves, and S is the decimal scale. Widths w_i are computed in quote units and must be converted to base units ($w_{i,\text{base}} = \lfloor w_i \cdot S \cdot 2^{48} / P_i \rfloor$) before consumption; see Appendix A.6 for details. The fee ϕ includes a staleness penalty $\rho \cdot \max(0, s_{\text{clock}} - s_{\text{pool}})$ that widens the spread when the oracle is not fresh.

Tessera V. Tessera V tracks a cumulative position p_d that depletes tier widths, and applies a flat fee ϕ (in ppm) to the aggregate output across 10 tiers:

$$\boxed{y = \frac{10^6 - \phi}{10^6} \cdot \sum_{i=0}^{9} \left\lfloor \frac{\min(x_{\text{rem}},\, w_i') \cdot \pi_i}{10^{15}} \right\rfloor} \tag{6.12}$$

where $\pi_i = \lfloor f_i \cdot R_d / 10^6 \rfloor$ is the per-tier mid-price (computed in a first rounding step), w_i is the width of tier i, and R_d is the direction-specific fixed-point rate. The effective width w_i' is obtained by depleting the position p_d cumulatively across tiers: p_d first reduces w_0; any excess carries into w_1, then w_2, and so on. Position overflow from one direction is converted and added to the other direction's position via the exchange rate.

6.2.2 Validation

Section 6.2.1 presented the recovered formulas; this subsection instead checks the toolchain itself: whether each quoter reproduces its on-chain binary.

We validated each quoter against its live on-chain binary using the soak harness of Section 4.1.5. A run is 100 fetch-simulate-replay iterations; each iteration sweeps 100 linearly spaced input amounts, spanning 1%–50% of the pool's vault balances, in both directions against a fresh mainnet snapshot (20,000 comparisons per pool). We used an absolute

tolerance of ±1 token unit (the smallest representable amount) to account for integer rounding differences, but observed no deviations beyond this level across all protocols and pools. Across the seven protocols, 769 regression snapshots accumulated during development; on the final quoters the residual mismatch rate is zero.

Toolchain validation against synthetic ground truth. The soak suite establishes that our Python quoters reproduce each on-chain binary's output, but cannot rule out the possibility that the recovered formulas are correct only by coincidence on the live state distribution. As a complementary check, we wrote three minimal sBPF programs whose quoting formulas are known by construction (a constant-product market maker, a fixed-rate-with-fee linear quote, and a piecewise two-tier quote) and ran the same toolchain against them across input amounts spanning four orders of magnitude. In every case the recovered pseudocode was structurally equivalent to the source-level formula and verified against the executed trace. The synthetic programs and validation harness are included in the project repository.

6.3 Empirical Analysis

We apply a market-microstructure lens to propAMMs, replaying the recovered quoters against the on-chain state updates described in Section 6.1.2. A propAMM is funded by a single operator (Section 2.1.4), so its operator is its sole liquidity provider. We keep the term LP below for continuity with the microstructure literature, but it always denotes the operator's capital, and "LP profitability" is the propAMM profitability of **RQ2**.

6.3.1 Quoting Behaviour

Figure 6.1 shows a 15-minute snapshot of three protocols with distinct quoting styles. The bid-ask spread is computed at a representative size of $100. Humidifi (top) maintains a tight bid-ask band (∼1 bps), continuously processing swaps in both directions. GoonFi V2 (middle) shows a distinctive pattern of transient spread spikes: in slots where a swap occurs, its staleness-gated fee mechanism triggers a minimum fee floor and position-dependent tier pricing, widening the spread to ∼25 bps; in non-swap slots, the fee drops to near zero, producing a baseline of ∼4 bps. ZeroFi (bottom) maintains the widest band (∼13 bps) and adjusts it continuously based on inventory, with visible widening during periods of directional flow.

Spread, flow, and market share. Since all propAMMs substantially reduce LVR by using oracle-driven pricing, LP revenue is primarily derived from the bid-ask spread. Table 6.3 summarises the key quoting metrics; tighter spreads are generally associated with higher swap counts. Two exceptions stand out. BisonFi captured the highest volume share (38.8%) despite only 14.0% of swap count, consistent with its deeper tier struc-

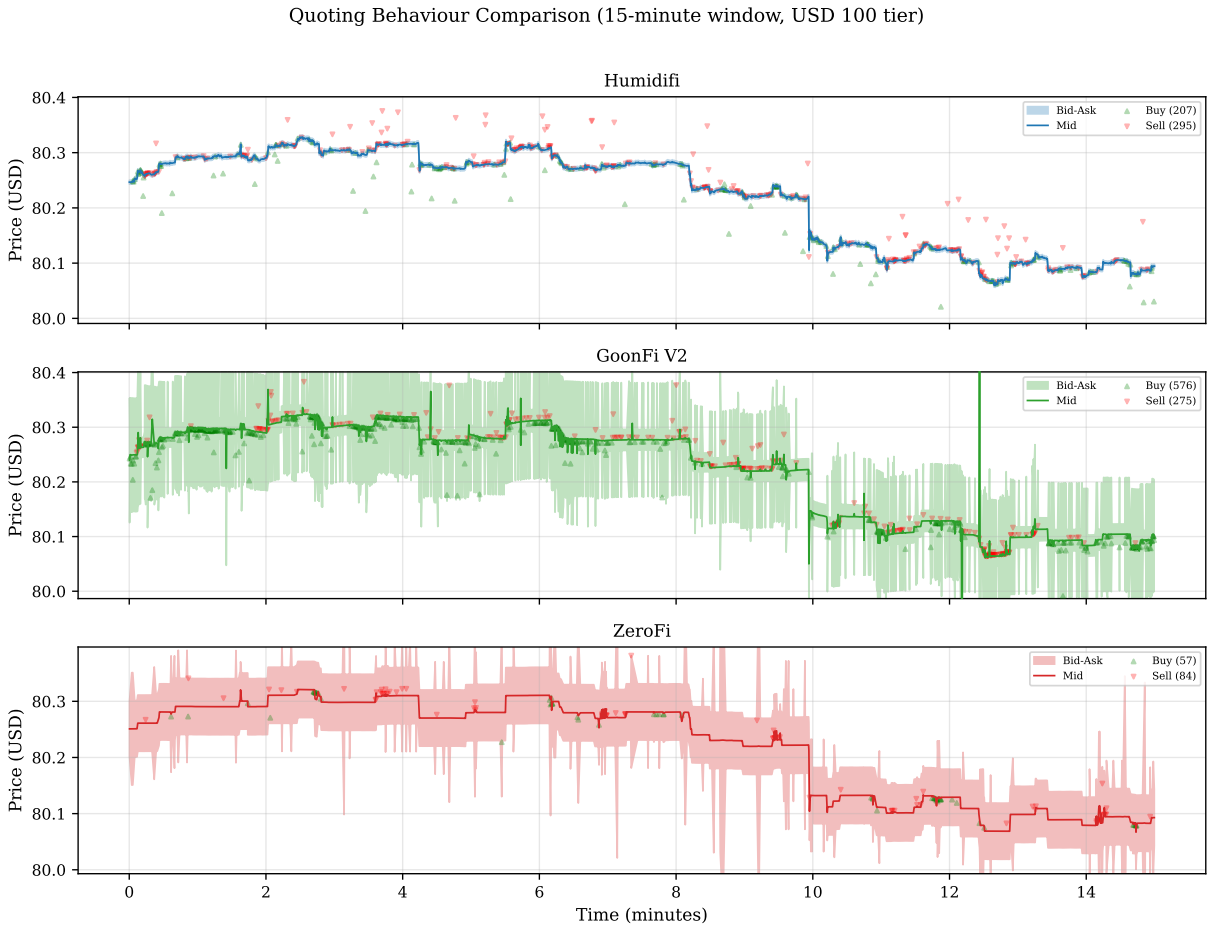


Figure 6.1: 15-minute snapshot comparing three quoting styles.

ture and higher Total Value Locked (TVL) ($9.4M) absorbing fewer but larger trades. GoonFi V2 is also a notable outlier: despite a time-weighted median spread of 4.83 bps, it captured 24.2% of all swaps. This is explained by its bimodal spread distribution: between swaps, its position-decay mechanism resets the cumulative position toward zero, producing a resting spread near its first-tier level (∼4 bps or below). The 4.83 bps median is dominated by non-swap observations and overstates the spread visible to the aggregator at routing time.

Depth schedules. Figure 6.3 shows how each protocol's spread grows with trade size on a log-log scale. All protocols exhibit convex slippage, but the shapes differ. Tessera V offers the tightest median spread across nearly all trade sizes (∼0.56 bps at $100, rising to ∼5 bps at $100K), followed by Humidifi (∼0.95 bps at $100, rising to ∼20 bps at $100K). BisonFi and Tessera V show step-like increases as trades consume successive depth tiers, while Humidifi's curve is smooth. GoonFi, SolFi, and ZeroFi cluster at wider baseline spreads (3–14 bps) with steeper convexity at large sizes.

Market segmentation by trade size. Quoting differences are associated with market segmentation by trade size (Figure 6.2). For trades below $1,000, Humidifi and GoonFi V2 together captured 75–80% of swap counts, consistent with their tight spreads at small sizes

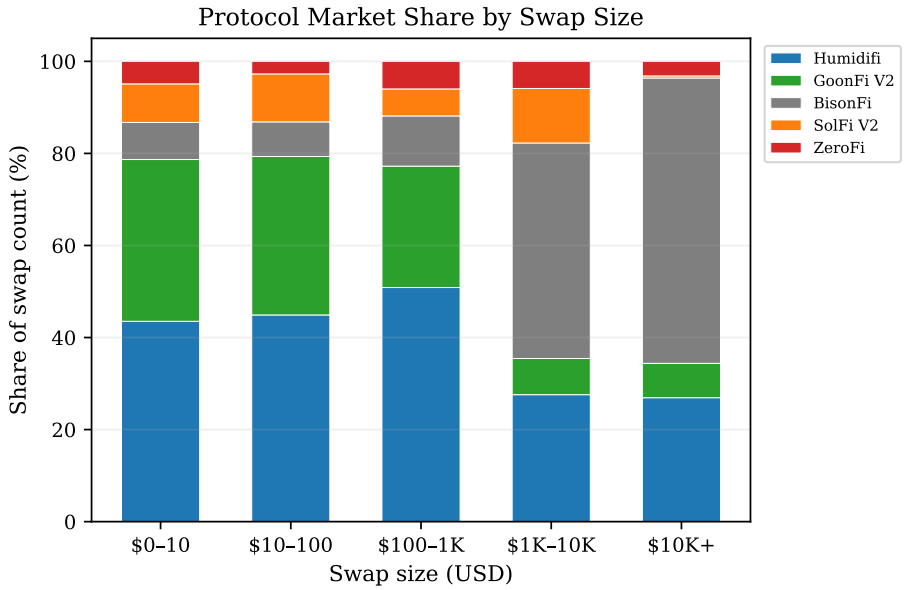


Figure 6.2: Protocol market share by swap size bin (by swap count). Humidifi and GoonFi dominate small trades; BisonFi captures the majority of trades above $1,000.

Protocol	Spread (bps)	Swaps	Swap share	Volume	Vol. share
AlphaQ	0.08	15,245	11.0%	$15.9M	17.6%
Humidifi	0.95	53,857	38.8%	$24.1M	26.7%
BisonFi	1.86	19,426	14.0%	$34.9M	38.8%
SolFi V2	4.59	10,676	7.7%	$4.2M	4.7%
GoonFi V2	4.83	33,644	24.2%	$7.1M	7.9%
ZeroFi	12.46	6,001	4.3%	$3.8M	4.3%
Total		**138,849**		**$90.1M**	

Table 6.3: Quoting summary over the 10-hour window, sorted by median spread at the $100 tier. Tessera V is omitted (zero swaps). Tighter spreads correlate with higher swap share but not necessarily volume share.

receiving aggregator routing priority. Above $1,000, BisonFi's share increases from ∼5% to over 60%.

6.3.2 Risk Management and Inventory Dynamics

A market maker must decide how to respond when its inventory becomes unbalanced. The theoretical literature predicts that an optimal dealer should widen spreads as inventory risk grows [7]. Our recording reveals qualitatively different inventory dynamics across protocols, visible in Figure 6.4.

Passive balance through tight spreads. Humidifi maintained a near-constant ∼0.9 bps spread and a relatively stable inventory balance. Despite processing 53,857 swaps ($24.1M in volume), its imbalance $q(t)$ fluctuated in a narrow band. This is consistent with tight spreads attracting balanced aggregator flow, though active operator intervention may also contribute.

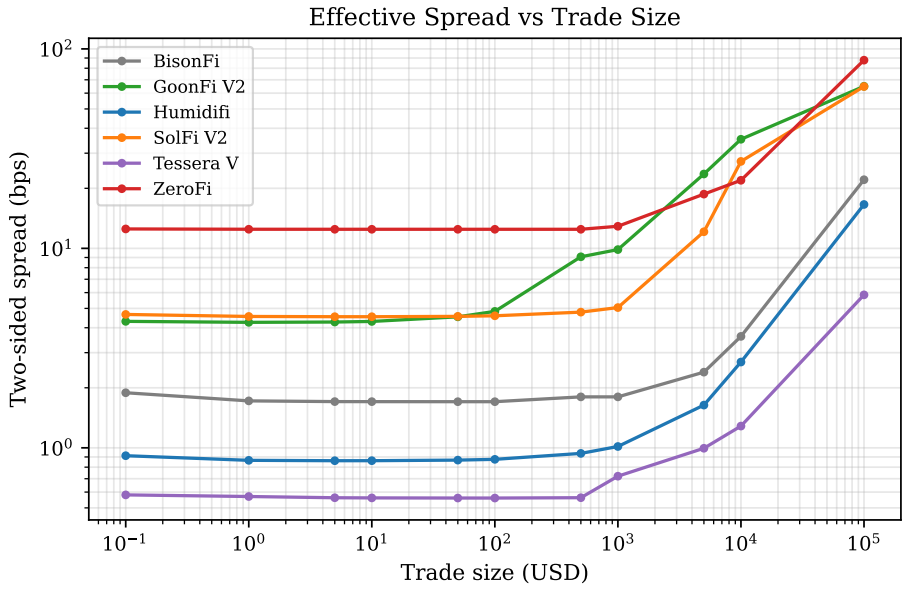


Figure 6.3: Median effective spread vs. trade size (primary SOL/USDC pool per protocol, log-log), computed by replaying the recovered quoter at each recorded state update across the full 10-hour window. Convex slippage is universal, but functional forms differ: smooth curves (Humidifi) vs. step-like tiers (BisonFi, Tessera V).

Formula-level inventory sensitivity. SolFi V2's spread widened from ∼4.6 bps (baseline) to ∼13 bps during periods of directional flow. This behaviour arises from two mechanisms in the recovered formula: a vault-imbalance correction that adjusts the effective price when reserves deviate from the target ratio, and a staleness offset that activates when the oracle has not been recently refreshed. Consecutive same-direction swaps push the vault out of balance, which widens the spread against further same-direction trades.

Off-chain parameter management. ZeroFi's on-chain formula does not reference vault balances in the fee calculation; the spread is determined entirely by two observation parameters (o_a, o_b) updated by the operator's off-chain cranker (see Appendix A.8). Nevertheless, the recorded data show a positive rank correlation between the effective spread and absolute inventory imbalance. One interpretation is that the operator's cranker adjusts o_a and o_b in response to vault imbalance, though the correlation could also reflect a driver like directional price movement. This illustrates a broader point about pAMM architecture: the on-chain formula defines the quoting *mechanism*, but the operator's off-chain strategy determines how that mechanism is *parameterised* over time.

Mean reversion speed. We quantify inventory mean reversion by fitting an AR(1) model to the imbalance time series: $q_{t+1} = \alpha + \beta q_t + \varepsilon$. Since state updates arrive at irregular intervals (event-driven), we resample the imbalance series to a uniform 1-second frequency using last-observation-carried-forward interpolation before fitting. The half-life $\tau_{1/2} = -\ln 2 / \ln \beta$ measures how quickly each pool reverted to its target imbalance during the observation window (Figure 6.5). Half-lives ranged from 29 s (Humidifi's fastest pool)

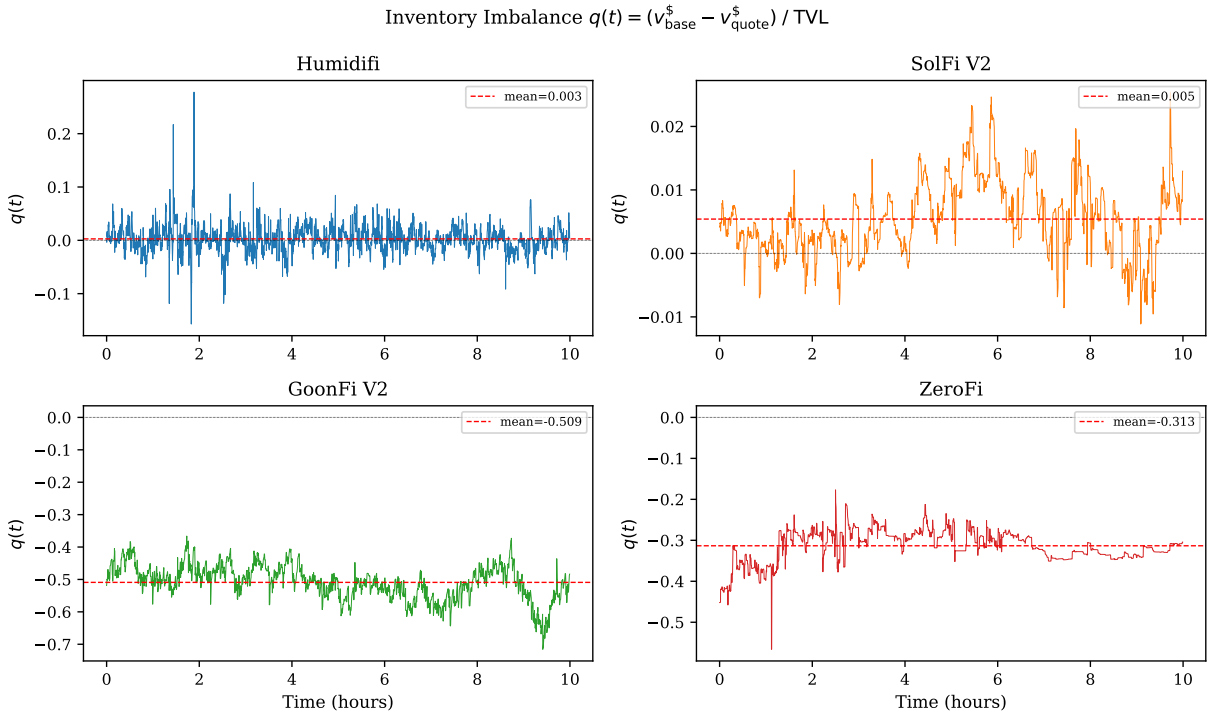


Figure 6.4: Inventory imbalance $q(t) = (v^{\$}_{\text{base}} - v^{\$}_{\text{quote}})/\text{TVL}$ for four representative pools. Humidifi maintains tight control ($\sigma = 2.4\%$); GoonFi V2 and ZeroFi show wider swings.

to 5,774 s (BisonFi), with GoonFi V2 (390 s), ZeroFi (456–1,186 s), and SolFi V2 (877 s) in between.

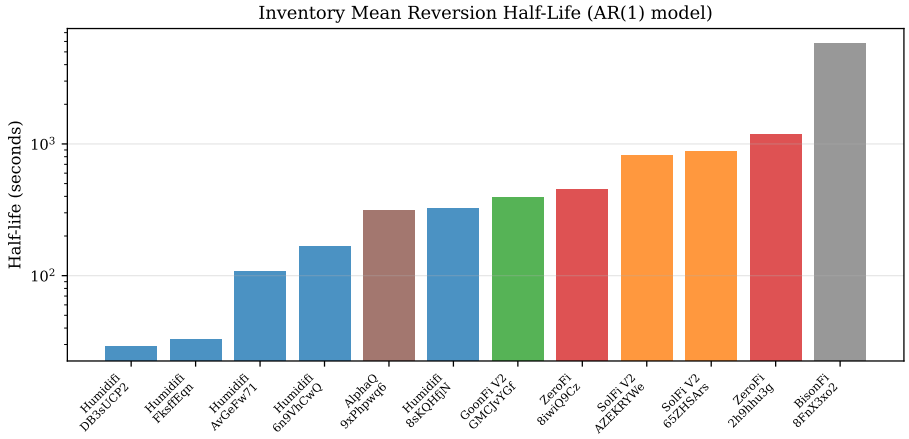


Figure 6.5: Inventory mean reversion half-life (AR(1) model, log scale) during the observation window. Faster reversion co-occurred with tighter spreads and higher swap frequency.

6.3.3 Order Flow Composition

LP markout analysis. We measure adverse selection using the *markout* approach standard in the market microstructure literature [31]. For each swap i executed at time t_i

with execution price P_i^{exec}, we compute the LP's per-trade profit at horizon τ:

$$\text{markout}_i(\tau) = d_i \cdot \frac{M_{t_i+\tau} - P_i^{\text{exec}}}{M_{t_i}} \times 10{,}000 \quad [\text{bps}], \tag{6.13}$$

where $d_i = +1$ if the LP sold base (B2Q swap) and $d_i = -1$ if the LP bought base (Q2B swap), and M_t is a *leave-one-out* consensus mid-price: the median mid across all active pools *excluding all pools of the protocol being evaluated*. Positive markout indicates LP profit; negative indicates adverse selection. We compute markout at horizons $\tau \in \{1, 2, 5, 10, 30, 60, 120, 300, 600\}$ s to avoid dependence on a single threshold.

Figure 6.6 shows the markout curves. The *equal-weighted* panel (left) treats every swap equally regardless of size and reveals a clear hierarchy. Humidifi's LP earned ∼1.7 bps per trade (95% CI at 5 s: [1.50, 1.86] bps), and this value was remarkably stable across all horizons from 1 s to 600 s. SolFi V2 (∼0.7 bps) and ZeroFi (∼0.5 bps) also showed positive per-trade markout. BisonFi's markout was the lowest (∼0.1 bps), consistent with its larger average trade size leaving less margin per trade after depth consumption.

The *volume-weighted* panel (right) weights each swap by its dollar size, giving large trades proportionally more influence. Here, only Humidifi remained solidly positive (∼0.35 bps). BisonFi, SolFi, and ZeroFi turned negative at several horizons. Thus typical (small) swaps were LP-profitable while large swaps lost money to adverse selection. GoonFi V2 hovered near zero volume-weighted, suggesting its flow was approximately balanced in dollar terms.

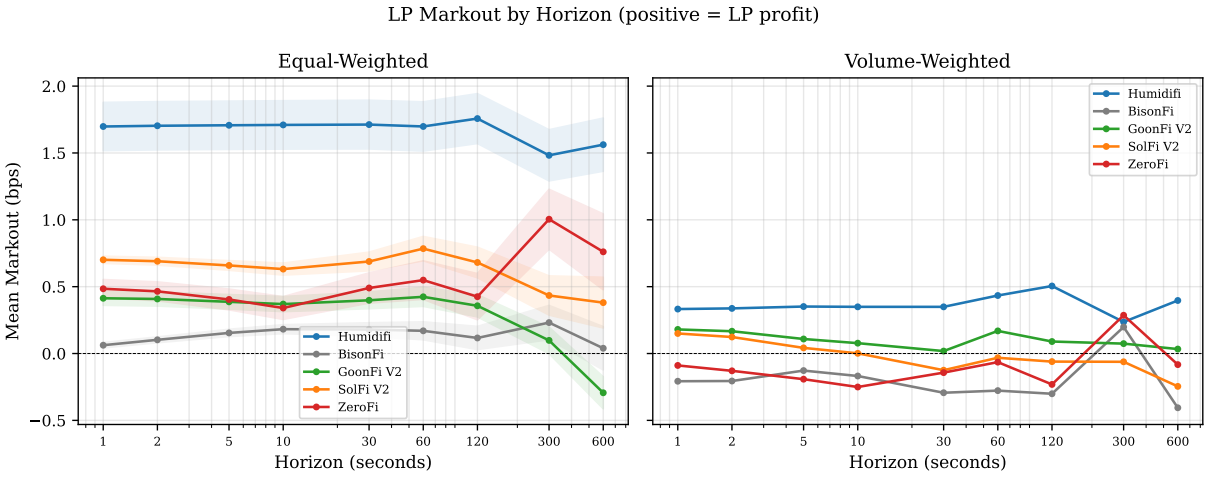


Figure 6.6: LP markout by horizon (positive = LP profit). Left: equal-weighted (per-trade). Right: volume-weighted. Humidifi's markout is stable and positive at all horizons under both weightings; other protocols show negative volume-weighted markout, indicating that large trades carry adverse selection. Shaded bands show 95% bootstrap CIs.

Size-dependent adverse selection. Figure 6.7 decomposes the markout by trade size quintile at the 5-second and 60-second horizons. All protocols show a monotonic decline from Q1 (smallest trades, highest markout) to Q5 (largest trades, near-zero or negative

markout). Humidifi's Q1 markout (∼3.4 bps at 5 s) was nearly ten times its Q5 markout (∼0.4 bps). BisonFi's Q4 and Q5 markout turned negative, consistent with its overall PnL volatility: individual large trades can generate substantial adverse selection.

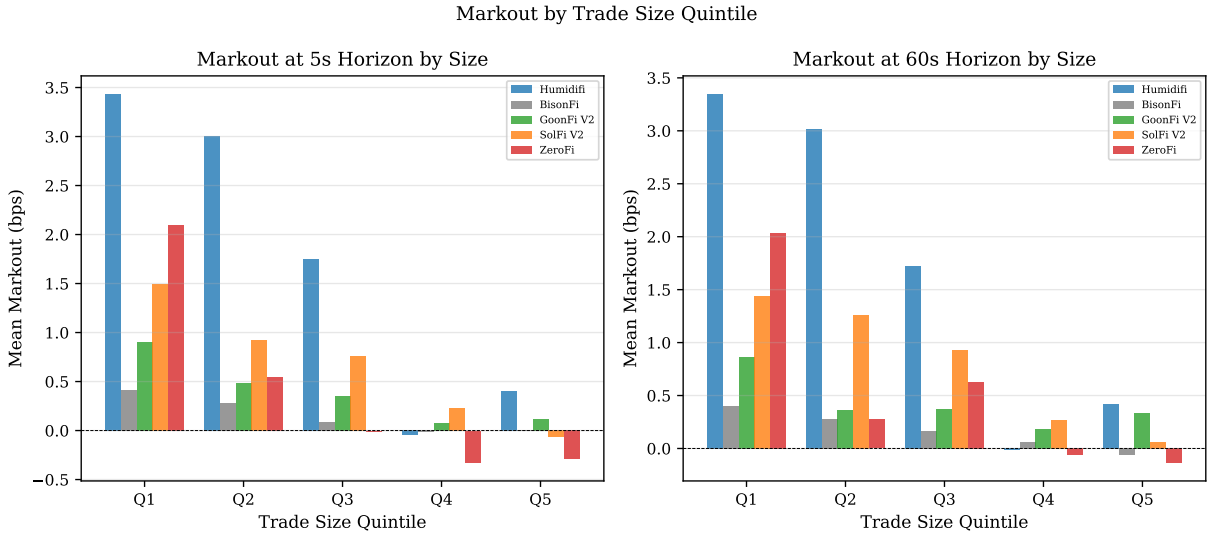


Figure 6.7: Markout by trade size quintile at 5-second and 60-second horizons. All protocols show monotonically declining markout with trade size. Small trades (Q1) are uniformly LP-profitable; large trades (Q4–Q5) approach zero or turn negative.

These results are consistent with tighter-spread protocols attracting more small, LP-profitable trades, though a single 10-hour snapshot cannot establish causality.

Realised spread decomposition. The markout pattern is corroborated by the realised spread analysis (Figure 6.8). We decompose each swap's execution into three components: the quoted half-spread (execution price vs. consensus mid at swap time), the adverse selection component (consensus mid movement against the LP within 5 seconds), and the realised spread (quoted minus adverse). BisonFi showed the smallest adverse selection at the 5-second horizon (0.20 bps) but its quoted half-spread was also small (−0.53 bps). GoonFi and ZeroFi showed negative adverse selection (i.e., the consensus mid moved *in favour of* the LP after the swap), consistent with their parameter updates repricing the pool immediately after execution.

6.3.4 LP Profitability

We test whether favourable flow composition translates to higher aggregate PnL using two complementary methodologies, both segmented around capital events to isolate trading-driven returns.

Reference price. Both metrics use a *consensus propAMM oracle* as the reference price P_{ref}: at each state transition in any pool, we compute the median mid-price across all active pools (requiring ≥ 3 contributors). This consensus price, built from 833,849 ob-

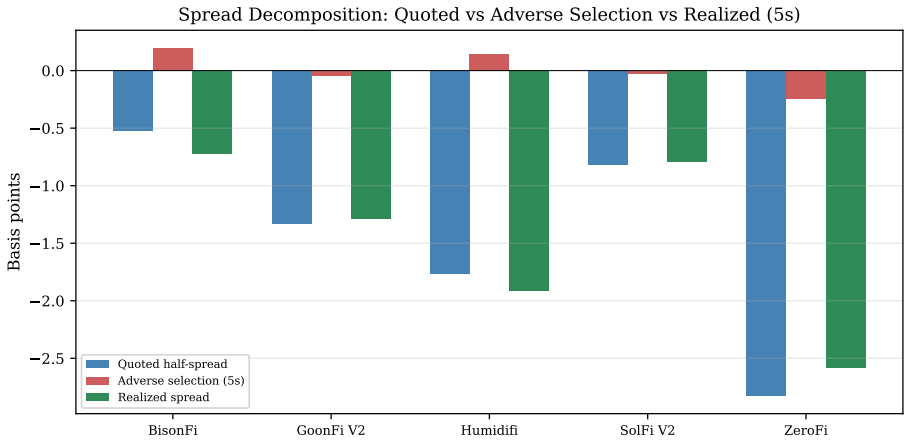


Figure 6.8: Spread decomposition at the 5-second horizon: quoted half-spread, adverse selection, and realised spread by protocol. Positive adverse selection values indicate the consensus mid moved against the LP; negative values (GoonFi, ZeroFi) are consistent with post-swap repricing.

servations across 11 pools, provides a reference derived directly from the protocols under study, avoiding the timing discrepancies introduced by external oracles.

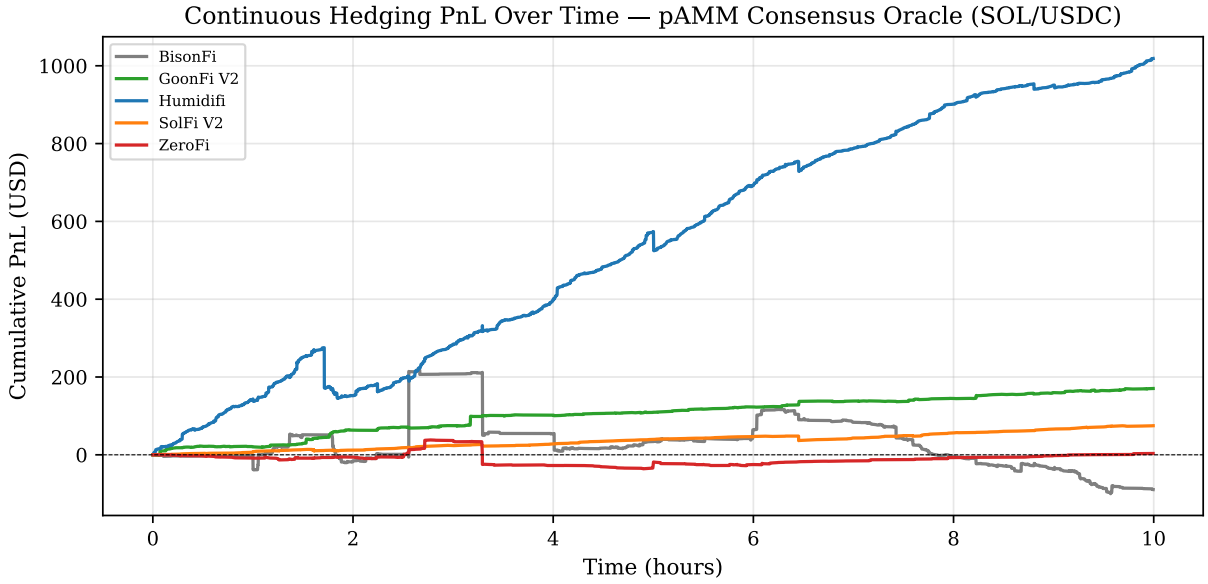


Figure 6.9: Continuous-hedging LP PnL over the 10-hour window by protocol (aggregated across pools).

Continuous-hedging PnL. Our primary metric follows the LVR framework of Milionis et al. [34]: for each state transition i, we mark the LP's inventory change to the consensus reference price:

$$\Pi_{\text{CH}} = \sum_i \left[\frac{\Delta v_{q,i}}{10^6} + \frac{\Delta v_{b,i}}{10^9} \cdot P_{\text{ref}}(t_i) \right], \tag{6.14}$$

where $\Delta v_{b,i}$ and $\Delta v_{q,i}$ are the base and quote vault balance changes at transition i. This measures the LP's cumulative trading revenue net of adverse selection: each inventory change is valued at the prevailing reference price, analogous to a market maker who

continuously hedges their delta. Positive Π_{CH} indicates that the LP earned more from spread revenue than it lost to adverse selection [31].

Static hedge PnL. As a robustness check, we also compute the LP's return relative to simply holding the initial portfolio [25]: $\Pi_{\text{SH}}(t) = [v_b(t) - v_b(0)] \cdot P_{\text{ref}}(t)/10^9 + [v_q(t) - v_q(0)]/10^6$. This path-independent metric depends only on endpoint states but conflates trading returns with mark-to-market gains.

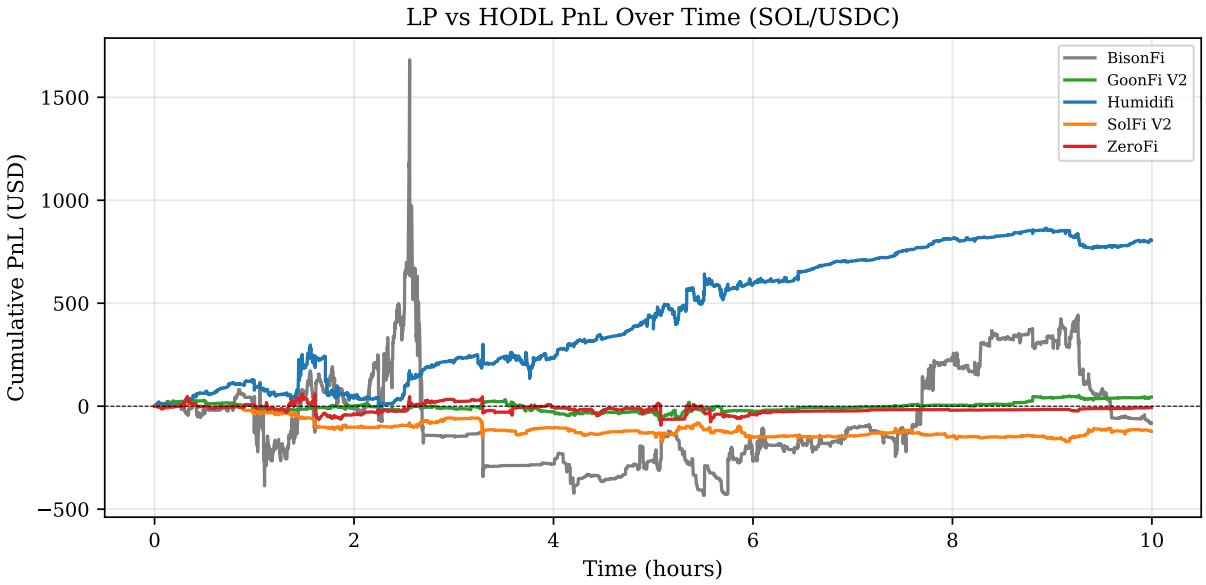


Figure 6.10: Static hedge LP PnL over the 10-hour window by protocol. The ranking is broadly consistent with the continuous-hedging metric (Figure 6.9), though absolute values differ substantially for BisonFi and GoonFi.

Figures 6.9 and 6.10 show the cumulative PnL under both metrics. Humidifi was the most profitable protocol during the recording window: its five pools collectively earned +$1,050 (continuous hedging) to +$805 (static hedge) over 10 hours. The largest pool (8sKQH, $449K TVL, 18,000 swaps) earned +$204 (continuous hedging), while the smaller DB3sU pool ($152K TVL) earned +$333 (static hedge). BisonFi's single active pool showed volatile PnL with large step-changes driven by individual large trades, ending at −$772 (continuous hedging) to −$82 (static hedge). GoonFi V2 earned +$331 (continuous hedging) to +$45 (static hedge). SolFi V2 and ZeroFi were near breakeven under both metrics.

6.3.5 Routing Efficiency

A natural question is whether the aggregator routes each swap to the *best available* pAMM. To test this, we replay all pool states simultaneously in a single interleaved timeline and, for each of the 115,274 detected swaps, query every other SOL/USDC pAMM with the *exact* input amount using the full on-chain quoting logic at the pre-swap state. This yields a precise counterfactual: for every executed swap, we know the output each alternative venue would have quoted at that snapshot.

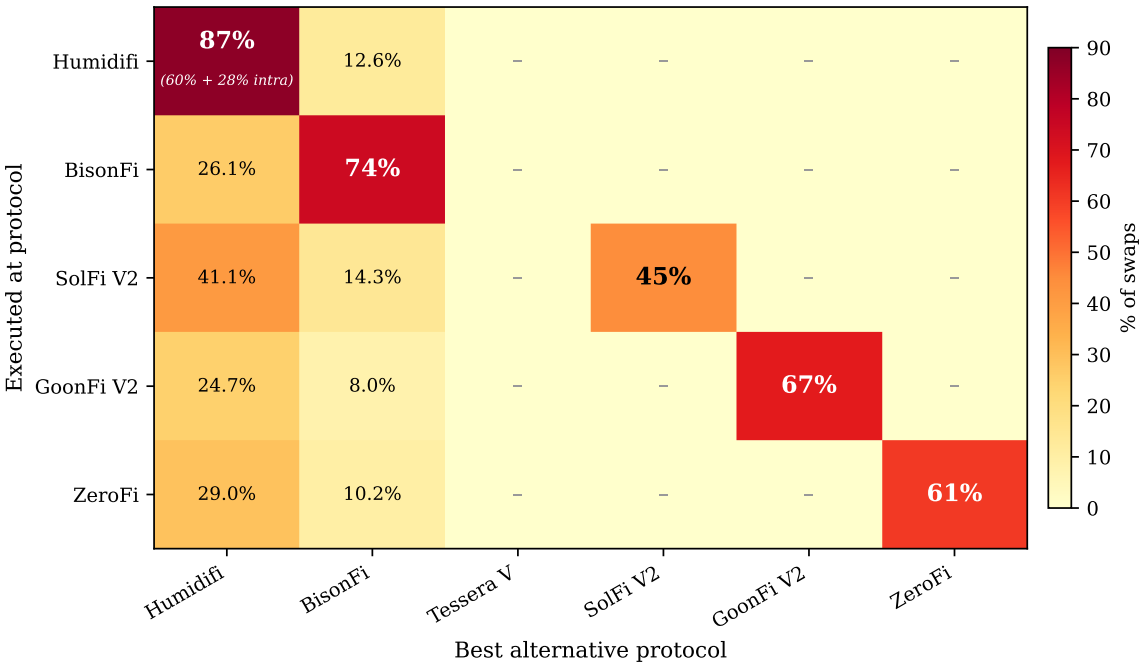


Figure 6.11: Routing efficiency matrix across SOL/USDC propAMMs (n = 115,274 swaps). Each cell shows the percentage of swaps executed at the row protocol for which the column protocol offered the best quote.

Figure 6.11 shows the resulting routing matrix. Each row represents the protocol where the swap was executed; each column shows the fraction of those swaps for which a given protocol would have offered the highest output.

The matrix reveals a clear routing hierarchy during the observation window. For Humidifi, 60% of swaps were already at the globally optimal pool; an additional 28% would have been better served by a different Humidifi pool, yielding a combined protocol-level retention of 87%. Same-pool retention for the single-pool protocols ranged from 74% (BisonFi) down to 45% (SolFi V2). In every instance in our dataset, when a better cross-protocol route existed, it was Humidifi or BisonFi, the two tightest-spread protocols.

When a better route does exist, the median improvement is modest: 3.25 bps for Humidifi swaps, 1.32 bps for SolFi V2, 0.83 bps for ZeroFi, 0.71 bps for GoonFi, and just 0.30 bps for BisonFi. The improvement is also size-dependent: 58% of sub-$10 swaps could have been improved (median 2 bps), while only 20% of $1K–$10K swaps have a better alternative (median <0.5 bps). This suggests the aggregator optimises more carefully for larger trades where the absolute saving justifies the computational cost.

6.3.6 Alternative Explanations

Several alternative explanations may contribute to these patterns independently of the spread mechanism.

TVL as a capacity constraint. Total value locked varied by a factor of 16× across protocols (GoonFi V2: $593K; BisonFi: $9.4M). BisonFi captured 38.8% of *volume* despite wider spreads (1.86 bps), consistent with its deeper reserves absorbing trades that smaller pools cannot fill. Conversely, GoonFi V2 captured 24.2% of swaps but only 7.9% of volume, suggesting that low TVL may constrain the trade sizes a protocol can absorb regardless of spread competitiveness.

Operator update frequency. All protocols were dominated by non-swap state updates (59–91% of all transactions were operator-initiated parameter or oracle writes). However, update intensity varied dramatically. Humidifi's operator issued 31,800 non-swap updates per hour (median inter-update gap: 0.01 s), with 37.5% of updates co-located in the same slot as a swap, suggesting reactive or pre-bundled quoting. By contrast, ZeroFi averaged 4,500 non-swap updates per hour with negligible co-location. This raises an alternative explanation: Humidifi's profitability may partially reflect a more sophisticated operator who simultaneously sets tighter spreads, updates parameters more frequently, and responds faster to incoming flow. The high co-location rate is reminiscent of the just-in-time liquidity provision studied by Capponi, Jia, and Zhu [11], who show that JIT providers can selectively provide liquidity to uninformed flow. PropAMMs with reactive quoting may achieve a similar effect at the protocol level. Our cross-sectional data cannot separate the spread mechanism from the algorithmic design of the off-chain program used to update parameters.

Summary. Oracle-driven pricing substantially reduces LVR relative to passive AMMs. Multi-horizon markout analysis shows that tighter-spread protocols attracted more LP-profitable flow, and aggregate PnL rankings were consistent with the per-trade markout hierarchy. The routing analysis found that only the two tightest-spread protocols appeared as best alternatives for swaps executed elsewhere. However, these observations are drawn from a single 10-hour window under low-to-moderate volatility, and alternative explanations (TVL constraints, operator sophistication) may contribute to the patterns independently of the spread mechanism.

Chapter 7

Discussion and Conclusion

7.1 Interpretation of Findings

The reverse-engineering and empirical evaluation reveal that propAMMs occupy a middle ground between passive AMMs and traditional market makers. Passive AMMs commit to a fixed, transparent pricing function that cannot be adjusted at will; traditional dealers, by contrast, have full discretion over their quotes. PropAMMs combine elements of both: the on-chain formula defines the quoting *mechanism* (how inputs are combined to produce a quote), while the off-chain operator determines the *parameterisation* (which values are fed into that mechanism). SolFi V2 embeds inventory sensitivity directly in its formula through a vault-imbalance correction, reducing dependence on operator intervention. ZeroFi's formula contains no inventory logic; the observed correlation between spread and imbalance reflects the operator's cranker strategy rather than a protocol-level feature. Humidifi maintains stable inventory through tight spreads that attract approximately balanced aggregator flow.

Despite being developed independently, all seven protocols converged on the same core design features: (i) oracle-based pricing, (ii) convex slippage, (iii) direction asymmetry, and (iv) staleness-dependent fees. This is consistent with selection pressure from Solana's aggregator-mediated routing layer, which routes volume to the highest-quoting protocol for each trade and thereby rewards the same design choices across independent teams. The most consistent empirical pattern is the association between spread competitiveness, flow composition, and LP profitability: tighter-spread protocols attracted more small, LP-profitable trades, and their aggregate PnL rankings followed this per-trade advantage.

7.2 Limitations

Recording window and market conditions. All empirical observations are drawn from a single 10-hour recording on the SOL/USDC pair under low-to-moderate volatility

(SOL/USD $78.50–$80.50). Results may differ under trending markets, high-volatility conditions, or different trading pairs.

Sample and generalisability. The study covers seven protocols on Solana only. Solana-specific features (sub-second slots, no public mempool, aggregator dominance) shape the competitive dynamics we observe; findings may not transfer to Ethereum or L2 environments where block times, MEV dynamics, and market structure differ. The recovered formulas are point-in-time snapshots of upgradeable programs. We monitored each program's binary hash throughout the recording; no upgrades occurred, so the recovered formulas held over the analysed period, though binaries may have changed since.

Methodological constraints. No wallet-level data was available, so we cannot identify individual traders or distinguish retail from institutional flow. Operator behaviour is also unobserved: we see parameters change but not the decision process behind them.

Reverse-engineering methodology. The backward dataflow analysis operates on concrete execution traces, not symbolic analysis, and therefore recovers the computation for specific inputs rather than proving properties over all inputs. The control dependency resolution adapts DTA++ to a backward setting; the register-only comparison at the post-dominator may miss value-influencing branches that operate through memory, though this is unlikely for the register-heavy sBPF calling convention.

Ethics and legal position. The dissertation operates only on artefacts that are public by design: propAMM bytecode is replicated to every Solana validator, account state is read through standard RPC, and no access controls or authentication are involved. All execution is local, no live state is modified, and the released artefacts are the analysis tooling and an independent re-implementation, not the operators' binaries or any non-public information. We view the resulting transparency as net beneficial: propAMMs now intermediate the majority of Solana DEX volume, and making their behaviour empirically tractable extends the reach of academic and user scrutiny that open-source venues already face.

7.3 Implications and Future Work

Opacity and transparency. PropAMMs create an opacity challenge: users cannot inspect the quoting logic that determines their execution price, and the aggregator-mediated competitive structure creates winner-take-most dynamics that concentrate volume in tighter-spread protocols. The reverse-engineering methodology shows this opacity is not insurmountable. By recovering closed-form formulas from closed-source binaries, the approach provides a basis for third-party transparency tools that could audit propAMM behaviour without requiring protocol cooperation.

Protocol design. The recovered formulas suggest that convex slippage and direction asymmetry are characteristic features of competitive on-chain market making, while the choice between embedding inventory logic in the formula versus delegating it to the operator represents a design trade-off between robustness and flexibility. Staleness-dependent fees, used by four of seven protocols, provide a practical defence against adverse selection.

Future work. *Is the observed convergence stable?* Longer recording periods across multiple volatility regimes would test whether the design patterns and profitability rankings we observe persist or are regime-dependent. *Is it general?* Multi-pair analysis (stablecoins, long-tail assets) would reveal whether propAMM strategies vary by asset type or whether the same aggregator selection pressure produces the same design features regardless of pair. *Why does it happen?* The empirical pattern is a self-reinforcing cycle: tighter spreads win more aggregator routing, the resulting retail flow yields positive LP PnL, and that PnL sustains the tight spreads. Whether this cycle is a stable equilibrium, whether it has a unique winner, and what could let a new entrant break it are open theoretical questions. A game-theoretic model of how heterogeneous market makers compete for aggregator-routed flow could address them.

7.4 Summary of Contributions

This dissertation made two contributions. First, we developed a semantic execution tracing methodology for Solana's sBPF virtual machine that recovers closed-form quoting formulas from closed-source programs through instrumented dataflow capture, backward graph reconstruction, and control dependency resolution. Second, we applied this methodology in the first empirical study of proprietary AMM behaviour, recovering the quoting logic of seven protocols and analysing their strategies, flow composition, LP profitability, and routing efficiency. The results connect two previously disjoint literatures, blockchain binary analysis and financial market microstructure, and provide the first empirical characterisation of a class of protocols that now accounts for a substantial share of decentralised exchange activity on Solana. More broadly, the methodology enables market microstructure questions to be asked of any opaque on-chain protocol with publicly deployed bytecode, extending empirical DeFi research beyond the open-source programs to which it has been confined.

References

[1] Hayden Adams, Noah Zinsmeister, and Dan Robinson. *Uniswap v2 Core*. Whitepaper. Uniswap, Mar. 2020. URL: https://app.uniswap.org/whitepaper.pdf.

[2] Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, and Dan Robinson. *Uniswap v3 Core*. Tech. rep. Uniswap Labs, Mar. 2021. URL: https://app.uniswap.org/whitepaper-v3.pdf.

[3] Hiralal Agrawal and Joseph R. Horgan. "Dynamic Program Slicing". In: *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation - PLDI '90*. White Plains, New York, United States: ACM Press, 1990, pp. 246–256. DOI: 10.1145/93542.93576. URL: http://portal.acm.org/citation.cfm?doid=93542.93576 (Accessed Apr. 6, 2026).

[4] Guillermo Angeris and Tarun Chitra. "Improved Price Oracles: Constant Function Market Makers". In: *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*. New York NY USA: ACM, Oct. 2020, pp. 80–91. DOI: 10.1145/3419614.3423251. URL: https://dl.acm.org/doi/10.1145/3419614.3423251 (Accessed Feb. 23, 2026).

[5] *Anza-Xyz/Agave*. anza-xyz. Apr. 2026. URL: https://github.com/anza-xyz/agave (Accessed Apr. 4, 2026).

[6] *Anza.Xyz Solana sBPF*. anza-xyz. Feb. 2026. URL: https://github.com/anza-xyz/sbpf (Accessed Feb. 18, 2026).

[7] Marco Avellaneda and Sasha Stoikov. "High-Frequency Trading in a Limit Order Book". In: *Quantitative Finance* 8.3 (Apr. 2008), pp. 217–224. DOI: 10.1080/14697680701381228. URL: https://doi.org/10.1080/14697680701381228 (Accessed May 20, 2026).

[8] Andrea Barbon and Angelo Ranaldo. "On the Quality of Cryptocurrency Markets: Centralized vs. Decentralized Exchanges". In: *Management Science* (Mar. 2026), mnsc.2024.07703. DOI: 10.1287/mnsc.2024.07703. URL: https://pubsonline.informs.org/doi/10.1287/mnsc.2024.07703 (Accessed Apr. 6, 2026).

[9] Philippe Bergault, Louis Bertucci, David Bouba, and Olivier Guéant. "Automated Market Makers: Mean-Variance Analysis of LPs Payoffs and Design of Pricing Functions". In: *Digital Finance* 6.2 (June 2024), pp. 225–247. DOI: 10.1007/s42521-023-00101-0. URL: https://link.springer.com/10.1007/s42521-023-00101-0 (Accessed Apr. 9, 2026).

[10] Agostino Capponi and Ruizhe Jia. *The Adoption of Blockchain-based Decentralized Exchanges*. July 2021. DOI: 10.48550/arXiv.2103.08842. arXiv: 2103.08842 [q-fin]. URL: http://arxiv.org/abs/2103.08842 (Accessed Feb. 23, 2026).

[11] Agostino Capponi, Ruizhe Jia, and Brian Zhu. *The Paradox Of Just-in-Time Liquidity in Decentralized Exchanges: More Providers Can Sometimes Mean Less Liquidity*. Feb. 2024. DOI: 10.48550/arXiv.2311.18164. arXiv: 2311.18164 [q-fin]. URL: http://arxiv.org/abs/2311.18164 (Accessed Apr. 6, 2026).

[12] Álvaro Cartea, Fayçal Drissi, and Marcello Monga. "Decentralized Finance and Automated Market Making: Predictable Loss and Optimal Liquidity Provision". In: *SIAM Journal on Financial Mathematics* 15.3 (Sept. 2024), pp. 931–959. DOI: 10.1137/23M1602103. URL: https://epubs.siam.org/doi/10.1137/23M1602103 (Accessed Apr. 6, 2026).

[13] Tarun Chitra, Kshitij Kulkarni, Mallesh Pai, and Theo Diamandis. *An Analysis of Intent-Based Markets*. Mar. 2024. DOI: 10.48550/arXiv.2403.02525. arXiv: 2403.02525 [cs]. URL: http://arxiv.org/abs/2403.02525 (Accessed Apr. 6, 2026).

[14] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. *A Simple, Fast Dominance Algorithm*. Technical Report TR-06-33870. Rice University, 2001. URL: https://repository.rice.edu/items/99a574c3-90fe-4a00-adf9-ce73a21df2ed (Accessed May 20, 2026).

[15] Siwei Cui, Gang Zhao, Yifei Gao, Tien Tavu, and Jeff Huang. "VRust: Automated Vulnerability Detection for Solana Smart Contracts". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. Los Angeles CA USA: ACM, Nov. 2022, pp. 639–652. DOI: 10.1145/3548606.3560552. URL: https://dl.acm.org/doi/10.1145/3548606.3560552 (Accessed Apr. 6, 2026).

[16] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. "Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability". In: *2020 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2020, pp. 910–927. DOI: 10.1109/SP40000.2020.00040. URL: https://ieeexplore.ieee.org/document/9152675/ (Accessed Apr. 6, 2026).

[17] DefiLlama. *DefiLlama — DEX Aggregator Volume*. 2024. URL: https://defillama.com/dexs/chains/solana (Accessed Apr. 7, 2026).

[18] *Deploying Programs*. URL: https://solana.com/docs/programs/deploying (Accessed Apr. 8, 2026).

[19] Michael Egorov. *StableSwap — Efficient Mechanism for Stablecoin Liquidity*. Curve Finance whitepaper. Nov. 2019. URL: https://berkeley-defi.github.io/assets/material/StableSwap.pdf (Accessed May 20, 2026).

[20] Robin Fritsch and Andrea Canidio. "Measuring Arbitrage Losses and Profitability of AMM Liquidity". In: *Companion Proceedings of the ACM Web Conference 2024*.

Singapore Singapore: ACM, May 2024, pp. 1761–1767. DOI: 10.1145/3589335.3651961. URL: https://dl.acm.org/doi/10.1145/3589335.3651961 (Accessed Apr. 6, 2026).

[21] Lawrence R. Glosten and Paul R. Milgrom. "Bid, Ask and Transaction Prices in a Specialist Market with Heterogeneously Informed Traders". In: *Journal of Financial Economics* 14.1 (Mar. 1985), pp. 71–100. DOI: 10.1016/0304-405X(85)90044-3. URL: https://linkinghub.elsevier.com/retrieve/pii/0304405X85900443 (Accessed Apr. 6, 2026).

[22] Neville Grech, Sifis Lagouvardos, Ilias Tsatiris, and Yannis Smaragdakis. "Elipmoc: Advanced Decompilation of Ethereum Smart Contracts". In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA1 (Apr. 2022), pp. 1–27. DOI: 10.1145/3527321. URL: https://dl.acm.org/doi/10.1145/3527321 (Accessed Apr. 6, 2026).

[23] Robin Hanson. "Combinatorial Information Market Design". In: *Information Systems Frontiers* 5.1 (Jan. 2003), pp. 107–119. DOI: 10.1023/A:1022058209073. URL: https://link.springer.com/10.1023/A:1022058209073 (Accessed Feb. 23, 2026).

[24] Joel Hasbrouck, Thomas J. Rivera, and Fahad Saleh. "An Economic Model of a Decentralized Exchange with Concentrated Liquidity". In: *Management Science* (Feb. 2025), mnsc.2024.04510. DOI: 10.1287/mnsc.2024.04510. URL: https://pubsonline.informs.org/doi/10.1287/mnsc.2024.04510 (Accessed Apr. 6, 2026).

[25] Lioba Heimbach, Eric Schertenleib, and Roger Wattenhofer. "Risks and Returns of Uniswap V3 Liquidity Providers". In: *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*. Cambridge MA USA: ACM, Sept. 2022, pp. 89–101. DOI: 10.1145/3558535.3559772. URL: https://dl.acm.org/doi/10.1145/3558535.3559772 (Accessed Apr. 6, 2026).

[26] Roger D. Huang and Hans R. Stoll. "The Components of the Bid-Ask Spread: A General Approach". In: *Review of Financial Studies* 10.4 (Oct. 1997), pp. 995–1034. DOI: 10.1093/rfs/10.4.995. URL: https://academic.oup.com/rfs/article-lookup/doi/10.1093/rfs/10.4.995 (Accessed Apr. 6, 2026).

[27] Jupiter. *Jupiter Developer Documentation*. 2026. URL: https://dev.jup.ag/docs/get-started (Accessed May 5, 2026).

[28] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. "DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation". In: *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA: Internet Society, 2011.

[29] Bogdan Korel and Janusz Laski. "Dynamic Program Slicing". In: *Information Processing Letters* 29.3 (Oct. 1988), pp. 155–163. DOI: 10.1016/0020-0190(88)90054-

3. URL: https://linkinghub.elsevier.com/retrieve/pii/0020019088900543 (Accessed Apr. 6, 2026).

[30] Albert S. Kyle. "Continuous Auctions and Insider Trading". In: *Econometrica* 53.6 (Nov. 1985), p. 1315. DOI: 10.2307/1913210. JSTOR: 1913210. URL: https://www.jstor.org/stable/1913210?origin=crossref (Accessed Apr. 6, 2026).

[31] Alfred Lehar and Christine Parlour. "Decentralized Exchange: The Uniswap Automated Market Maker". In: *The Journal of Finance* 80.1 (Feb. 2025), pp. 321–374. DOI: 10.1111/jofi.13405. URL: https://onlinelibrary.wiley.com/doi/10.1111/jofi.13405 (Accessed Apr. 6, 2026).

[32] Thomas Lengauer and Robert Endre Tarjan. "A Fast Algorithm for Finding Dominators in a Flowgraph". In: *ACM Transactions on Programming Languages and Systems* 1.1 (Jan. 1979), pp. 121–141. DOI: 10.1145/357062.357071. URL: https://dl.acm.org/doi/10.1145/357062.357071 (Accessed Apr. 8, 2026).

[33] *LiteSVM/Litesvm*. LiteSVM. Apr. 2026. URL: https://github.com/LiteSVM/litesvm (Accessed Apr. 4, 2026).

[34] Jason Milionis, Ciamac C. Moallemi, Tim Roughgarden, and Anthony Lee Zhang. *Automated Market Making and Loss-Versus-Rebalancing*. May 2024. DOI: 10.48550/arXiv.2208.06046. arXiv: 2208.06046 [q-fin]. URL: http://arxiv.org/abs/2208.06046 (Accessed Feb. 23, 2026).

[35] Orca. *Orca-so/Whirlpools*. Orca. URL: https://github.com/orca-so/whirlpools (Accessed Apr. 9, 2026).

[36] *Orca Documentation*. URL: https://docs.orca.so (Accessed Apr. 8, 2026).

[37] Andreas Park. "The Conceptual Flaws of Decentralized Automated Market Making". In: *Management Science* 69.11 (Nov. 2023), pp. 6731–6751. DOI: 10.1287/mnsc.2021.02802. URL: https://pubsonline.informs.org/doi/10.1287/mnsc.2021.02802 (Accessed Apr. 6, 2026).

[38] *Raydium*. URL: https://raydium.io (Accessed Apr. 8, 2026).

[39] *Rpcpool/Yellowstone-Grpc*. Triton One RPC. Apr. 2026. URL: https://github.com/rpcpool/yellowstone-grpc (Accessed Apr. 4, 2026).

[40] Jakub Sliwinski, Quentin Kniep, Roger Wattenhofer, and Fabian Schaich. "Halting the Solana Blockchain with Epsilon Stake". In: *Proceedings of the 25th International Conference on Distributed Computing and Networking*. Chennai India: ACM, Jan. 2024, pp. 45–54. DOI: 10.1145/3631461.3631553. URL: https://dl.acm.org/doi/10.1145/3631461.3631553 (Accessed Apr. 6, 2026).

[41] Sven Smolka, Jens-Rene Giesen, Pascal Winkler, Oussama Draissi, Lucas Davi, Ghassan Karame, and Klaus Pohl. "Fuzz on the Beach: Fuzzing Solana Smart Contracts". In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. Copenhagen Denmark: ACM, Nov. 2023, pp. 1197–1211. DOI: 10.1145/3576915.3623178. URL: https://dl.acm.org/doi/10.1145/3576915.3623178 (Accessed Apr. 6, 2026).

[42] *Solana-Labs/Whitepaper*. Solana Labs. Apr. 2026. URL: https://github.com/solana-labs/whitepaper (Accessed Apr. 4, 2026).

[43] *Solana's Proprietary AMM Revolution*. Aug. 2025. URL: https://www.helius.dev/blog/solanas-proprietary-amm-revolution (Accessed Apr. 7, 2026).

[44] Meet Udeshi, Prashanth Krishnamurthy, Hammond Pearce, Ramesh Karri, and Farshad Khorrami. "REMaQE: Reverse Engineering Math Equations from Executables". In: *ACM Transactions on Cyber-Physical Systems* 8.4 (Oct. 2024), pp. 1–25. DOI: 10.1145/3699674. URL: https://dl.acm.org/doi/10.1145/3699674 (Accessed Apr. 6, 2026).

[45] Mark Weiser. "Program Slicing". In: *IEEE Transactions on Software Engineering* SE-10.4 (July 1984), pp. 352–357. DOI: 10.1109/TSE.1984.5010248. URL: https://ieeexplore.ieee.org/document/5010248/ (Accessed Apr. 6, 2026).

[46] *What Is eBPF? An Introduction and Deep Dive into the eBPF Technology*. URL: https://ebpf.io/what-is-ebpf/ (Accessed Apr. 8, 2026).

[47] Xiangyu Zhang, R. Gupta, and Youtao Zhang. "Precise Dynamic Slicing Algorithms". In: *25th International Conference on Software Engineering, 2003. Proceedings.* Portland, OR, USA: IEEE, 2003, pp. 319–329. DOI: 10.1109/ICSE.2003.1201211. URL: http://ieeexplore.ieee.org/document/1201211/ (Accessed Apr. 6, 2026).

[48] Jiahua Xu, Krzysztof Paruch, Simon Cousaert, and Yebo Feng. "SoK: Decentralized Exchanges (DEX) with Automated Market Maker (AMM) Protocols". In: *ACM Computing Surveys* 55.11 (Nov. 2023), pp. 1–50. DOI: 10.1145/3570639. URL: https://dl.acm.org/doi/10.1145/3570639 (Accessed Feb. 23, 2026).

[49] Shenghao Yuan, Zhuoruo Zhang, Jiayi Lu, David Sanan, Rui Chang, and Yongwang Zhao. "A Complete Formal Semantics of eBPF Instruction Set Architecture for Solana". In: *Proceedings of the ACM on Programming Languages* 9.OOPSLA1 (Apr. 2025), pp. 1–27. DOI: 10.1145/3720414. URL: https://dl.acm.org/doi/10.1145/3720414 (Accessed Apr. 6, 2026).

Appendix A

Recovered propAMM Quoting Specifications

A.1 Worked Example: Reverse-Engineering BisonFi

This section walks through the reverse-engineering of the BisonFi program, applying the methodology of Section 4.2. Each iteration starts from a soak case the current quoter cannot reproduce. The failing trace then reveals something new, and the iteration ends with a quoter that addresses the case that broke the previous one. The trace excerpts are shortened to keep them readable; trivial byte-unpacking is left out.

Inputs to the process

The inputs are the same for every protocol, and are all obtained mechanically as described in Section 4.2. First, the deployed program binary, via `solana program dump`. Second, a fresh mainnet snapshot of the pool account at slot S_0 together with its base and quote vaults, fetched in a single atomic RPC call. Third, a test transaction template that routes a swap through the Jupiter aggregator into the BisonFi program via CPI. None of these requires understanding the protocol.

Reading the trace excerpts

Every trace excerpt below is the output of `trace_value.py` (Section 4.2.1). Starting from the observed swap output, the tool walks the recorded sBPF execution backwards through every register and memory dependency it touched. The result is the pseudocode listing reproduced here. Each leaf carries a tag for where the program got it: `[IN] xN = value @runtime_addr [account+offset]` is a memory load from an account, with the bracketed tag giving the account and offset. `[ARG]` is a value from the instruction's input data. `[CONST]` is an immediate value in the binary. `[SYS]` is a read from a Solana sysvar (e.g. the runtime clock). The `tN = ...` lines are intermediate

computations. Indices skip where the tool numbers leaves we don't show; trivial byte-unpacking that just reassembles `u64`s is left out to keep the listings short.

Iteration 0: first base-to-quote trace

We run the simulator with a base-to-quote swap, e.g. 1 SOL expecting roughly $93 of USDC based on the market price at the time of reverse-engineering, and invoke `trace_value.py` on the observed output amount. After leaving byte-unpacking and the multi-precision (64-bit limb) arithmetic the binary uses to multiply two large numbers, the computational core fits in a few lines. The trace below reproduces the actual on-chain values that led to the output `93352746`:

```
1  INPUTS (semantic leaves; byte-unpacking and 128-bit limb arithmetic elided):
2    [ARG]   amount = 1000000000                  # 1 SOL, from instruction data
3    [IN]    x1   = 0x1b89840000000000   @0x400002c00 [8FnX3xo2..+0x340]
4    [IN]    x2   =         0x5d5c7812   @0x400002c08 [8FnX3xo2..+0x348]
5    [IN]    x3   =                232   @0x400002c14 [8FnX3xo2..+0x354]
6    [IN]    x4   =               1000   @0x400002a30 [8FnX3xo2..+0x1D0]
7    [CONST] c1   =            2560000   # inline constant
8
9  COMPUTATION (core):
10   t1     = (x2 << 64) | x1            = 288938941067932372060434595 84
11   t2     = amount * x3                = 232000000000
12   t3     = t2 // c1                   = 90625
13   t4     = amount - t3                = 999909375
14   t5     = t4 * t1                    = ...          ; 128-bit multiply
15   t6     = t5 >> 88                   = 93352746274
16   result = t6 // x4                   = 93352746
```

Observation 1: Q-Format Multiplication Stored in Two Limbs. We can first observe (line 10) that the values `x1` and `x2`, read from adjacent pool offsets `0x340` and `0x348`, are combined to produce `t1`. From the trace, we can assume that due to the left shift by 64, `t1` is a 128-bit value where `x1` is the lower 64 bits and `x2` the upper 64 bits. This number is then multiplied by the net amount (line 14) and right-shifted by 88 bits (line 15). The right shift by 88 suggests `t1` is a fixed-point number with 88 bits of fractional precision.

Moreover, the shape $(\ldots)\cdot$ `t1` $\gg$ `88` looks like a Q-format fixed-point multiplication: if `t1` were a coefficient scaled by 2^{88}, then the right-shift by 88 would divide it back out. We try applying that interpretation and check whether the resulting number resembles anything we might recognise. Indeed, $\texttt{t1}/2^{88} \approx 93.36$, which is roughly the SOL/USDC price at slot S_0. The Q88 reading therefore fits. We therefore name the combined 128-bit value `price_fp` and its two limbs `price_hi` (`x1` at `0x340`) and `price_lo` (`x2` at `0x348`),

and conjecture the price model

$$y \;=\; \big((x - \text{something}) \cdot \texttt{price_fp}\big) \gg 88 \;/\; \texttt{x4}.$$

Observation 2: Trailing Divide of Undetermined Role. The final step (line 16, `result = t6 // x4`) divides by `x4 = 1000`. From this single trace the role is undecidable: `1000` might be a protocol-wide constant for output precision, or it varies per pool and compensates for the token-decimal difference ($10^{9-6} = 10^3$ happens to match SOL's 9 decimals minus USDC's 6). We refuse to commit to a name. The quoter reads the value from offset `0x1D0` rather than hard-coding it, the leaf stays labelled `x4` in our working notes, and resolution is deferred until non-SOL/USDC pools enter the soak.

Observation 3: Fractional Deduction that Behaves like a Fee. Before the Q-format multiplication, the trace does three steps (lines 11–13): `t2 = amount · x3`, `t3 = t2 // c1`, `t4 = amount − t3`. With `x3 = 232` and `c1 = 2,560,000`, the deduction `t3 = 90,625` is ≈ 9 bps of the input, and `t4` is what the price multiplication subsequently consumes (line 14). A small fraction of every swap that is removed before the price multiplication is what we would call a trading fee. The denominator `c1 = 2,560,000` factors as $256 \cdot 10{,}000$, suggesting a 2^8 scaling on top of a bps denominator; we adopt the protocol's own scaling rather than remapping it. We name `x3` as `fee_b2q` (restricting the direction tag until a Q2B trace confirms a mirror field exists) and `c1` as `fee_denom`.

We encode the conjectured formula in Python:

```
def quote_b2q_v0(pool, amount):
    price_hi = u64(pool, 0x340)
    price_lo = u64(pool, 0x348)
    fee_b2q = u16(pool, 0x354)
    x4 = u64(pool, 0x1D0)                       # role still undetermined
    price_fp = (price_lo << 64) | price_hi      # 128-bit Q88 coefficient
    fee_denom = 2_560_000                       # inline constant from trace
    fee = amount * fee_b2q // fee_denom
    net = amount - fee
    return ((net * price_fp) >> 88) // x4
```

The soak (Section 4.1.5), restricted to B2Q on `8FnX3xo2..`, passes on small inputs but reveals slippage above roughly a few thousand SOL: e.g. a 5,000 SOL swap returns `466,407,376,496` where our quoter predicts `466,763,731,370`, a ≈ 7.6 bps overstatement. The model seems correct for small swaps but is missing something that is applied as the swap consumes more liquidity.

Iteration 1: discovering tier consumption

To find it, we re-trace a failing input (a 5,000 SOL swap, which the iter-0 quoter overstates by ≈ 7.6 bps) and run `trace_value.py` again. The trace is now several hundred steps long, against iteration 0's $\sim$50, and shows new leaves we did not see for the 1 SOL case:

```
1  TRACE EXCERPT (B2Q, 5000 SOL, byte-unpacking and limb arithmetic elided):
2    [ARG]   amount = 5000000000000          ; 5000 SOL, from instruction data
3    [IN]    y1   = ...                      ; large value from a different account
4    [IN]    y2   =           150     @0x400002c1a [8FnX3xo2..+0x37A]
5    [IN]    y3   =          1024     @0x400002c12 [8FnX3xo2..+0x352]
6    [IN]    y4   =           100     @0x400002a24 [8FnX3xo2..+0x124]
7    [IN]    y5   =    0xfffffc00     @0x400002a28 [8FnX3xo2..+0x128]   ; i32 =
       -1024
8    ; (further leaves at 0x134, 0x144, 0x154 follow a tight 16-byte layout)
9
10   ; 1) The iteration-0 pattern appears to be the same at the start
11   s2     = amount * fee_b2q                ; (same as iter 0)
12   s3     = s2 // fee_denom                 ; (same as iter 0)
13   s4     = amount - s3                     ; net
14   s5     = s4 * price_fp                   ; 128-bit multiply
15   s6     = s5 >> 88
16   s7     = s6 // x4                        ; partial output
17
18   ; (a capacity step on s4 is elided; rem = s4 - cap0 spills to the next block)
19
20   ; 2) ...but then a similar pattern repeats with a scaled coefficient ----
21   u1     = fee_denom + (1 * y5) = 2558976  ; signed add (y5 negative)
22   u2     = price_fp * u1 // fee_denom      ; coefficient REDUCED slightly
23   u3     = y1 * y2 // 10000                ; some capacity bound
24   u4     = min(rem, u3)
25   u5     = (u4 * u2) >> 88
26   u6     = u5 // x4                        ; second partial output
27
28   ; 3) ...and then again, with the signed add stepped further...
29   v1     = fee_denom + (2 * y5) = 2557952
30   v2     = price_fp * v1 // fee_denom
31   v3     = y1 * y4 // 10000                ; capacity now reads y4
32   v4     = min(rem', v3)
33   v5     = (v4 * v2) >> 88
34   v6     = v5 // x4                        ; third partial output
35   ; 4) (two more repetitions follow with the signed add stepped to 3*y5 and 4*y5,
36   ;  capacities read from offsets 0x134 and 0x144...)
37   result = s7 + u6 + v6 + ...              ; sum of partial outputs
```

Observation 4: Iter-0 Formula Is Only the First Chunk. The opening of the new trace (lines 11–16) is byte-for-byte the iteration-0 computation, but its terminal value `s7`

is no longer the final output: the final `result` on line 37 sums it with several more partial outputs. The iter-0 formula was correct for the part of the swap consumed by this first chunk, then a different rule took over. That is why iteration 0 passed for small inputs (entirely contained within the first chunk) and failed for large ones.

Observation 5: Each Repetition Scales the Coefficient by a Small Factor. After the iter-0 block, the trace contains a sequence of near-identical computations `u1..u6` (lines 21–26), `v1..v6` (lines 29–34), ..., all using the iter-0 shape but with the coefficient adjusted. From the rebuild step at the top of each repetition (lines 21–22, 29–30), the adjustment has the form

$$\text{adj_coeff}_i \;=\; \texttt{price_fp} \cdot \big(\texttt{fee_denom} + i \cdot \texttt{y5}\big) \;/\; \texttt{fee_denom},$$

where the index i steps from 1 across repetitions and `y5 = 0xfffffc00` (line 7) is a signed 32-bit integer ($= -1024$). The denominator `fee_denom = 2,560,000` is the same constant we already met in the fee step. Each step therefore multiplies the coefficient by a factor of $1 - i \cdot 1024/2{,}560{,}000 \approx 1 - 0.04\% \cdot i$, slightly less than 1. The effective price worsens monotonically as the swap consumes deeper layers, which intuitively makes sense as a convex-slippage mechanism. Because `y5` is the constant numerator that drives each per-step adjustment, we name it `adj_const`.

Observation 6: Capacities Come from a Leaf Times a Per-Tier Multiplier. Second, each repetition's capacity (lines 23, 31, ...) has the shape `y1 · m / 10,000`, where the multiplier `m` varies per repetition: `y2 = 150` (line 4) for the first repetition, `y4 = 100` (line 6) for the second, ... Cross-referencing `y1` (declared as "a large value from a different account" on line 3) with the snapshot's pool vault balances, we find it matches the quote-vault token balance. We name `y1` as `v_q`. The division by `10,000` is the standard bps denominator, so each `m` is a per-tier bps fraction of `v_q`. Combining Observations 5 and 6 we can see the swap is consumed across a fixed sequence of depth tiers, each capped by a fraction of the quote vault and each priced via a multiplicatively-adjusted coefficient.

Observation 7: Remaining Tier Entries Live in a Packed 4-Entry Table. The multiplier `y4` and the adjustment numerator `y5` come from offsets `0x124` and `0x128` (lines 6–7), and there are further leaves at `0x134`, `0x144` and `0x154`. That is an array of four 16-byte records starting at `0x120`. Reading the surrounding bytes confirms each record as `[pad(4), mult(u32), adj_val(i32), pad(4)]`: the multiplier (`y4` for the first record) sits at +4, the signed adjustment numerator (`y5` for the first record) at +8. We call this the B2Q tier table; it stores the per-tier capacity multiplier and the per-tier numerator that combines with `fee_denom` in Observation 5.

Encoding Observations 1–7 in Python and restricting to base-to-quote:

```
def quote_b2q_v1(pool, vault_q_balance, amount):
    price_hi   = u64(pool, 0x340)
    price_lo   = u64(pool, 0x348)
    fee_b2q    = u16(pool, 0x354)
    x4         = u64(pool, 0x1D0)         # role still undetermined
    mult_0     = u16(pool, 0x37A)         # tier-0 capacity multiplier
    adj_const  = u16(pool, 0x352)
    fee_denom  = 2_560_000
    price_fp   = (price_lo << 64) | price_hi

    fee = amount * fee_b2q // fee_denom
    net = amount - fee
    v_q = vault_q_balance

    def tier_quote(net_in, adj_val, cap_mult):
        adj_price = price_fp * (fee_denom + adj_val) // fee_denom
        cap       = v_q * cap_mult // 10_000
        used      = min(net_in, cap)
        return used, ((used * adj_price) >> 88) // x4

    # Tier 0: raw price_fp, capacity scaled by mult_0
    used, out_total = tier_quote(net, 0, mult_0)
    rem = net - used

    # Tiers 1..4: (mult, adj_val) read from the packed table at 0x120..0x15F
    for i in range(4):
        entry_off = 0x120 + i * 0x10
        cap_mult  = u32(pool, entry_off + 4)
        adj_val   = i32(pool, entry_off + 8)
        if rem == 0 or cap_mult == 0:
            break
        used, out_i = tier_quote(rem, adj_val, cap_mult)
        out_total += out_i
        rem       -= used

    return out_total
```

The B2Q-only soak now passes for all input sizes on `8FnX3xo2...`

Iteration 2: Adding the Quote-to-Base direction

The quoter so far handles only base-to-quote. The natural next step is to run the simulator with a small quote-to-base swap and trace it. We send 100 USDC expecting roughly 1.07 SOL and `trace_value.py` produces a trace that mirrors the B2Q shape but takes its parameters from different offsets:

```
TRACE EXCERPT (Q2B, 100 USDC):
```

```
2   [ARG]   amount = 100000000                  ; 100 USDC, from instruction data
3   [IN]    x1   = 0x1b89840000000000   @0x400002c00 [8FnX3xo2..+0x340]
4   [IN]    x2   =         0x5d5c7812   @0x400002c08 [8FnX3xo2..+0x348]
5   [IN]    x4   =               1000   @0x400002a30 [8FnX3xo2..+0x1D0]
6   [IN]    z1   =                 26   @0x400002c16 [8FnX3xo2..+0x356]
7   [CONST] c1   =            2560000              ; same denominator
8
9   t1     = (x2 << 64) | x1            = 288938941067932372060434595 84
10  t2     = amount * z1                = 2600000000
11  t3     = t2 // c1                   = 1015
12  t4     = amount - t3                = 99998985
13  t5     = (t4 * x4) << 88            ; scale UP by 88 bits and by x4
14  result = t5 // t1                   = 1071097815
```

Observation 8: Mirror Fee Field at a Different Offset. The Q2B trace reuses `x1`, `x2`, `x4`, and `c1` unchanged from the B2Q trace (lines 3–5, 7), the price is one number, the trailing divisor is one number, and the fee denominator is one constant. The only new leaf is `z1 = 26` at `0x356` (line 6), which sits two bytes after `fee_b2q` at `0x354`. The fee arithmetic on lines 10–12 is structurally identical to B2Q (`amount · leaf / fee_denom`, then subtracted from `amount`), so by the reasoning of Observation 3 the leaf plays the same role on the Q2B side. We name `z1` `fee_q2b`. The protocol stores both directions' fee parameters at adjacent `u16` slots in the pool account.

Observation 9: Inverted Q-Format Multiplication. The multiplication direction inverts. Where B2Q computed `t6 = (net · price_fp)` $\gg$ `88` and then divided by `x4`, Q2B computes `t5 = (net · x4)` $\ll$ `88` on line 13 and then divides by `price_fp` on line 14. The arithmetic is structurally identical with multiply and divide swapped, and the unit roles flip: the input is now in the quote token (with `x4` decimal-scale applied before the shift), and the output is in the base token. This is the price inversion $y = x/p$ implemented in 128-bit fixed-point.

A larger Q2B input then makes the trace lengthen the same way the 5,000 SOL B2Q trace did in iteration 1: an opening single-chunk computation followed by repeated capped-divide blocks summed at the end. Running `trace_value.py` on this longer trace, the chunk-0 capacity multiplier sits at `0x378` (a `u16`, immediately before B2Q's `0x37A`), and the per-tier table lives at `0x160`–`0x19F` (immediately after the B2Q table that ends at `0x15F`). The signed adjustment numerator in each Q2B tier entry is *positive*, so the adjusted coefficient now *rises* as the tier index grows, meaning the divisor gets larger and the output gets smaller, the same convex-slippage effect as B2Q but mirrored. We patch the quoter into a bidirectional form; both B2Q and Q2B now pass on `8FnX3xo2..` for all input sizes across all refetch and re-run iterations.

Iteration 3: Expanding the soak to more pools

Pool `8FnX3xo2..` is now fully reproduced. A quoter that fits one pool is not yet a quoter that fits the protocol, however, so we expand the soak to the other BisonFi pools we have configurations for: `fc9p..` (SOL/USDT), `ckc2..` (SOL/USD1), `2vpj..` (cbBTC/USDC), and `4xke..` (USDC/USDT). The three pools paired against a 6-decimal stable pass immediately on parameters that differ in value but live at the same offsets, which is strong evidence that we recovered general logic rather than one pool's configuration. The `4xke..` pool surfaces one new fact.

Observation 10: `trail_div` Tracks the Decimal Mismatch. The USDC/USDT pool reads `trail_div = 1` (rather than `1000` as on the SOL pairs). The value varies across pools, ruling out the protocol-wide-constant hypothesis of Observation 2. Tabulating the decimal differences for each pair gives $|\Delta\text{dec}| \in \{3, 0\}$, and $10^{|\Delta\text{dec}|}$ matches `trail_div` in every case. The field compensates for the token-decimal mismatch, so we rename it `decimal_scale`. The quoter already reads the field and needs no functional change.

Observation 11: USDC/USDT Pools Use Inline Fee Stepping. The USDC/USDT pool still fails on medium and large inputs. The failing trace shows that, unlike the SOL pools, the fee is not read directly from `fee_b2q` (`0x354`). Instead the program computes:

$$\text{fee} \;=\; \texttt{fee_floor} \;+\; \texttt{fee_adj} \cdot \max\big(0,\, \lfloor(\text{new_base} - \texttt{thresh})/\texttt{step}\rfloor\big),$$

reading `fee_floor` from `0x37E`, `fee_adj` from `0x35C`, the threshold from `0x360`, and the step size from `0x368`. The SOL pools never triggered this branch because they have `step = 0` (the stepping is disabled). We add the conditional:

```
if step_size > 0:                       # stable-stable pools
    steps = max(0, (new_base - thresh) // step_size)
    fee   = fee_floor + fee_adj * steps
else:
    fee   = u16(pool, 0x354)             # cached fee, used by SOL pools
```

With this fix the soak passes on all four current pools across both directions.

Iteration 4: The Slot-Staleness Fee Penalty

The single-snapshot soak now passes on every pool and direction. A propAMM quoter must also stay correct as on-chain state ages between transactions, however. We therefore run an extended soak across many freshly-fetched snapshots taken seconds to minutes apart. Failures reappear in the form of the simulator returning *less* output than our quoter predicts, and the failure magnitude is strongly correlated with the time gap between when the snapshot was captured and when the simulator runs the transaction: older snapshots

fail by more. It seems some mechanism inside the pool penalises swaps when the pool's state is stale. Tracing one of these failing cases, the fee computation now reads several extra leaves before producing its final value:

```
1 TRACE EXCERPT (stale-pool fee path):
2   [IN]   w1   = 412394056      @0x400002908 [8FnX3xo2..+0x048]
3   [IN]   w2   = 128            @0x400002c32 [8FnX3xo2..+0x372]
4   [IN]   w3   = 102            @0x400002c3e [8FnX3xo2..+0x37E]
5   [SYS]  w4   = 412394089      @0x400000040 [SysvarC1ock..+0x00]
6   w5    = w4 - w1              = 33
7   w6    = w5 * w2              = 4224
8   w7    = max(fee_b2q, w3)     = 232    ; cached fee vs. floor
9   w8    = w7 + w6              = 4456   ; this replaces fee_b2q in the fee step
```

Observation 12: Time-Tracking Leaf and a Per-Tick Penalty Rate. We have two new leaves from this trace. First, `w1 = 412394056` at offset `0x48` (line 2) trails the runtime sysvar clock `w4` (line 5) by exactly 33 (their difference is computed on line 6). Checking the on-chain observation log (Section 4.3), the same offset increases monotonically across snapshots, advancing by one or more units between consecutive slots. A monotonic counter that tracks the chain clock is most parsimoniously interpreted as the pool's last-update slot, so we name `w1 pool_slot`. Second, `w2 = 128` at `0x372` (line 3) multiplies the slot delta and the product `w6` (line 7) is added directly into the fee total. A constant that scales a per-slot delta into a fee contribution is a penalty rate per stale slot, so we name `w2 slot_penalty`.

Observation 13: Hard Floor Replaces Fee When the Pool Is Stale. The cached fee `fee_b2q` from iteration 0 still appears, but on line 8 it is only one argument of a `max` with a new leaf `w3 = 102` at `0x37E` (line 4). When the pool is stale, the cached fee may no longer reflect current oracle prices, and the protocol enforces a minimum to compensate. We name `w3 fee_floor`. The full fee formula becomes

$$\text{fee} \;=\; \max(\text{cached_fee}, \text{fee_floor}) \;+\; \text{slot_penalty} \cdot \max(0, \text{clock} - \text{pool_slot}),$$

with the convention that when clock = pool_slot (fresh state, just rewritten by an oracle update CPI), the cached fee is used directly. Patching the quoter and re-running the extended soak across all four pools and both directions produces 100 iterations $\times 100$ input amounts $\times 8$ pool-direction combinations = 80,000 cumulative comparisons, every comparison within the ± 1 token tolerance of Section 4.1.5.

Recovered formula

Once iteration 4 has converged, the recovered closed-form expression for the BisonFi quoting function, for both directions $d \in \{\text{b2q}, \text{q2b}\}$ and with $\texttt{FD} = 2{,}560{,}000$, is:

$$
\begin{aligned}
\text{cached}_d &= \texttt{fee}_d \quad (\texttt{u16 at 0x354 for b2q, 0x356 for q2b}) \\
\text{fee_bps} &= \begin{cases} \text{cached}_d & \text{if clock} = \text{pool_slot} \\ \max(\text{cached}_d, \texttt{fee_floor}_d) + \texttt{slot_pen} \cdot (\text{clock} - \text{pool_slot}) & \text{otherwise} \end{cases} \\
\text{fee} &= \lfloor \text{amount} \cdot \text{fee_bps} / \texttt{FD} \rfloor \\
\text{net} &= \text{amount} - \text{fee} \\
\texttt{adj_price}_i &= \texttt{price_fp} \cdot (\texttt{FD} + \texttt{adj_val}_i) \;/\; \texttt{FD} \\
y_{\text{b2q}} &= \frac{1}{\texttt{decimal_scale}} \sum_{i \in \text{tiers}} \left(\min(\text{net}_i, m_i^{\text{b2q}} \cdot v_q / 10{,}000) \cdot \texttt{adj_price}_i \right) \gg 88 \\
y_{\text{q2b}} &= \texttt{decimal_scale} \cdot \sum_{i \in \text{tiers}} \frac{\min(\text{net}_i, m_i^{\text{q2b}} \cdot v_b / 10{,}000) \ll 88}{\texttt{adj_price}_i},
\end{aligned}
$$

where $\texttt{price_fp} = (\texttt{price_lo} \ll 64) \mid \texttt{price_hi}$ is the 128-bit Q88 price coefficient assembled from `u64` loads at `0x340` and `0x348`; net_i is the remainder after tiers $0..i-1$ have consumed their share; tier-0 takes $\texttt{adj_val}_0 = 0$ and m_0^{b2q}, m_0^{q2b} from the dynamic `u16` fields at `0x37A` and `0x378` respectively; and subsequent tiers' $(m_i, \texttt{adj_val}_i)$ pairs come from the B2Q tier table at `0x120`–`0x15F` and the Q2B tier table at `0x160`–`0x19F`.

Validation outcome

The final soak completes 100 iterations ×100 input amounts ×2 directions ×4 v3 pools = 80,000 comparisons against the live binary, every comparison within the ±1 token tolerance of Section 4.1.5. The same procedure was applied to all seven propAMMs in this dissertation, with a cumulative regression-snapshot count of 769.

The remaining sections give the formal quoting specification of each protocol. Notation follows the conventions introduced in Section 6.2.1.

A.2 AlphaQ

Parameters.

Symbol	Type	Description
P	u64	Operator-set mid-price (scaled by $10^{10+d_a-d_b}$)
Q_q, Q_b	u64	Direction-specific midpoint multipliers ($\times 10^{12}$)
Q_{flat}	u64	Safety multiplier for the flat fallback price
Q_{bnd}	u64	Hard ceiling/floor multiplier (tight clamp on Q_{adj})
L	u64	Liquidity capacity (target inventory in quote units)
α	u64	$\sqrt{x}$ trade-impact coefficient
β	u64	Quadratic large-trade penalty
γ	u64	$\sqrt{\lvert r\rvert}$ inventory-impact coefficient
δ	u64	Quadratic inventory-impact coefficient
v_a	u64	Base vault balance
d_a, d_b	u8	Base / quote token decimals

Derived scales: $S_{\text{eff}} = 10^{10+d_a-d_b}$, $\sigma = 10^{18-d_b}$ (isqrt scale), $\Delta = 10^{5+2d_b}$ (quadratic divisor).

Step 1: Fee computation. No explicit fee: the spread is embedded entirely in the multiplier adjustment. The α, β terms widen the spread with trade size; the γ, δ terms reward trades that move inventory toward L and penalise trades that move it away.

Step 2: Price adjustment. Let $\text{inv} = \lfloor v_a \cdot P / S_{\text{eff}} \rfloor$ be the quote-equivalent inventory, $r = L - \text{inv}$ the remaining capacity, and $s = \text{sgn}(r)$ (so $s = -1$ when $\text{inv} > L$). For B2Q the input is first converted to quote units, $x_{\text{eq}} = \lfloor xP / S_{\text{eff}} \rfloor$; for Q2B the input is already in quote units and we set $x_{\text{eq}} = x$. Then:

$$I_{\surd} = \left\lfloor \frac{\mathsf{isqrt}(x_{\text{eq}} \cdot \sigma) \cdot \alpha}{10^{14}} \right\rfloor, \qquad I_{\beta} = \left\lfloor \frac{x_{\text{eq}}^2 \cdot \beta}{\Delta} \right\rfloor, \tag{A.1}$$

$$J_{\gamma} = \left\lfloor \frac{\mathsf{isqrt}(\lvert r\rvert \cdot \sigma) \cdot \gamma}{10^{14}} \right\rfloor, \qquad J_{\delta} = \left\lfloor \frac{\lvert r\rvert^2 \cdot \delta}{\Delta} \right\rfloor.$$

The complex multiplier is

$$Q_{\text{cplx}} \;=\; Q_d \;\mp\; (I_{\surd} + I_{\beta}) \;+\; s \cdot (J_{\gamma} + J_{\delta}), \tag{A.2}$$

where the upper sign $(-)$ is for B2Q (with $Q_d = Q_q$) and the lower $(+)$ for Q2B (with $Q_d = Q_b$). This introduces a penalty or reward depending on whether the trade moves inventory in the desired direction towards the target L.

Step 3: Output. Moreover, a flat fallback multiplier and clamp are applied on top of Q_{cplx}. For B2Q, larger Q means a better price for the trader, so the flat formula is the

floor and Q_{bnd} is the floor of the floor:

$$\begin{aligned} Q_{\text{adj}}^{B2Q} &= \max\big(\min(Q_{\text{cplx}},\ 2{\cdot}10^{12} - Q_{\text{flat}}),\ 2{\cdot}10^{12} - Q_{\text{bnd}}\big), \\ P_{\text{adj}} &= \left\lfloor Q_{\text{adj}}^{B2Q} \cdot P/10^{12} \right\rfloor, \qquad\qquad y = \lfloor x \cdot P_{\text{adj}}/S_{\text{eff}} \rfloor . \end{aligned} \tag{A.3}$$

For Q2B, larger Q means a worse price for the trader, so the roles of min/max swap:

$$\begin{aligned} Q_{\text{adj}}^{Q2B} &= \min\big(\max(Q_{\text{cplx}},\ Q_{\text{flat}}),\ Q_{\text{bnd}}\big), \\ P_{\text{adj}} &= \left\lceil Q_{\text{adj}}^{Q2B} \cdot P/10^{12} \right\rceil, \qquad\qquad y = \lfloor x \cdot S_{\text{eff}}/P_{\text{adj}} \rfloor . \end{aligned} \tag{A.4}$$

Q_{bnd} is a worst-case guarantee: e.g. when the pool is over-inventoried and $s \cdot (J_\gamma + J_\delta)$ is large, the trader receives no better than the price implied by Q_{bnd}.

Q2B differences. B2Q subtracts $I_{\surd} + I_\beta$ from Q_q and floors P_{adj}; Q2B adds them to Q_b and ceils. The inventory term $s \cdot (J_\gamma + J_\delta)$ enters with the same sign in both directions, so a long-inventory pool ($r > 0$) cheapens base-out swaps and penalises base-in swaps symmetrically.

Reference implementation. The full byte-offset parsing and integer-arithmetic implementation is in `scripts/alphaq/quoter.py` of the released repository (`https://github.com/TraceSVM/propamm-simulator`).

A.3 SolFi V2

Parameters.

Symbol	Type	Description
P	u64	Oracle mid-price
Δ	u64	Oracle divisor
v_a, v_q	u64	Base and quote vault balances
m	u64	Base fee multiplier
$\delta_{b2q}, \delta_{q2b}$	u32	Per-direction fee offsets
σ_{base}	f32	Static spread floor
κ	u64	Vault correction multiplier
Δ_2	u64	Vault correction divisor
$\hat{c}$	u64	Correction-cap activation threshold
c^+, c^-	u64	Correction caps (positive / negative imbalance)
$\phi_{\max}$	u64	Hard fee cap (scale 10^7)
$s_{\text{oracle}}, s_{\text{pool}}, s_{\text{clk}}$	u64	Oracle, pool, and clock slots
$\mathbf{b}_t, \mathbf{f}_t$	arrays	Per-direction tier table (7 boundaries, 8 fees)
$\mathbf{b}_s, \mathbf{f}_s$	arrays	Staleness table: slot-age boundaries and widths
$\mathbf{b}_{s4}, \mathbf{f}_{s4}$	arrays	S4 staleness-offset table
ℓ_{s4}	u32	S4 lookup level (per direction)

Derived: $S_{\text{eff}} = \max(s_{\text{clk}}, s_{\text{pool}})$, $t = \max(0,\ S_{\text{eff}} - s_{\text{oracle}})$ (oracle age in slots).

Step 2: Price adjustment. The oracle price is corrected for vault imbalance, with truncation toward zero and a magnitude cap activated beyond a threshold $\hat{c}$:

$$\begin{aligned} \text{imb} &= v_q - \lfloor v_a \cdot P/\Delta \rfloor, \qquad \text{corr}_{\text{raw}} = \lfloor |\text{imb}| \cdot \kappa \,/\, (2\Delta_2) \rfloor, \\ |\text{corr}| &= \begin{cases} \min(\text{corr}_{\text{raw}},\ c^{\text{sgn(imb)}}) & \text{corr}_{\text{raw}} > \hat{c}, \\ \text{corr}_{\text{raw}} & \text{otherwise}, \end{cases} \\ P_{\text{adj}} &= \left\lfloor P \cdot \frac{10^7 + \text{sgn(imb)} \cdot |\text{corr}|}{10^7} \right\rfloor . \end{aligned} \tag{A.5}$$

A direction-specific oracle-skip predicate (combining the sign of corr, the relative ordering of s_{pool} and s_{oracle}, and the staleness-table bracket of t) selects $P_{\text{use}} \in \{P, P_{\text{adj}}\}$; when the correction would worsen the active side the raw oracle is used.

Step 1: Fee computation. Three components combine into a single fee. The staleness factor and tier weight are interpolated from their tables (linear within each segment):

$$w_{\text{stale}}(t) = \text{interp}(\mathbf{b}_s,\ \mathbf{f}_s,\ t), \qquad\qquad w_{\text{tier}}(\cdot) = \text{interp}(\mathbf{b}_t,\ \mathbf{f}_t,\ \cdot).$$

An S4 staleness offset σ_{s4} widens the fee when the pool's oracle slot advanced but the price did not and the pool is not behind the oracle:

$$\sigma_{s4} = \begin{cases} \text{interp}(\mathbf{b}_{s4},\, \mathbf{f}_{s4},\, \ell_{s4}) & \ell_{s4} > 0,\ \ s_{\text{pool}} \geq s_{\text{oracle}},\ \ \text{snapshot} = \text{current}, \\ 0 & \text{otherwise.} \end{cases} \tag{A.6}$$

The total widening, direction offset, and base spread are assembled with on-chain f32 arithmetic and capped at $\phi_{\max}$:

$$\begin{aligned} W &= \sigma_{s4} + \lfloor w_{\text{stale}}(t) \cdot w_{\text{tier}} \cdot m \,/\, 10^6 \rfloor, \\ \phi &= \min\Big(\text{f32}\big(\delta_d + W\big) + \text{f32}(\sigma_{\text{base}}),\ \ \phi_{\max}\Big), \end{aligned} \tag{A.7}$$

where $\delta_d \in \{\delta_{b2q}, \delta_{q2b}\}$ and $\text{f32}(\cdot)$ denotes IEEE 754 single-precision cast (matching on-chain behaviour). If $\delta_d + W \geq 10^6$ the spread term is bypassed.

Step 3: Output. The raw output uses the selected oracle, then the multiplicative fee is applied and the result is clamped by the outgoing vault:

$$y_{\text{raw}} = \left\lfloor \frac{x \cdot P_{\text{use}}}{\Delta} \right\rfloor, \qquad\qquad y = \boxed{\min\left(\left\lfloor y_{\text{raw}} \cdot \frac{10^7 - \phi}{10^7} \right\rfloor,\ \ v_q\right)}. \tag{A.8}$$

For B2Q the tier interpolation uses $w_{\text{tier}}(y_{\text{raw}})$, indexing depth in quote units.

Q2B differences. Invert the raw computation and use the Q2B parameter set: $y_{\text{raw}} = \lfloor x \cdot \Delta / P_{\text{use}} \rfloor$, $w_{\text{tier}} = \text{interp}(\mathbf{b}_t^{q2b},\, \mathbf{f}_t^{q2b},\, x)$ (interpolated on the input amount x, already in quote units), $\delta_d = \delta_{q2b}$, and the output clamp uses v_a.

Reference implementation. The full byte-offset parsing and integer-arithmetic implementation is in `scripts/solfiv2/quoter.py` of the released repository (`https://github.com/TraceSVM/propamm-simulator`).

A.4 GoonFi V2

The protocol operates in one of two modes determined by a flag $\texttt{eq} = [v_s = v_{sp}]$ that compares two stored vault-correction parameters. The flag selects (i) the vault correction (simple vs. piecewise) and (ii) the fee composition on the fully-stale path. Within each mode, three sub-paths are selected by the staleness counters $t_a = s_{\text{clk}} - s_A$ and $t_b = s_{\text{clk}} - s_B$.

Parameters.

Symbol	Type	Description
$P_{\text{bid}}, P_{\text{ask}}$	u64	Oracle bid/ask prices
s_{oracle}	u32	Slot of last oracle update
σ_p	u32	Spread scaling parameter ($\times 10^{-6}$)
s_A, s_B	u64	Slot of last pool-state update / last trade
p_d	u64	Cumulative position (per direction $d \in \{b2q, q2b\}$)
$\mathbf{w}_d[10]$	u64	Per-tier spread array (per direction)
$\mathbf{B}_d[10]$	u64	Cumulative tier boundaries (per direction)
ϕ_{base}	u64	Base fee
η	u8	Staleness offset for fee scaling
v_{dec}	u64	Position decay rate (per slot)
v_{ref}	u64	Reference vault balance
v_a	u64	Current base-vault balance
v_{th}	u64	Vault-imbalance threshold (piecewise breakpoint)
v_s, v_{sp}, v_{sh}	u64	Vault-correction divisors (simple, piecewise, half: $v_{sh} = v_{sp}/2$)

Step 1: Fee computation. The effective per-tier spread is built from a base spread w_i plus a staleness-scaled fee ϕ_{stale}:

$$\phi_{\text{stale}} = \phi_{\text{base}} \cdot \text{adj}_{\text{st}}, \qquad w_{\text{eff},i} = \left\lfloor \frac{(w_i + \phi_{\text{stale}}) \cdot \sigma_p}{10^6} \right\rfloor, \tag{A.9}$$

where $\text{adj}_{\text{st}} = \max(\ell,\ s_{\text{clk}} - s_{\text{oracle}} - \eta)$ with floor $\ell = 1$ if $t_a = 0$ or the oracle was updated this slot, and $\ell = 0$ otherwise.

Step 2: Price adjustment. The oracle base price (P_{bid} for B2Q, P_{ask} for Q2B) is corrected for vault imbalance by c_{vault}, signed by $\text{sgn}(\delta)$:

$$c_{\text{vault}} = \begin{cases} \left\lfloor \dfrac{|\delta| \cdot 10^6 + v_\bullet/2}{v_\bullet} \right\rfloor & \mathsf{eq} \text{ or } |\delta| \le v_{\text{th}}, \\ \left\lfloor \dfrac{v_{\text{th}} \cdot 10^6 + v_s/2}{v_s} \right\rfloor + \left\lfloor \dfrac{(|\delta| - v_{\text{th}}) \cdot 10^6 + v'/2}{v'} \right\rfloor & \mathsf{eq}^c \text{ and } |\delta| > v_{\text{th}}, \end{cases} \tag{A.10}$$

$$P_{\text{adj}} = \left\lfloor P \cdot \frac{10^6 + \text{sgn}(\delta) \cdot c_{\text{vault}}}{10^6} \right\rfloor. \tag{A.11}$$

The simple-mode divisor is $v_\bullet = v_{sp}$ (used when eq); the piecewise excess divisor v' depends on direction and on $\text{sgn}(\delta)$:

$$v' = \begin{cases} v_{sp} & (d = b2q,\ \delta \ge 0) \text{ or } (d = q2b,\ \delta < 0), \\ v_{sh} & (d = b2q,\ \delta < 0) \text{ or } (d = q2b,\ \delta \ge 0). \end{cases}$$

The per-tier price is $\pi_i = \lfloor P_{\text{adj}}\,(10^6 - w_{\text{eff},i})/10^6 \rfloor$ (B2Q) or $\lceil P_{\text{adj}}\,(10^6 + w_{\text{eff},i})/10^6 \rceil$ (Q2B at the interpolation step; floor at the base).

Step 3: Output. Let $\pi_d(p)$ denote the interpolated price at cumulative position p, linearly between adjacent boundaries (floor for B2Q, ceiling for Q2B). The output for B2Q is

$$y = \begin{cases} \dfrac{p_{\text{new}}\,\pi_d(p_{\text{new}}) - p_{\text{old}}\,\pi_d(p_{\text{old}})}{10^{d_a}} & t_a = 0, \\ \text{position integral on } [p',\, p'+x],\ p' = \max(0,\, p_{\text{old}} - v_{\text{dec}}\,t_a), & t_b = 0, \\ \displaystyle\sum_{i \geq i_0} \frac{\min(x_{\text{rem}},\, B_{i+1} - B_i) \cdot \bar{\pi}_i}{10^{d_a}} & \text{otherwise}, \end{cases} \tag{A.12}$$

where $p_{\text{old}} = p_{b2q}$, $p_{\text{new}} = p_{b2q} + x$, and $\bar{\pi}_i$ is the input-weighted price within tier i. The starting tier $i_0 = 1$ when $B_0 = 0$ (zero-width sentinel tier), else $i_0 = 0$.

Q2B differences. The branching is mirrored with the slot-age roles swapped: the position-integral path triggers on $t_b = 0$ (instead of $t_a = 0$), and the position-decay path triggers on $t_a = 0$ when the decayed position $p' = p_{q2b} - v_{\text{dec}}\,t_b$ remains positive. When $p' \leq 0$ the path falls through to the input-weighted branch. In the fully-stale `eq` branch the spread degenerates to $w_i + \phi_{\text{base}}\,[t_a = 0]$ (no σ_p scaling). The price ratio is inverted: divide instead of multiply, $y = \lfloor 10^{d_a} x / \pi_i \rfloor$.

Reference implementation. The full byte-offset parsing and integer-arithmetic implementation is in `scripts/goonfi/quoter.py` of the released repository (`https://github.com/TraceSVM/propamm-simulator`).

A.5 Humidifi

Parameters.

Symbol	Type	Description
P	u64	Q48 fixed-point price (quote/base $\times 2^{48}$)
$L_{0,\text{hi}}, L_{0,\text{lo}}$	u64	Pool normalisation constants (encode token decimals)
ℓ	u64	Reserve-side liquidity anchor (defines Δ)
$\theta_{b2q}, \theta_{q2b}$	u64	Direction-specific spread parameters (selected by sign of $\hat{x}_1 - \hat{x}_2$)
θ_1	u64	Primary spread (used when impact is inactive)
$\theta_1^{(1)}, \theta_1^{(2)}$	u64	Replacement spreads for impact level 1 / level 2
$\bar{\theta}_1$	u64	Offset-term scaling factor
τ_1, τ_2	u64	Impact activation thresholds (level 1, level 2; quote units)
μ_1, μ_2	u64	Impact multipliers (level 1, level 2)
ξ_1, ξ_2	u64	Impact isqrt reference points
$m_0, m_1, x_1, t_0, t_{\max}, \phi_{\text{cap}}$	u32	4-segment piecewise imbalance schedule
$\phi_{\text{bps}}, \phi_{\text{den}}$	u32	Linear-mode parameters / tier-0 width threshold
κ	u64	Tier coefficient
μ_f	u64	Fee multiplier
$\lambda \in \{0, 1\}$	u8	Offset mode (`linear_mode`: 0 = isqrt, 1 = linear)
v_a, v_q	u64	Base / quote vault balances

Derived quantities.

$$S_{\text{eff}} = (L_{0,\text{hi}} \ll 16) \mid (L_{0,\text{lo}} \gg 48) \quad \text{decimal-scale divisor (shared } S_{\text{eff}}\text{, Section 6.2.1)}$$
$$\Lambda = (L_{0,\text{hi}} \ll 50) \mid (L_{0,\text{lo}} \gg 14) \quad \text{high-precision } \hat{x} \text{ divisor}$$
$$\Delta = v_a - \ell \quad \text{signed reserve deviation state}$$

Constants. $\alpha = \lceil 2^{48}/1000 \rceil$, $c_0 = \lfloor 700 \cdot 2^{48}/1000 \rfloor$, $\beta = \texttt{0x86A0000000000000}$.

Input normalisation. The input is expressed in base atoms throughout: $x' = x$ for B2Q and $x' = \lfloor x \cdot 2^{48}/P \rfloor$ for Q2B. The direction sign is $\epsilon = +1$ for B2Q and $\epsilon = -1$ for Q2B.

Step 1: Fee computation. The imbalance rebate $\mathcal{I}$ is applied only when the swap reduces the pool's reserve imbalance; otherwise $\mathcal{I} = 0$. Form the post-trade reserve gap and total

$$N = (v_a + \epsilon\, x') \cdot P, \qquad Q = (v_q - \epsilon \lfloor x' P / 2^{48} \rfloor) \ll 48,$$

and the raw signed ratio $\rho_{\text{raw}} = \lfloor (N - Q) \cdot 2^{58}/(N + Q) \rfloor \gg 11$. Rescale via the on-chain pattern $\rho = \left| \lfloor (\beta\, \rho_{\text{raw}} + \rho_{\text{raw}})/2^{48} \rfloor + \rho_{\text{raw}} \ll 16 \right|$ and set $r = \rho \gg 48$. The rebate sign condition is: B2Q rebates iff $\rho_{\text{raw}} > 0$; Q2B rebates iff $\rho_{\text{raw}} < 0$.

Linear mode ($\phi_{\text{bps}} \neq 0$):

$$\mathcal{I} = \left\lfloor \left\lfloor \frac{\phi_{\text{bps}}\, r}{\phi_{\text{den}}} \right\rfloor \cdot \frac{2^{48}}{10} \right\rfloor. \tag{A.13}$$

Tiered mode ($\phi_{\text{bps}} = 0$, $\phi_{\text{den}} \neq 0$; otherwise $\mathcal{I} = 0$):

$$t(r) = \begin{cases} 0 & r < \phi_{\text{den}}, \\ \lfloor t_0(r - \phi_{\text{den}})/(m_0 - \phi_{\text{den}}) \rfloor & \phi_{\text{den}} \le r < m_0, \\ t_0 + \lfloor (\phi_{\text{cap}} - t_0)(r - m_0)/(m_1 - m_0) \rfloor & m_0 \le r < m_1, \\ \phi_{\text{cap}} + \lfloor (t_{\max} - \phi_{\text{cap}})(r - m_1)/(x_1 - m_1) \rfloor & m_1 \le r < x_1, \\ t_{\max} & r \ge x_1, \end{cases} \qquad \mathcal{I} = \lfloor t(r) \cdot 2^{48}/10 \rfloor. \tag{A.14}$$

If the swap *increases* imbalance, $\mathcal{I} = 0$ regardless of mode.

Step 2: Price adjustment. Compute high-precision normalised reserve states before and after the trade with asymmetric rounding:

$$\hat{x}_2 = \left\lfloor |\Delta| \cdot 2^{82}/\Lambda \right\rfloor, \qquad \hat{x}_1 = \left\lfloor |\Delta + \epsilon x'| \cdot 2^{82}/\Lambda \right\rfloor + 1. \tag{A.15}$$

The $+1$ on $\hat{x}_1$ matches on-chain rounding. Slippage factor (mode fixed per pool by the impact-version field):

$$\text{sf}_2 = \left\lfloor \frac{\left(|\hat{x}_1^p - \hat{x}_2^p| \gg \sigma\right) \cdot S_{\text{eff}}}{x'} \right\rfloor + 1, \qquad (p, \sigma) = \begin{cases} (2,\ 48) & x^2 \text{ mode}, \\ (3/2,\ 24) & x^{3/2} \text{ mode}, \end{cases} \tag{A.16}$$

where $\hat{x}^{3/2} \equiv \hat{x} \cdot \mathsf{isqrt}(\hat{x})$. Let $s = -1$ when $\hat{x}_1 < \hat{x}_2$ (the trade reduces $|\Delta|$, i.e. rebalances) and $s = +1$ otherwise; the corresponding $\theta_d \in \{\theta_{b2q}, \theta_{q2b}\}$ is selected by s.

The large-trade impact H activates on the quote-equivalent size y_q (with $y_q = \lfloor Px'/2^{48} \rfloor$ for B2Q, $y_q = x$ for Q2B):

$$H = \begin{cases} 0 & y_q \le \tau_1, \\ \lfloor \mu_1 \cdot \mathsf{isqrt}(|\hat{x}_{\text{n}} - \xi_1|)/2^{24} \rfloor & \tau_1 < y_q < \tau_2, \\ \lfloor \mu_2 \cdot \mathsf{isqrt}(|\hat{x}_{\text{n}} - \xi_2|)/2^{24} \rfloor & y_q \ge \tau_2, \end{cases} \qquad \hat{x}_{\text{n}} = \lfloor x' \cdot 2^{48}/S_{\text{eff}} \rfloor. \tag{A.17}$$

When $H > 0$ the spread base is replaced: $\theta_{1,\text{eff}} = \theta_1^{(1)}$ for $\tau_1 < y_q < \tau_2$ and $\theta_1^{(2)}$ for $y_q \ge \tau_2$; otherwise $\theta_{1,\text{eff}} = \theta_1$. The offset term is active only when $H = 0$:

$$\theta_{\text{off}} = \begin{cases} \lfloor \bar{\theta}_1 \cdot \mathsf{isqrt}(\hat{x}_{\text{n}})/2^{24} \rfloor & \lambda = 0, \\ \lfloor x' \cdot \bar{\theta}_1 / S_{\text{eff}} \rfloor & \lambda = 1. \end{cases} \tag{A.18}$$

Step 3: Output. Form the signed accumulator and the effective denominator:

$$A = \mathcal{I} \;+\; s \cdot \lceil \mathrm{sf}_2 \cdot \theta_d / 2^{48} \rceil \;+\; (\theta_{1,\mathrm{eff}} + H) \;+\; \theta_{\mathrm{off}}, \tag{A.19}$$

$$N_A = \begin{cases} A & s = -1 \text{ and } H = 0, \\ \lfloor A \cdot \mu_f \cdot \alpha / 2^{48} \rfloor & \text{otherwise} \end{cases} \;+\; \kappa\,\alpha \;+\; c_0, \tag{A.20}$$

$$D_{\mathrm{eff}} = 2^{48} \;+\; \left\lfloor \frac{N_A \cdot 2^{48}}{10^4 \cdot 2^{48} + 1} \right\rfloor \;+\; 1. \tag{A.21}$$

The pass-through branch in N_A skips the fee-multiplier scaling when the trade rebalances and the impact term is inactive, producing a markedly tighter quote.

$$\boxed{y = \left\lfloor \frac{P \cdot x}{D_{\mathrm{eff}}} \right\rfloor .} \tag{A.22}$$

Q2B differences. Q2B substitutes the base-equivalent input $x' = \lfloor x \cdot 2^{48}/P \rfloor$ everywhere it appears in $\hat{x}_1$, $\hat{x}_{\mathrm{n}}$, sf_2, and θ_{off}, and uses $\epsilon = -1$ in both the post-trade Δ update and the imbalance numerator (the input adds to quote and removes from base). The activation check uses the raw quote input directly ($y_q = x$, no P-scaling). The final stage merges P with D_{eff} into a single emulated 128-bit divisor:

$$y = \left\lfloor \frac{x \cdot 2^{48}}{\lfloor P \cdot D_{\mathrm{eff}} / 2^{48} \rfloor} \right\rfloor . \tag{A.23}$$

Reference implementation. The full byte-offset parsing and integer-arithmetic implementation is in `scripts/humidifi/quoter.py` of the released repository (`https://github.com/TraceSVM/propamm-simulator`).

A.6 BisonFi

BisonFi has two protocol versions: V2 (Q48 fixed-point, fee denominator 10^5) and V3 (Q88 fixed-point, fee denominator 2,560,000). This section specifies V2; V3 is the protocol reverse-engineered step by step in the worked example of Appendix A.1, and follows the same tiered structure with higher precision and a pre-computed cached fee.

Parameters.

Symbol	Type	Description
P	u64	Q48 fixed-point price (quote/base $\times 2^{48}$)
S	u64	Decimal scale (e.g. 10^3 for SOL/USDC)
v_a, v_q	u64	Stored base/quote reserves
α	u32	Tier price adjustment step (doubled when flag set)
ϕ_d, m_d	u8, u16	Direction-specific fee base and multiplier
ϕ_{stored}	u32	Stored fee component used for cross-directional floor
ρ	u16	Staleness penalty per slot
$s_{\text{pool}}, s_{\text{clk}}$	u64	Slot of last oracle update; runtime clock slot
$\mu_{0,d}$	u32	Direction-specific tier-0 pivot multiplier
$\text{tiers}_d[n]$	struct	Per tier i: $(\text{mult}_i, \text{adj}_i)$ with $\text{adj}_i \in \{-4, \ldots, +4\}$

Step 1: Fee computation. The fee rate (in units of $1/10^5$) is the sum of a base rate, a per-slot staleness penalty, and an optional cross-directional term:

$$\phi = \phi_d + 10\, m_d + \rho \cdot \max(0,\, s_{\text{clk}} - s_{\text{pool}}) + \phi_{\text{cross}}, \qquad x_{\text{net}} = x - \left\lceil \frac{\phi \cdot x}{10^5} \right\rceil. \tag{A.24}$$

The cross-directional term ϕ_{cross} activates only when staleness is positive and the opposite direction's multiplier is non-zero; it floors the total fee at ϕ_{stored} by setting $\phi_{\text{cross}} = \max(0,\, \phi_{\text{stored}} - \phi_{\text{ref}})$, where ϕ_{ref} is the already-accumulated fee (B2Q) or $\phi_{\text{b2q,base}}$ (Q2B). When the oracle is freshly updated ($s_{\text{pool}} > s_{\text{last}}$ and $s_{\text{clk}} = s_{\text{pool}}$), the protocol resets the dynamic tier multipliers and fee parameters to their template values; this discontinuity is sensitive to slot ordering.

Step 2: Price adjustment. For each tier i, the adjusted price and the per-tier capacity (in quote units) are

$$P_i = \left\lfloor \frac{P \cdot (10^4 + \text{adj}_i \cdot \alpha)}{10^4} \right\rfloor, \qquad w_i = \left\lfloor \frac{v_q \cdot \text{mult}_i}{10^4} \right\rfloor, \tag{A.25}$$

with $P_0 = P$ (tier 0 uses the raw oracle price) and $\text{mult}_0 = \mu_{0,b2q}$. Capacities are then converted to base units for consumption:

$$w_{i,\text{base}} = \left\lfloor \frac{w_i \cdot S \cdot 2^{48}}{P_i} \right\rfloor. \tag{A.26}$$

Step 3: Output. The net input is consumed greedily across tiers in increasing order of $|\text{adj}_i|$. Let $x_{\text{rem}}^{(0)} = x_{\text{net}}$ and $x_{\text{rem}}^{(i+1)} = x_{\text{rem}}^{(i)} - \min(x_{\text{rem}}^{(i)},\, w_{i,\text{base}})$. The output is

$$\boxed{y = \sum_{i=0}^{n} \left\lfloor \frac{\min(x_{\text{rem}}^{(i)},\, w_{i,\text{base}}) \cdot P_i}{2^{48} \cdot S} \right\rfloor}. \tag{A.27}$$

If $x_{\text{rem}}^{(n+1)} > 0$ after all tabulated tiers are exhausted, the final tier's P_n is reused for the remainder.

Q2B differences. Pivots come from the base reserves: $w_i^{\text{q2b}} = \lfloor v_a \cdot \text{mult}_i / 10^4 \rfloor$, with $\text{mult}_0 = \mu_{0,q2b}$ and $\text{adj}_i \in \{+1, \ldots, +4\}$ (price worsens for the buyer, so $P_i \geq P$). Capacities are converted to quote units, $w_{i,\text{quote}} = \lfloor w_i^{\text{q2b}} \cdot P_i / (2^{48} \cdot S) \rfloor$, and the output is

$$y = \sum_{i=0}^{n} \left\lfloor \frac{\min(x_{\text{rem}}^{(i)},\ w_{i,\text{quote}}) \cdot S \cdot 2^{48}}{P_i} \right\rfloor . \tag{A.28}$$

When tier-0's effective multiplier is split (the dynamic $\mu_{0,q2b}$ exceeds the post-reset effective value), tier 0 itself is partitioned: an inner segment priced at P and a middle segment priced at P_1, both consumed before tier 1 proper.

Reference implementation. The full byte-offset parsing and integer-arithmetic implementation is in `scripts/bisonfi/quoter.py` of the released repository (`https://github.com/TraceSVM/propamm-simulator`). A worked end-to-end recovery of BisonFi's formula is given in Appendix A.1.

A.7 Tessera V

Parameters.

Symbol	Type	Description
R_d	u64	Direction-specific fixed-point rate ($d \in \{b2q, q2b\}$)
ϕ	u16	Flat fee in parts per million
p_d^{raw}	u64	Raw consumed position (per direction)
c_p	u64	Cross-direction position correction field
$\{(w_i, f_i)\}_{i=0}^{9}$	struct[10]	Per-direction tier table: width w_i and pricing factor f_i

Step 1: Fee computation. A flat fee ϕ is applied to the aggregate output: the pre-fee sum is multiplied by $(10^6 - \phi)/10^6$ with truncation. No per-tier fee component is used.

Before quoting, the effective position per direction is reconstructed from c_p and the two raw counters. Let $\Delta = \lfloor c_p \cdot R_{q2b} / 10^{15} \rfloor$. Then $p_{b2q} = \max(0,\ p_{b2q}^{\text{raw}} - \Delta)$. If Δ exceeds p_{b2q}^{raw}, the b2q position is converted back via R_{b2q} and the residual is added to the q2b position: $p_{q2b} = p_{q2b}^{\text{raw}} + c_p - \lfloor p_{b2q}^{\text{raw}} \cdot R_{b2q} / 10^{15} \rfloor$; otherwise $p_{q2b} = p_{q2b}^{\text{raw}}$.

Step 2: Price adjustment. The per-tier mid-price is precomputed in a separate rounding step, fixing one rounding error per tier:

$$\pi_i = \left\lfloor \frac{f_i \cdot R_d}{10^6} \right\rfloor .$$

Step 3: Output. Tiers are walked in order while a remaining position p_{rem} (initialised to p_d) depletes successive tier widths. For tier i, if $p_{\text{rem}} \geq w_i$ the tier is skipped and $p_{\text{rem}} \leftarrow p_{\text{rem}} - w_i$; otherwise the effective width is $w'_i = w_i - p_{\text{rem}}$ and $p_{\text{rem}} \leftarrow 0$. The trade output is

$$\boxed{y = \left\lfloor \frac{(10^6 - \phi) \cdot \sum_{i=0}^{9} \lfloor \min(x_{\text{rem}},\, w'_i) \cdot \pi_i / 10^{15} \rfloor}{10^6} \right\rfloor} \tag{A.29}$$

where x_{rem} is decremented by the consumed chunk after each tier. Any input exceeding the cumulative tier capacity is capped (silently discarded), not processed at a worse price.

Q2B differences. The Q2B path uses R_{q2b}, p_{q2b}, and $\{(w_i, f_i)\}$ from the Q2B tier table (laid out symmetrically to B2Q in the pool account). The fee, mid-price, and cumulative-depletion logic are identical to B2Q.

Reference implementation. The full byte-offset parsing and integer-arithmetic implementation is in `scripts/tesserav/quoter.py` of the released repository (`https://github.com/TraceSVM/propamm-simulator`).

A.8 ZeroFi

Parameters.

Symbol	Type	Description
$O_{\text{bid}}, O_{\text{ask}}$	f64 (raw bits)	Oracle base→quote and quote→base rates
ϕ_{den}	u64	Primary fee denominator (arb path; Jupiter path uses a lower variant)
$\phi_{\text{den},b}$	u64	Secondary fee denominator
ϕ_{extra}	u64	Additive fee parameter (combined with the per-tier base half)
o_a, o_b	i32	Cranker-updated fee observations
B_a, B_b	i32	Pre-computed observation budget floors
$\phi_{\text{cap},d}$	i32	Per-direction fee cap
$\mathbf{t}[n]$	struct	Tier n: breakpoint bp_n, parameter τ_n, onset pair $(\omega_n^{lo}, \omega_n^{hi})$
ψ_1, ψ_2	u64	Per-slot staleness scaling factors (pre- and post-onset)
$s_{\text{clk}}, s_{\text{pool}}, s_{\max}$	u64	Clock slot, pool last-update slot, staleness rejection gate
v_a, v_q	u64	Base / quote vault balances (output cap)

Step 1: Fee computation. Let the slot age be $\Delta s = \max(0,\, s_{\text{clk}} - s_{\text{pool}})$; the swap is rejected when $\Delta s \geq s_{\max}$. The base fee differential is observation-driven,

$$\Delta\phi \;=\; o_a - o_b \;\;\text{(B2Q)} \quad \text{or} \quad o_b - o_a \;\;\text{(Q2B)},$$

with a constant -150 bias and (for token pools) a clamp $\Delta\phi \leftarrow \max(\Delta\phi, B_d)$. For tier i with parameter τ_i and onset $(\omega_i^{lo}, \omega_i^{hi})$, the staleness multiplier is

$$\mu_i \;=\; \begin{cases} 0, & \Delta s \leq \omega_i^{lo}, \\ \min(\Delta s - \omega_i^{lo},\, N_i)\,\psi_1 + \max(0,\, \Delta s - \omega_i^{lo} - N_i)\,\psi_2, & \Delta s > \omega_i^{lo}, \end{cases}$$

where $N_i = \omega_i^{hi} - \omega_i^{lo}$ when $\omega_i^{hi} > 0$ and $\mu_i = (\Delta s - \omega_i^{lo})\,\psi_1$ otherwise. The per-tier half is

$$\begin{aligned} h_i \;&=\; \left\lfloor \frac{m\,\phi_{\text{den}}}{1000} \cdot \left\lfloor \left(\left\lfloor \tfrac{\phi_{\text{den},b}\,\tau_i}{2000} \right\rfloor + \phi_{\text{extra}} \right) \cdot \tfrac{10^6+\mu_i}{10^6} \right\rfloor \right\rfloor, \\ \phi_i \;&=\; \min\bigl(\Delta\phi^{(i)} - h_i,\; \phi_{\text{cap},d}\bigr), \end{aligned} \tag{A.30}$$

where $m \in \{1, 5\}$ is the large-swap multiplier (set to 5 when the quote-side value exceeds $\sim 4\times10^{15}$ scaled units; 1 on the Jupiter path). Two cumulative budgets track consumption: an input-side budget (initialised from depth minus base vault for B2Q, or the analogous Q2B form) and an output-side budget (initialised from the opposing vault). When budget a or b exhausts at tier i, the corresponding observation is permanently subtracted from the running $\Delta\phi^{(i)}$ and saturated to i32. For B2Q, the effective input is capped at the available depth extra_a_tier_total $- v_a$.

Step 2: Price adjustment. The per-tier correction factor is built in BPF-exact soft float:

$$c_i \;=\; \begin{cases} 1.0 + |\phi_i| \cdot 10^{-7}, & \phi_i \geq 0, \\ 1.0 - |\phi_i| \cdot 10^{-7}, & \phi_i < 0, \end{cases}$$

with sign determined by ϕ_i alone (independent of swap direction). The factor $|\phi_i| \cdot 10^{-7}$ is computed via the soft-float multiply described below, and $1.0 \pm \cdot$ uses the IEEE-754 double result.

Step 3: Output.

$$\boxed{\; y \;=\; \min\Bigl(v_d,\; \textstyle\sum_i \text{bpf_f64_to_int}\bigl(c_i \cdot O_d \cdot \min(x_{\text{rem}},\, \text{bp}_i)\bigr)\Bigr) \;} \tag{A.31}$$

where $O_d \in \{O_{\text{bid}}, O_{\text{ask}}\}$ selects the directional oracle, and the tier sum walks $i = 0, 1, \ldots$ consuming bp_i at each step until x_{rem} falls below the next breakpoint (the residual is priced at the next tier's parameters). The composite operator bpf_f64_to_int$(c_i \cdot O_d \cdot x)$ replicates two sBPF VM quirks:

1. *Soft-float multiply.* Each f64 multiplication is performed on full 53-bit mantissas with three additional rounding bits. The result is normalised to 52 explicit mantissa bits; if the three rounding bits exceed 4, the mantissa is incremented by one, otherwise it is truncated. This deviates from IEEE-754 round-to-nearest-even at exact halfway points.
2. *Conversion to integer.* The resulting f64 is converted by extracting the (implicit-1) mantissa, shifting by $\exp - 52$ (right-shift if negative, left-shift if positive), and truncating toward zero with no further rounding.

Q2B differences. The tier breakpoint set switches: B2Q uses $\mathbf{bp}_i^{(\mathrm{alt})}$ stored in the cumulative-output table, while Q2B uses the dedicated table $\mathbf{bp}_i^{(q)}$ at the Q2B layout offsets. When the cranker-written gate slot equals s_{clk}, the corresponding direction's breakpoints are reduced by per-tier adjustments adj_i. The oracle selection is $O_d = O_{\mathrm{bid}}$ for B2Q and $O_d = O_{\mathrm{ask}}$ for Q2B; the budget assignment of (o_a, o_b) to input- vs. output-side trackers is also direction-dependent.

Reference implementation. The full byte-offset parsing and integer-arithmetic implementation (including the soft-float helpers) is in `scripts/zerofi/quoter.py` of the released repository (`https://github.com/TraceSVM/propamm-simulator`).

List of Abbreviations

AMM Automated Market Maker

BPF Berkeley Packet Filter

CFG Control Flow Graph

CFMM Constant-Function Market Maker

CLMM Concentrated Liquidity Market Maker

CPI Cross-Program Invocation

DAG Directed Acyclic Graph

DDG Dynamic Dependence Graph

DeFi Decentralised Finance

DEX Decentralised Exchange

DTA Dynamic Taint Analysis

ELF Executable and Linkable Format

EVM Ethereum Virtual Machine

IL Impermanent Loss

IPDOM Immediate Post-Dominator

LP Liquidity Provider

LVR Loss-Versus-Rebalancing

MEV Miner Extractable Value

PnL Profit and Loss

propAMM Proprietary Automated Market Maker

RPC Remote Procedure Call

sBPF Solana BPF

SOL Solana

SPL Solana Program Library

SVM Solana Virtual Machine

TVL Total Value Locked

USDC USD Coin

VM Virtual Machine

XOR Exclusive OR